A GRAPHIC TOOL FOR GENERATING ADA LANGUAGE SPECIFICATIONS

by

DONALD E. BODLE, JR.

B.S., Kansas State University, 1984

A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

Approved by:

William Hankley
Major Professor

A Graphic Tool for Generating Ada Language Specifications

LD
2668
.T4
1985
B62
C.2

A11202 942306

by Don Bodle

Abstract

Methods for specifying software systems have gained increasing attention as the size and complexity of computer applications has grown. The purpose of this paper is to present the current state of software specification techniques and to propose improvements in one component of these techniques, the user interface.

The use of automated tools for specification is described, with particular emphasis on their user interfaces. Many features of these tools are highlighted. From this study, a proposal for a graphic interface for software system specification is developed, describing the desirable features of such an interface. Finally, a prototype of the proposal is examined.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. OVERVIEW

Methods for specifying software systems have gained increasing attention as the size and complexity of computer applications has grown. The purpose of this paper is to review the current state of software specification techniques and to propose improvements in one component of these techniques, the user interface.

Basic background information on requirements specifications is provided in Chapter 2. It presents a summary of characteristics of specifications and then focuses on some of the formal models used as a basis for requirements specifications. The chapter also discusses the varieties of requirements specification languages.

In chapter 3, methodologies such as Higher Order Software (HOS) (Hamilton, 1976; Hamilton, 1983), Program Statement Language/ Program Statement Analyzer (PSL/PSA) (Teichroew, 1977), Technology for Automated Generation of Systems (TAGS) (Sievert, 1985), and Software Requirements Engineering Methodology (SREM) (Alford, 1985) are reviewed for their contributions to automated requirements specifications. Additionally the tool Gambit (Braegger, 1985), though not a specification tool, is reviewed for its graphic interface features.

The main contribution of this paper, a model for a graphic tool for generating Ada language specifications, is described in Chapter 4. This model draws on some of the concepts of the tools described in Chapter 3 and adds ideas such as "direct manipulation" and "spatial management" (Schneiderman, 1983).

Chapter 5 presents a prototype of the interface model. The prototype is written in Turbo Pascal using the Turbo Graphix Toolbox. This implementation is a limited demonstration of the ideas in the developed model. The program allows drawing and deleting of objects and directed arcs and naming and specifying procedures and their inputs and outputs for each object. It automatically modifies the underlying data structure corresponding to graphic actions. The program will create Ada language specifications from the graphic specification, and allows saving a display file on disk which can be retrieved and further edited.

Chapter 6 is used to evaluate the model and the implementation. It also presents recommendations for extensions to the model and further work in the area of graphic interfaces.

1.1 Requirements Specifications

One of the many steps in software engineering between

problem recognition and problem solution is describing the problem. As software systems became more complex, more formal steps were defined between recognition and solution. In the "traditional" life-cycle, the steps include requirements analysis and definition, specification, design, programming, verification and testing, performance, operation and maintenance, and configuration management (Myers, 1978). Requirements specifications consisted of hand-drawn data flow diagrams, hierarchy diagrams, control structure diagrams, or data structure diagrams (or any combination of these). Added to these were text specifications, usually functional in nature, and data dictionaries to precisely describe the structure and usage of data.

More recently a life-cycle model called the functional life-cycle has been offered, with four phases: define, analyze, resource allocate, and execute (Hamilton, 1983). Again, a combination of graphic and textual components are used to define the system to be developed. The major difference with this model has to do with the steps between requirements specification ("define") and an executable software system.

With the Department of Defense-sponsored development of the Ada programming language, some concept of specifications has entered directly into a high level language (DOD, 1983)

(Booch, 1983). Functional components in the Ada language consist of two separate parts, a specification part and a body. The specification part describes the interface to the component but none of the implementation details. This follows the basic idea accomplished in other specification methods, describing the "what" rather than the "how" of system components. The implementation or the "how" of the components can be developed at a later time. Therefore, the entire software system can be described using these specification parts and these specifications handed out to many different implementors to be coded.

1.2 Levels of Specification

The purpose of a requirements specification is to describe as accurately as possible the elements of the problem to be solved. These elements include the information to be processed, the functions which are to be accomplished, and the operating constraints under which the processing is to take place. Most often the requirements are stated at different levels of refinement. Each successive level is a refinement or decomposition of the components of the previous level.

One example of such refinement is seen in Yourdon's analysis of a data flow diagram for a system. The diagram is divided into the afferent, transform, and efferent components

(Pressman, 1982). This is the first level of refinement and is more readily understood as input, process, and output. These three components are then each refined into their logical components, and this process is repeated until a component is a single-function, coherent, easily understood unit.

1.3 Graphic Interfaces

Requirements specifications gained importance as software systems became larger and more complex. Initially they existed as flowcharts, data flow diagrams, or other individually-styled picture representations of the software system. These were drawn by hand, and required text specifications to correspond to them. Since these pictures were non-standard, much confusion arose when someone different than their creator was required to code the system. Text specifications were helpful, but often incomplete or ambiguous. This resulted in software systems that did what the specifications required but not what was really wanted.

In efforts to more formally and accurately describe system requirements, new methodologies and formal languages have been developed. These require designers to learn the language syntax and then try to express the system in that language. Since "a picture is worth a thousand words" and

managers don't have time for a thousand words, various styles of printed graphic representations are generated from the specification.

As interactive graphics hardware and software have improved, tools to use these capabilities are being developed. At least one automated tool allows interactive, graphically developed system specification.

Requirements specification has moved from manual graphic representations with details textually specified, to computer analyzable formal specification languages with graphic diagrams produced after the formal specification, to interactive graphic specification with a corresponding text specification.

1.4 The Problem with Tools

Commonly used specification methods begin with diagrams and then add the details. Typically, the first diagram pictures the entire software system as a few major components, often the interfaces to the external environment. This diagram is decomposed into its components, and each resulting diagram is similarly decomposed until the components become cohesive, single-process untis. During or after the decomposition, the details about inputs, outputs, and other information required for the specification are added. The various earlier automated tools either did not allow

designers to work from a graphic representation to detailed specification, or did not allow easy transition from one form to the other.

Of the five automated tools presented in Chapter 3, SREM, TAGS, AND PSL/PSA provide a graphic representation of the software specification once the text specification has been entered. Since designers often like to pictorially define the problem to be solved before adding details, these tools don't help in this area. Many designers are likely to draw by hand the initial breakdown of the problem and then specify it in the requirements statement language of the tool they are using.

HOS now provides interactive, graphic decomposition of the system specification through its USE.IT tools (Hamilton, 1983; Martin, 1985). The recent addition of these tools moves HOS into the arena of "direct manipulation" and addresses many of the issues of graphic user interfaces.

Gambit implements many of the graphic interface features recommended in the model presented in Chapter 4. Unfortunately, this is a database design tool and is not useful in non-database applications.

1.5 A Model for a Graphic Tool

The desire to "physically" manipulate a software system model (graph) and at the same time correspondingly manipulate the text specification of the system has motivated the design of a Graphic Tool for Generating Ada Language Specifications (GTGALS). GTGALS allows the user to create or modify a graphic representation of a software system (see figure 1.6.1) and its corresponding text specification. (see figure 1.6.2)

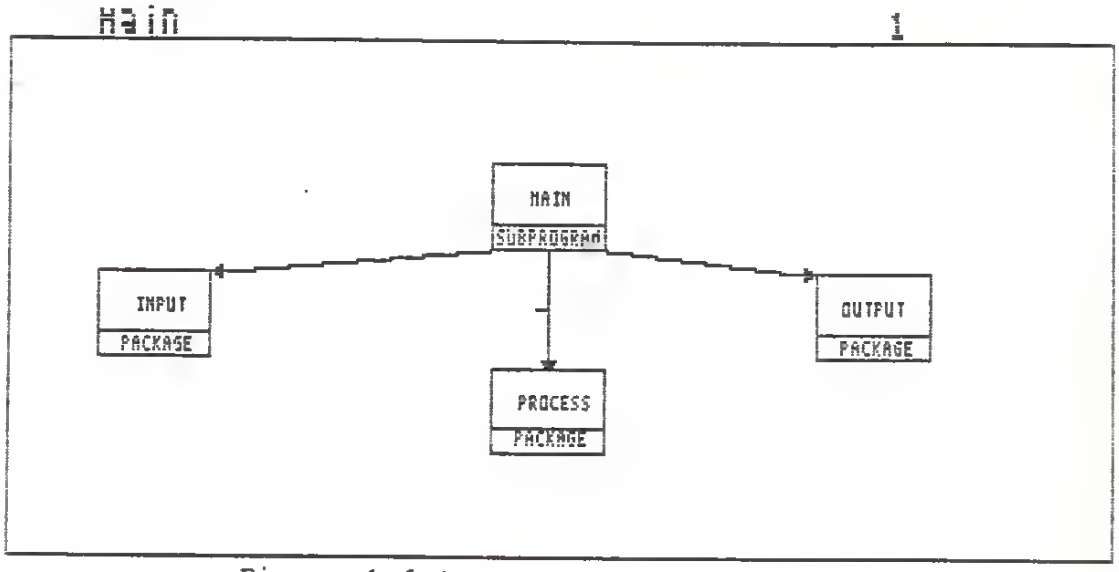


Figure 1.6.1 - GTGALS Access-graph

```
--This is the controller
with process,
     input,
     output;
procedure main(in_msg : in msg_packet;
                    out_msg : out msg_packet);


--This package handles all data modification
package process is
--This procedure breaks the incoming message
--packet into its components
   procedure split_msg(in_msg : in msg_packet;
                       out_char : out character;
                       out_int : out integer;
                       out_string : out string;
                       out_float : out float);
--returns the base ten ascii equivalent
--of the character it is applied to
   function ascii(any : in character)
                       return integer;
end process;

--This packages interfaces to the "outside world"
package input is
--for reading entire message packets
   procedure read_msg(got_msg : out msg_packet);
end input;

--This handles output intefacing to environment
package output is
--Writes the message to the standard output file
   procedure write_msg(in_msg : in msg_packet);
end output;
```

Figure 1.6.2 - Ada language specification of 1.6.1

Direct creation and manipulation of a graph and its related data structure is a primary feature of GTGALS. Drawing and deleting objects, specifying their procedures, inputs and outputs, designating relations between objects using directed arrows, viewing and modifying component

specifications from the graph, and receiving both a graphic and text representation of the software system specification are the key functions of GTGALS. The GTGALS model is presented in detail in Chapter 4, with a prototype implementation presented in Chapter 5.

CHAPTER 2. SPECIFICATIONS

Specifying software systems is a current topic of software engineering courses, publications, and textbooks. This chapter summarizes answers to many questions about software specifications. These questions include : what should be specified?; what characterizes good specifications?; what areas are used for comparing specification techniques?; what formal bases are used in specifications?; and how are specifications expressed?

The majority of this information comes from a survey by Roman (1985). The subject is also covered in textbooks such as Pressman (1982) and chapter two of Gilbert (1983), and a paper by Balzer (1979).

2.1 Types of requirements specifications:

2.1.1 Functional

Functional requirements describe what the software system is supposed to do based on the interaction between the system and its environment. The model of description has been called a conceptual model. These requirements are an abstraction of the problem to be solved.

2.1.2 Non-functional

Non-functional requirements describe under what constraints the software system is required to operate. Some of these constraints include interface constraints, performance constraints, operating constraints, life-cycle constraints, economic constraints, and political constraints.

2.2 Characteristics of specifications

Several characteristics of specifications have been identified in the attempt to define what comprises a good specification. One such collection of these characteristics is summarized here. (Roman, 1985)

Adaptability - can it represent many classes of problems

Analyzability - how well can the specification be analyzed for the characteristics described here

Appropriateness - how accurately can the model represent the problem domain

Completeness - are all relevant aspects of the problem domain covered

Conceptual Cleanliness - how readily understandable is the resulting specification

Consistency - are none of its parts contradictory

Constructability - what (if any) systematic approach for developing the specification is provided

Easy modifiability - how can it be changed, and with what results

Economy of expression - what are its storage requirements

Executability - can the specification be machine processed for simulation of design

Formality - to what extent is machine processing possible

Lack of ambiguity - can the specification be interpreted in only one way

Precision - can it be determined that the design meets the specification

Testability - can the design be verified as meeting the specification

Tolerance to temporary incompleteness - can the technique handle incompleteness in the specification

Traceability - can the requirements specification be cross-referenced with the design specification

2.3 Areas for analysis

Along with characteristics of specifications, certain areas have been used as a basis for analyzing and comparing different specification methodologies.

2.3.1 Formal model

The formal model is the conceptual model on which the specification methodology is based. A description of many of these models follows in 2.4.

2.3.2 Scope

Scope describes the type of requirements the methodology attempts to express. This could be functional only, non-functional only, or a combination of functional and non-functional requirements.

2.3.3 Level of formality

The level of formality of a methodology determines the machine processability of the information. The more formal and well defined the language of specification, the greater the opportunities for automated analysis of the specification.

2.3.4 Degree of specialization

The degree of specialization describes the size of the problem domain that can be expressed in the methodology.

2.3.5 Specialization area

The specialization area defines the type of requirements that the methodology can express. This could include database models, sequential process models, or concurrent process models. From a different view, this could also describe whether the methodology can be used for hardware, software, organizations, or some combination thereof.

2.3.6 Development method

This area includes both how the information is collected and managed, as well as under what basic life-cycle model it fits.

Traditional	- state requirements completely before proceeding with design
Rapid-prototyping	- build incrementally, simulate, and redesign "on the fly"
Mixed	- combination of stating requirements and prototyping

Human-interface - how the information is made accessible to the tool and the user.

2.4 Formal Models of Specifications

Formal models of specification are models by which various individuals have described software systems. (These models have been used to describe much more than just software systems. However, the emphasis of this paper is on software applications of the models.) Either sufficient study and formalization, sufficient publication, or sufficient application of a model establishes it as a "formal" model. Each model attempts to describe a problem in such a way as to make it easy to visualize the components and structure of the problem. The formal models discussed below are various perspectives on how to describe a software system and its environment.

2.4.1. Access-graph model

An access graph shows the various components within a software system and their "access rights". Each component will have directed arcs connected to those system components which it is allowed to use. This model easily relates the concept of composition, building a software system by giving

new control modules access to already constructed library modules. In the Ada programming language, this model would graphically describe the with clauses of the components. In C-Pascal, access graphs describe the access parameters of processes, classes, and monitors (Hansen, 1977). Figure 2.4.1 shows a simple access-graph diagram.

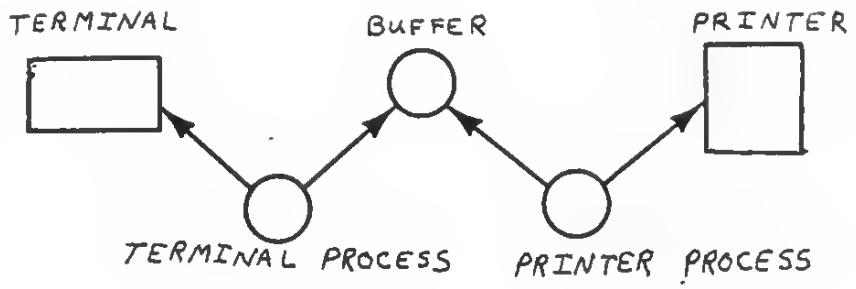


Figure 2.4.1 - An Access-graph

2.4.2. Communicating concurrent processes

This model describes a system as a collection of components which run concurrently. Each component is seen as an independent object and is described by its interaction with the environment and the processing done based on the interaction. Interaction occurs through communication "ports" as data input from the environment and data output to the environment.

2.4.3. Data flow

Data flow diagrams, or similarly requirements diagrams, describe a system as a collection of processes (transformations) and their connections (data). A top level diagram shows the entire system as one process, and its interaction with the environment as arcs representing data flow in and out of the system. Each level is decomposed until a process represents a logical functional unit. Each process and arc is labeled, and further detailed in detailed specifications, data dictionaries, and other documentation. Figure 2.4.2 provides an example of a simple data flow diagram.

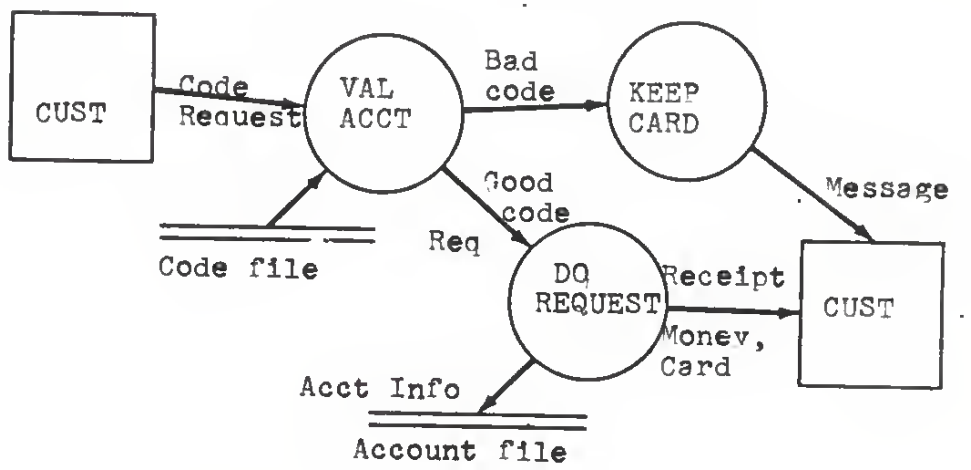


Figure 2.4.2 - A Data flow diagram

2.4.4 Entity relationship model

The entity-relationship model describes a system by its data entities and the relationships between those entities (Ullman, 1982). Rather than looking at processes and sequences of processing, the E-R model is data oriented. Since it is a model for database applications, it is assumed that all necessary processing can be accomplished if the data is properly related. Therefore, an E-R diagram will show nothing of the processes accomplished. However, it is a useful model for conceptualizing a database design. Figure 2.4.3 shows a sample E-R diagram.

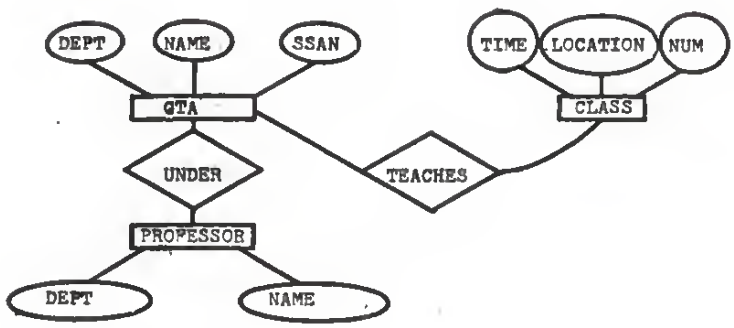


Figure 2.4.3 - An E-R diagram

2.4.5. Finite-state machines

A finite-state machine expresses a software system as a finite number of states and a set of transition functions. In general, the machine will begin in some known state. A

change in states (a transition) is caused by some input, and can produce some output. The new state is determined by the old state and the input. Finite-state machines are readily represented graphically. Figure 2.4.4 shows a sample finite-state machine.

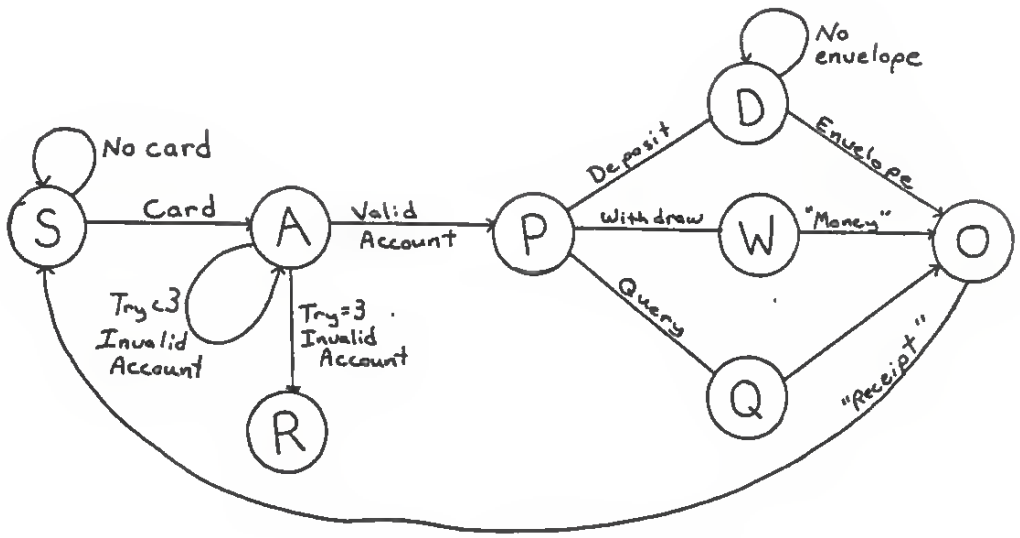


Figure 2.4.4 - A finite-state machine

2.4.6. Functional composition

In functional composition, a system is a composition of hierarchically subordinate functions. Graphically a tree structure, each parent is a function which is a composition of its children (also functions). Procedurally, each parent uses its children to accomplish its task. This is recursive, so that all of the functionality of the system is accomplished at the leaf nodes of the tree.

2.4.7. Petri nets

A Petri net describes a software system as a collection of places and transitions (Peterson, 1981). Petri net graphs include directed arcs connecting the places and transitions, indicating inputs and outputs of the places and transitions. The sequence of processing from inputs to outputs is defined by the "enabling" and "firing" of transitions within the net. A transition fires when it has available to it all of its inputs. This model is similar to a finite-state machine model, describing a system's current state and a next-state function to describe the results of inputs into the system. Figure 2.4.5 is a sample Petri net graph.

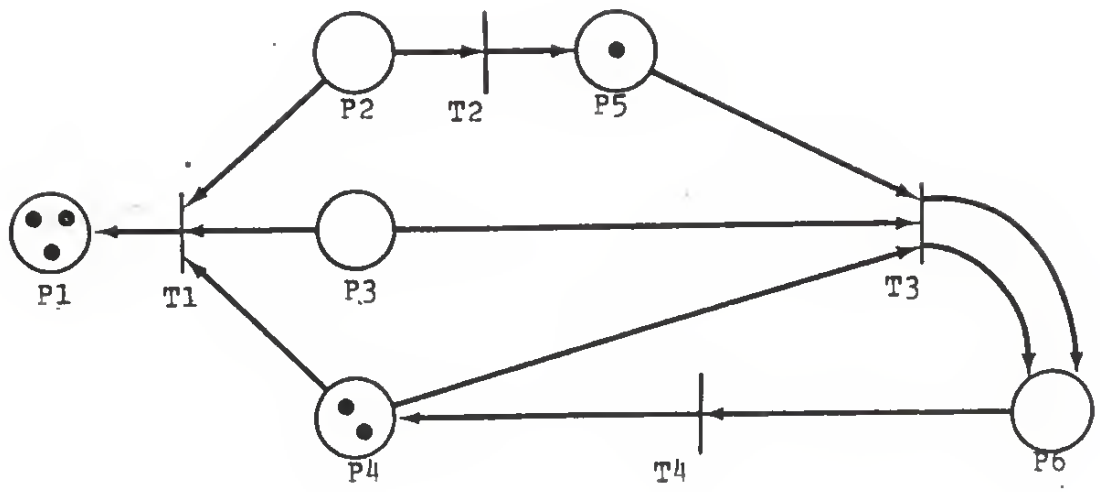


Figure 2.4.5 - A Petri net graph

2.4.8 Stimulus response paths

This model is almost indistinguishable from the finite state machine model. In fact, Roman (1985) attributes its success to SREM, whereas Alford (1985) writes that "The model of software requirements on which SREM is based is that of a highly structured finite state machine."

Many different methods have been used to express the various formal models for human and/or computer consumption. These methods, or languages, have included requirements diagrams, requirements statement languages, requirements specification documents, and many other methodology-specific languages.

2.5 Specification Languages

Though the term language causes one to think of letters, words, and sentences, the language of specification includes drawings as suggested by the formal model of the specification methodology. Requirements diagrams, data flow diagrams, state-machine diagrams, and so on exist for each model and more. Probably the earliest, albeit low-level, specification language was the flowchart. In general, designers like graphic representations of problems and their solutions.

Prior to computer generated graphics, and even with the availability of such graphics, diagrams have been created by

hand. As computer graphics capabilities have increased significantly both in hardware and software, the use of computer generated diagrams has slowly moved into the area of software engineering and analysis (Grafton, 1985; Jacob, 1985; Brown, 1985; Schneiderman, 1983).

CHAPTER 3. AUTOMATED TOOLS FOR SPECIFICATION

Many methodologies have been developed to help formalize, visualize, analyze, and process software specifications. Five sample systems are detailed in this chapter.

Four methods designed specifically for describing software systems are examined for their features, focusing primarily on their formal models, user interfaces, and outputs. These are HOS, PSL/PSA, SREM, and TAGS. A fifth tool, Gambit, is used for data base design. It is examined especially for its graphic interface features. These systems are presented here in alphabetic order.

3.1 Gambit - (Braegger, 1985)

Though Gambit is not specifically a requirements specification tool, it provides many features which are significant for this paper. Among these features are graphic model design of entities and relationships; interactive entry of data attributes; logical, automatic manipulation of data from actions taken to the graphic model; and access to data from the graphs.

The purpose of Gambit is to aid in the design of a database schema. This process requires analysis of the enterprise's data, discovering the requirements of the database (both functional and non-functional), and organizing the information into a logical structure.

3.1.1 Formal model - extended entity relationship model

A database model is largely concerned with the data to be manipulated and the relationships between data groups (or entities). The functional aspect of the system is more a peripheral issue and the data organization and accessibility is expected to support any reasonable application program. The entity-relationship model groups data items as attributes of entities, and then describes the relationships between the entities.

3.1.2 User Interface

The user interface for Gambit has many useful features. Designed for use on a single-user Lilith personal computer, it offers graphic design of entity block diagrams, mouse movement of a marker for object selection and placement, windowing for data retrieval, a "dialogue" section on the screen for interactive entry of necessary information for the design, and menu selection of different steps in the design process.

Entity block diagrams consist of rectangles to represent entities, lines to represent relationships, and text labels to indicate names, associative cardinalities, and other descriptive information. (see figure 3.1.1)

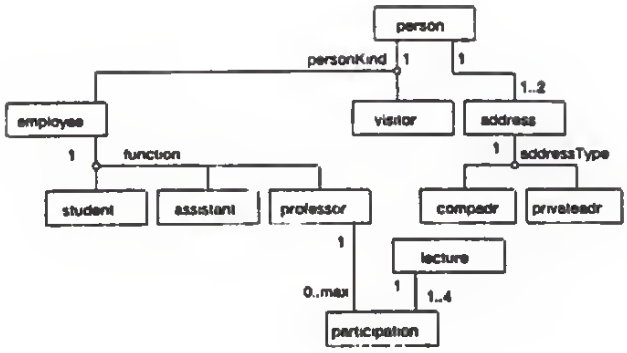


Figure 3.1.1 - An Entity Block Diagram
(Braegger, 1985 - IEEE TOSE)

After menu-selecting the operation to define an entity set, the system provides the designer with a triangular marker. Moving the mouse to position the marker, the designer types in the name of the entity set at its desired location. Gambit then draws the rectangle around the name and initiates a uniqueness check on the name. The designer then steps through a dialogue, providing information about the entity set as requested (data entry may be temporarily bypassed). Menu-selecting the operation to define a relationship starts a dialogue to describe the entities involved, and other information. Gambit then does the appropriate line drawing and labeling. (see figure 3.1.2)

At any point in the design process, the designer "may see a global entity block diagram with all entity sets and relationships defined, or the verbal specification of one entity set with all details,..." In defining global

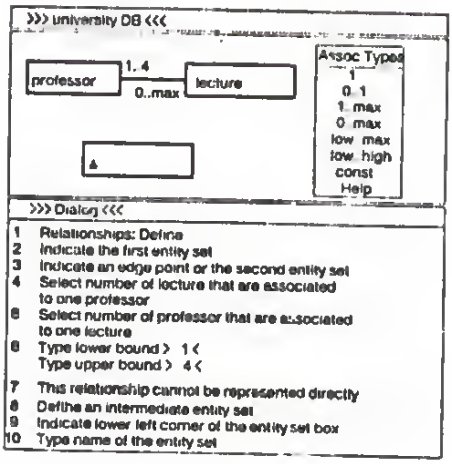


Figure 3.1.2 - Defining relationships with Gambit
(Braegger, 1985 - IEEE TOSE)

attributes, the designer points at an entity set. Gambit then provides a window for the description of the entity set. It automatically retrieves identification attributes from other entities related to the chosen set, and interactively allows attribute renaming or maintaining the same name for local use in the entity set being specified.

3.1.3 - Output

Once a design session has been completed, Gambit generates an entity block diagram and the Modula/R database module containing the details concerning the entity sets. Further interaction allows defining of data constraints, transactions, some transaction pre-assertions, and

transaction propagation. This information is used to build database access modules through which interactive users and application programs must access the database.

3.1.4 - Observations

Key concepts of graphic interfacing to design tools are applied in Gambit. The ability to start with a graphic model and add details later is a major step in the natural design direction. Use of a mouse to touch entities for data retrieval, to position a marker for graphic object placement, and for menu selection is a very "user-friendly" feature. Easy movement from graphic representation to textual description and back is another desirable feature of Gambit.

The limitation of Gambit to design of Modula/R databases is an unfortunate one. Databases are not the answer to all software requirements, and the availability of a software design tool such as Gambit would be an aid to other software design. Also, the limited documentation provided by Gambit may not be considered sufficient for a system specification.

3.2 HOS - Higher Order Software - (Hamilton, 1976)

Higher Order Software is a methodology based on mathematical functions. A set of tools called USE.IT has been developed to automate much of the HOS methodology (Hamilton, 1983;

Martin, 1985). These tools operate with the HOS design "laws" enforced so that the resulting design obeys HOS methodology axioms.

3.2.1 Formal model - functional decomposition

HOS is based on a hierarchical decomposition of functions, in particular mathematical functions. One function represents the entire software system, with input as the domain of the function and output as the range of the function. This function is decomposed into subfunctions. This decomposition is iterated until each leaf of the functional tree provides "one and only one element of the output set for a particular element of the input set." (Hamilton, 1977)

3.2.2 User Interface

The HOS methodology is supported by USE.IT, a set of tools developed to support the functional model of the software life-cycle. The first phase of that life-cycle model is definition, roughly equivalent to specification in the traditional life-cycle.

The tool most significant for this paper is the graphic editor and its use of the specification language AXES (Martin, 1985). The graphic editor operates on three different images. The "display tree" mode provides an

overview of an HOS tree. From this mode, one can move to a detailed representation of a selected node in the "edit" mode. At this point the user can edit any of up to six nodes centered on the selected node. Moving off-screen results in a new screen with the node moved to as the center of the diagram. The user can also move to a "display documentation" mode which shows details and allows editing of a textual description of the selected node.

The graphic images are annotated with the language AXES, which details control structure and data for each node. Data named on the left of a node is output data, that on the right is input data. Abbreviated control structures are displayed at the bottom of each node. An un-connected vertical line going out of the bottom of a node indicates that more of the HOS tree exists beneath that node.

The user interface is currently under improvement to include mouse control, windows, pop-up menus, and other similar "user friendly" features.

3.2.3 Output

The HOS methodology develops sufficiently formal output that automatic generation of program code is possible. This is a result of the strict design laws enforced by the methodology and decomposition to the levels of detail necessary for code generation.

3.2.4 Observations

The addition of the USE.IT tools to the HOS methodology may increase its popularity. No longer restricted to manual drawing of HOS trees of mathematical functions, the USE.IT tools are rapidly moving in the direction of a natural, relatively easily used method for rapidly specifying software systems.

3.3 PSL/PSA (Teichroew, 1977)

PSL/PSA combines a Problem Statement Language (PSL) with a Problem Statement Analyzer (PSA) to develop and analyze systems specifications. Its purpose is to record in machine readable form the data collected or developed during the entire software life-cycle. These activities are grouped into data collection, analysis, logical design, evaluation, and improvements. PSL is the language used to describe a proposed system, and may be used in batch or interactive environments.

3.3.1 Formal model - "a general system" model

The general system model is very similar to the entity-relationship model, and is specialized for information system processing applications. It contains objects (entities and processes), properties (attributes), and relationships between objects.

3.3.2 User Interface

The Problem Statement Language is the form into which specifications are developed. The designer translates the data collected through personal contact, interviews, forms analysis, and other standard methods of collection into the Problem Statement Language. This can be done either interactively or with batch processing in text format only.

3.3.3 Output

The Problem Statement Analyzer produces four basic classifications of reports. Database modification reports record changes made in the database and any resulting diagnostics or warnings. Reference reports provide various ways of formatting the database information into human-consumable products. Summary reports provide similar information only in summary form. Analysis reports do I/O comparisons, process interactions, and a hypergraphic process flow chart.

3.3.4 Observations

Though any automation is a great improvement over manual specification, more could be done with PSL/PSA. Its major benefits are providing automated means of maintaining documentation throughout the software life-cycle. This is done by recognizing that most documents are simply different

ways of expressing all the available information or different levels of abstracting summaries of the available information. That graphic representation of the information is useful is reinforced by the presence of a tool to provide such a representation, even if it is a rather crude printer-character graphics method. Unfortunately, this comes at the end of the specification process, showing what has been accomplished. It is likely that many, if not most, users of PSL/PSA manually produce an E-R diagram, or some similar diagram, of the system to aid them in developing the PSL representation of the system.

3.4 SREM (Software Requirements Engineering Methodology) (Alford, 1985)

SREM was sponsored by the Ballistic Missile Defense Advanced Technology Center in 1973 to formalize and automate development of software requirements specifications. It consists of a Requirements Statement Language (RSL), the Requirements Engineering Validation System (REVS) (a set of tools to manipulate RSL and analyze the resulting system), and the SREM methodology.

3.4.1 Formal model - finite state machine

The developers of SREM felt that the hierarchy of functions

model of specifications was a primary cause of inadequate requirements specifications. They chose to use a finite state machine model to base SREM on. "The state-machine model is used to define processing requirements by specifying a set of inputs and outputs, a set of states, and a function that maps inputs plus current state onto outputs plus updated state." To overcome some of the limitations of a finite state machine, particularly the size of the diagram of large systems, SREM structures its inputs, outputs, state, and processing.

Inputs and outputs are structured as message packets which contain the data that passes between subsystems. States are defined by sets of information about objects in the system. The processing is described by Requirements networks (R-nets). An R-net "specifies the transformation of a single input message plus current state into some number of output messages plus an updated state."

3.4.2 User Interface

The requirements specification is developed in RSL, SREM's Requirements Statement Language. It consists of elements (nouns), attributes (adjectives), relationships (verbs), and structures (processing graphs). All of these items are maintained within a database.

The specification is described by its elements, each of

which have attributes (such as name). The elements are connected by different types of relationships. The processing sequences are expressed through its R-net and subnet structures.

This information is currently entered using simple text-editing methods. The graphic portions (R-nets and subnets) have language counter-parts, (see figure 3.4.1) which are then translated into graphic representations by one of the tools in the REVS.

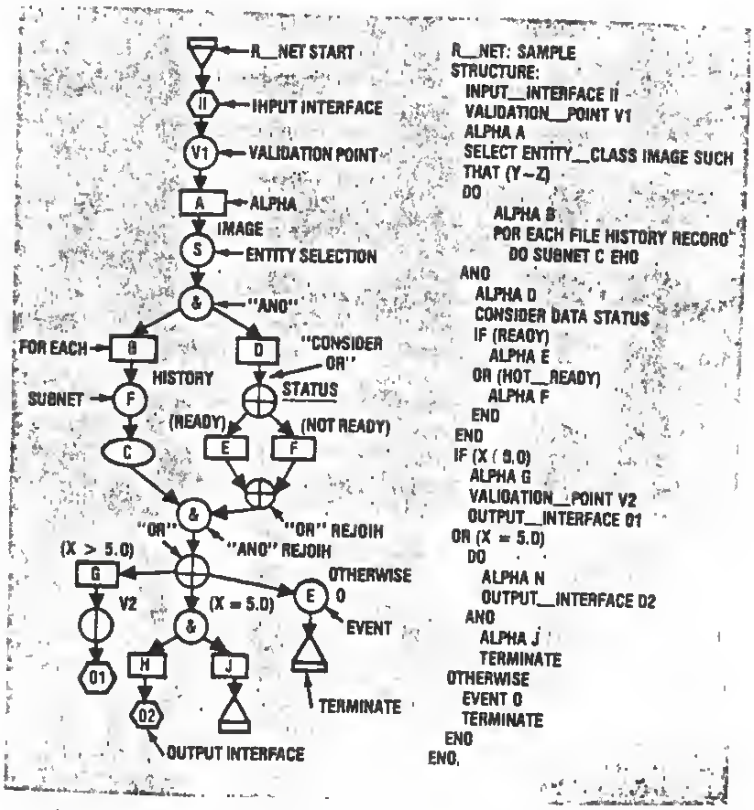


Figure 3.4.1 - An R-net graph and text (Alford, 1985 - IEEE Computer)

3.4.3 Output

Among the outputs of REVS (the SREM support tools) are:

The automated database from the RSL
Consistency and completeness reports
Query type output of the data
Functional or analytical simulator of required processing
Graphical descriptions of the R-nets and subnets

3.4.4 Observations

SREM provides a method for formally describing requirements specifications. Its formality allows many diagnostics to be computer generated, and allows for concise expression of the requirements. Also, it maintains information in a database, allowing relatively easy retrieval.

As one of the older software engineering tools, SREM depends heavily on text-editing input. This input is then translated into graphic representations once complete. Although an interactive forms-entry capability is under development, the system still progresses from textual details to graphic descriptions. Going from a graphic, conceptual model of a system to later filling in the details seems a more natural method of development.

3.5 TAGS (Technology for the Automated Generation of Systems) (Sievert, 1985)

Software specification is just part of TAGS, a complete software development methodology that covers the entire software life-cycle. The specification phase is accomplished through use of its Input/Output Requirements Language (IORL), which consists of graphs and data tables. Using a graphics workstation, the designer expresses the user-supplied requirements in IORL. Four tools are available for use to aid the designer.

The Storage and Retrieval tool is used for data management, placing the design into disk files and accessing the data as required. A Diagnostic Analyzer checks for static errors such as syntax errors, range errors, input/output inconsistencies, and some 200 other types of errors. Once past the Diagnostic Analyzer, the Simulation Compiler finds any dynamic errors. When successfully compiled, the designer can interactively describe a system state on which the compiled system prototype can execute. Any errors detected along any step of the process can be corrected using the Storage and Retrieval tool, and the process continued. Finally, a configuration manager helps keep the various releases, test versions, and associated diagnostic outputs under control.

3.5.1 Formal model - communicating concurrent processes

The formal model on which this system is designed is communicating concurrent processes. This model allows the specification to naturally handle systems that require concurrent processing as well as sequential processing. The "end product of the design effort manifests the basic components of a system or a group of parts that interact through data links, a controlling mechanism that directs how information passes among the parts of the system, and an identified hierarchy within the system."

3.5.2 User interface

The specifications are represented through the use of IORL, the Input/Output Requirements language. This language combines graphic diagrams to show the systems structure and tables to detail the data. Graphic workstations are used to develop the elements of the language, which are described below.

DIAGRAMS - each diagram has the system name, date, id, section, and page

SBD - the Schematic Block Diagram is the highest level diagram. It shows the major components of the software system, with the first level SBD usually diagraming the system with its

environment. If necessary, the top level SBD can be decomposed into lower level SBD's. The primary function of the SBD is to give a conceptual view of the system, and is useful for seeing a quick synopsis of the design. It describes the major structures of the system and its major data flow.

- see figure 3.5.1

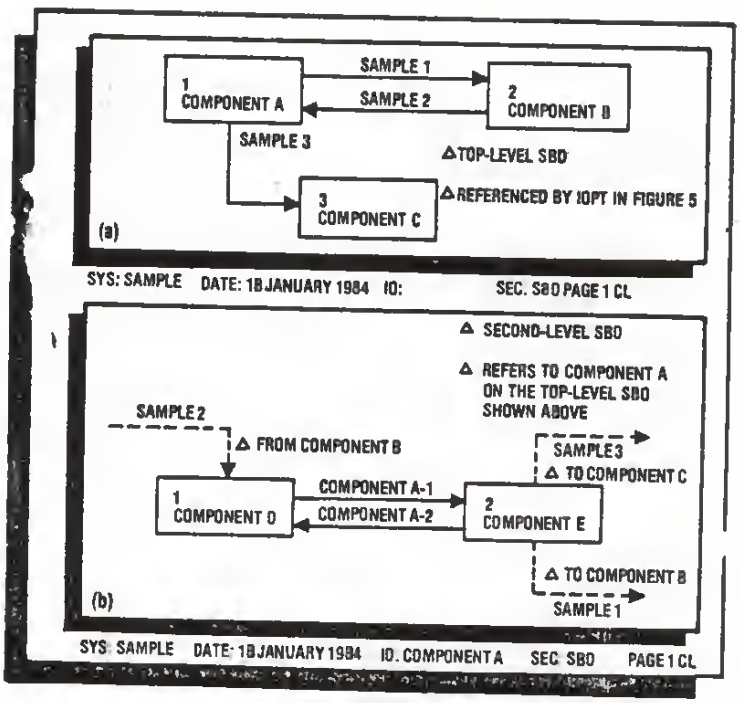


Figure 3.5.1 - A Schematic Block Diagram (Sievert, 1985 - IEEE Computer)

IORTD - each component of an SBD has an associated Input/Output relationships and timing diagram to show control flow within that SBD component.

- see figure 3.5.2

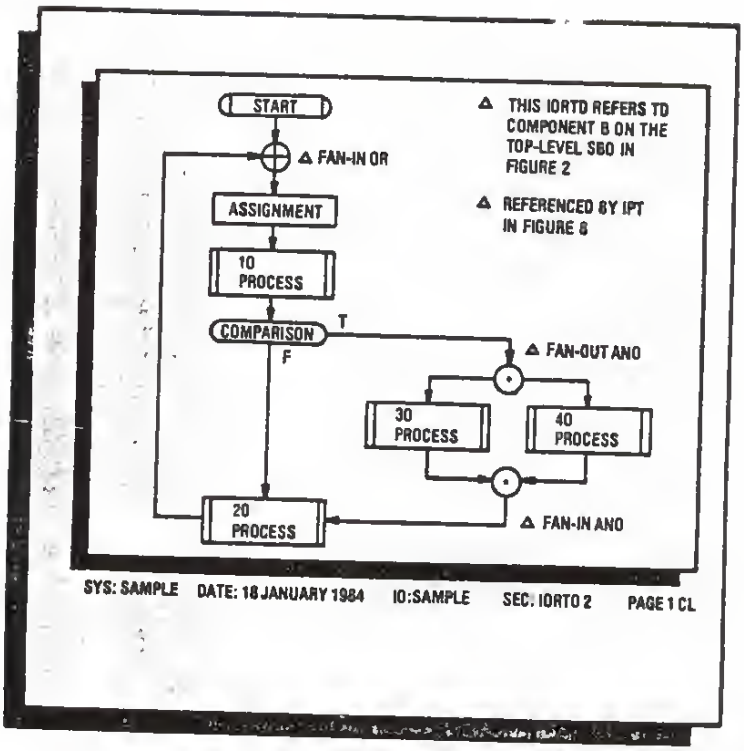


Figure 3.5.2 - An IORTD
(Sievert, 1985 - IEEE Computer)

PPD - Predefined Process Diagrams show detailed logic flow of a single predefined process referenced in an IORTD or another PPD
- see figure 3.5.3

DSD - Data Structure Diagrams were not described in the article.

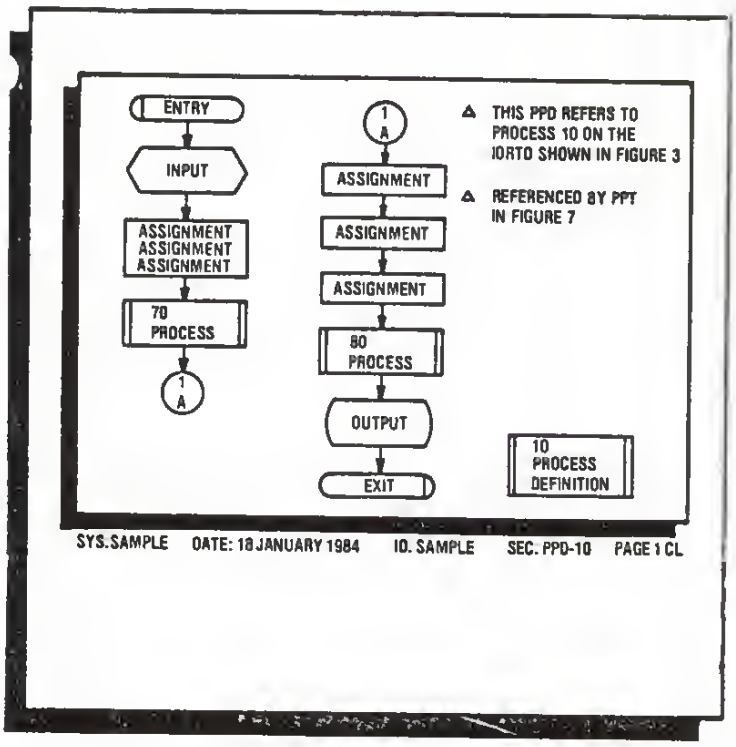


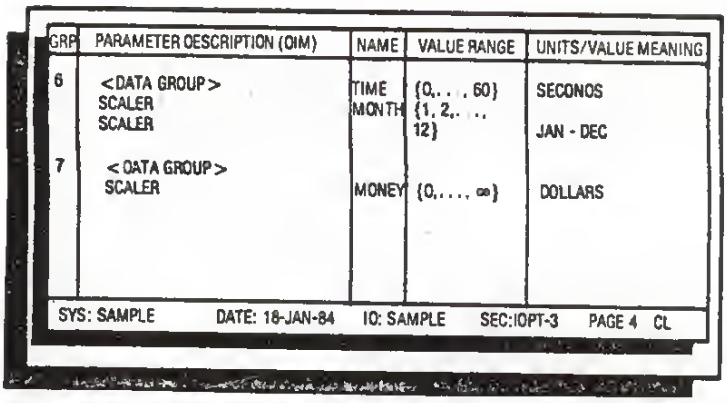
Figure 3.5.3 - A Predefined Process Diagram
(Sievert, 1985 - IEEE Computer)

TABLES

IPT-0 - Internal parameter table 0 defines the data that is global to the entire system.

IOPT - an Input/Output table defines interface variable parameters. Variables in this table are defined for both components involved in the interface.

- see figure 3.5.4



GRP	PARAMETER DESCRIPTION (DIM)	NAME	VALUE RANGE	UNITS/VALUE MEANING
6	<DATA GROUP>			
	SCALER	TIME	{0, . . , 60}	SECONDS
	SCALER	MONTH	{1, 2, . . , 12}	JAN - DEC
7	<DATA GROUP>			
	SCALER	MONEY	{0, . . . , ∞}	DOLLARS

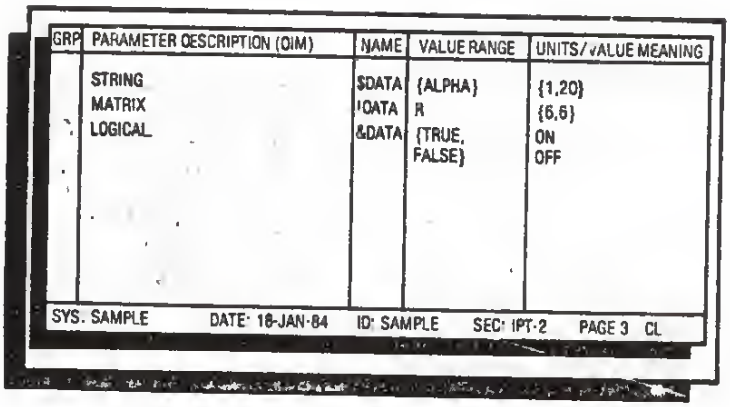
SYS: SAMPLE DATE: 18-JAN-84 ID: SAMPLE SEC:IOPT-3 PAGE 4 CL

Figure 3.5.4 - An I/O Parameter Table
(Sievert, 1985 - IEEE Computer)

IPT-n - an internal parameter table of level n (n>0) defines data that is global to component n.

IPT - an internal parameter table. Data defined for an individual PPD.

- see figure 3.5.5



GRP	PARAMETER DESCRIPTION (DIM)	NAME	VALUE RANGE	UNITS/VALUE MEANING
	STRING	$DATA	{ALPHA}	{1,20}
	MATRIX	!DATA	R	{6,6}
	LOGICAL	&DATA	{TRUE, FALSE}	ON OFF

SYS: SAMPLE DATE: 18-JAN-84 ID: SAMPLE SEC: IPT-2 PAGE 3 CL

Figure 3.5.5 - An Internal Parameter Table
(Sievert, 1985 - IEEE Computer)

PPT - Pre-defined process parameter table. "Defines

parameters that are local to one PPD." May include references to variables in other sections used by the PPD.

3.5.3 Output

The Diagnostic Analyzer emits Ada templates to be used in simulating the software system. The Simulation Compiler creates Ada source code that links the templates into an Ada simulation package. This package is then executed on data and constraints interactively supplied during the process of the Simulation Compiler. The desire is to allow the designer to test the performance of different algorithms and system configurations.

3.5.4 Observations

The graphic and tabular language of IORL is a step forward from hand-drawn requirements diagrams and pages of data dictionaries. As a recently available tool (commercially available in 1979), TAGS is displaying the increasing usefulness of graphic interfaces to software engineering tools. The designer is able to build a graphic model of the software system at a graphics workstation, have the information saved on disk, and modify or add to it as necessary during the development of the system. The traditional data dictionary is represented by data tables, with data entered into tabular form from the terminal.

Also, the methodology greatly aids the early detection of errors and design performance weaknesses. The Diagnostic Analyzer and Simulation Compiler are able to detect static and dynamic errors early in the design. Additionally, the ability of TAGS to create executable prototypes is significant. This allows fine-tuning to be accomplished early in the development stage, helping to reduce modification costs later.

No indication is given of any natural link from the various diagrams to their associated data tables. It would be useful to be able to easily move from one representation to the other. When developing a large system made of hundreds of components, it would be helpful to be able to move through the various levels of the Schematic Block Diagrams and, when information is needed about a certain component, to simply bring it up on the screen right then. Once the designer learns what is needed, moving back to the SBD screen should be equally simple.

3.6 Summary

From Gambit we see an example of "direct manipulation" and development from graphic representations to detailing text specifications. Gambit also moves easily from graphic specification, to data entry and review, and back to graphics. In HOS's USE.IT tools we see the use of different

modes such as the display-tree mode, the graphic edit mode, and the documentation mode. Again, easy movement between modes is provided. SREM, HOS, and PSL/PSA show the ability to analyze specifications for inconsistencies, and PSL/PSA gives an example of pre-graphic-workstation hypergraphic output. SREM adds some handling of non-functional requirements, though not graphically. TAGS adds the dimension of generating Ada language templates. Each of these features has a part in a good automated graphic specification tool.

CHAPTER 4.

GRAPHIC TOOLS FOR GENERATING SOFTWARE SPECIFICATIONS

This chapter discusses general desirable characteristics of tools for software specifications. It focuses on the formal models, user interfaces, and resulting output of such tools. Because the desire has been to develop specifications for Ada language software systems, the discussion of the user interface covers general graphic oriented issues and then Ada language oriented issues. Types of output from such a tool are examined for their use either by themselves or as input to other tools.

This chapter presents concepts developed from integration of information from the literature cited in the previous three chapters and insights acquired through development of the prototype detailed in chapter five.

4.1 A Formal Model

Choosing a specific formal model for specifying systems is mostly a matter of personal taste. Each model deals with the same basic information. Functional descriptions take the form of mathematical formulas, state transitions, text descriptions, processes, or others. Graphically these may be boxes, rectangles, circles, tree-nodes, ovals, or some other geometric shape. Data takes the form of entities,

BNF-like descriptions, text descriptions, high-level-language user-defined types, or data dictionary entries. Graphically data may be bubbles, rectangles, labeled arcs, or simply text names beside processes. Control information takes the form of text cross-referencing, "uses" clauses, or procedural calling hierarchies. Graphically control is normally shown through some connections between components.

Two graphic representation methods are well known for use with Ada language software systems (Booch, 1983; Buhr, 1984). Though they take a little work to understand, they are quite rich in information. Both methods combine control flow and data flow, as well as more detailed interface information. However, they go much closer to design specification as opposed to requirements specification than is desired for this paper. However, a good example of a graphic software development tool based on the design of Buhr (1984) can be found in Buhr (1985).

An access-graph model represents very well the concept of building software systems from existing components. Specifically with the Ada language in mind, although other languages offer similar concepts, building systems from a program library of general purpose generic and non-generic packages is one way of rapidly developing a software system. The access-graph model pictures such development in a conceptually clean way.

Top-down, step-wise refinement is a method found to some extent in almost any problem solving technique. The functional decomposition of HOS (Hamilton, 1976), the refinement of Schematic Block Diagrams in TAGS (Sievert, 1985), and the hierarchical decomposition of SADT (Ross, 1985) all show use of some version of step-wise refinement. Therefore, such a development methodology seems to be popular and useful.

Though top-down development and composition appear to be contradictory development methods, this is not necessarily the case. As a designer refines a system he/she may discover that the next step in the refinement requires previously designed components. Simply naming the library package and giving a component access to it completes that refinement step.

4.2 User Interface

Two main issues face the user interface described here. These are the graphic issues such as methods of drawing, moving, deleting, viewing details, or otherwise manipulating the graphic representation, and the issues dealing with the specification language of choice, the Ada language specification.

4.2.1 Graphic Issues

Interactive, graphic development of a system specification is the theme of this paper. The main areas of interest are how to draw objects, how to connect objects, how to move objects, how to delete objects, and how to enter, view, and edit the specification details.

Interactive drawing of diagrams can be accomplished using many methods. One method requires the user to place a marker (cursor) at the location of the desired object, and then enter a one-key or one-word command for drawing the object. This works fairly well when there are a limited number of commands to remember. Two methods make use of a menu of graphic objects. One has the user move a marker to the desired object on the menu. Pressing a key highlights or otherwise indicates which object has been selected. The user then moves the marker to a chosen position on the screen and again presses a key. The selected object is drawn at the marker location. The second method is similar, except that when an object is selected from the menu, a copy of it replaces the marker and moves just like the marker would until a "release" command is given in the form of a command or a mouse button. (This is known as "dragging" the object.) The latter of these methods would appear to provide the better visual feeling desired of a graphic interface. A third method requires the user to actually

draw an object physically using a mouse, "pen and pad", or touch sensitive screen. Though this is great for drawing pictures, it would detract from the formality of predesigned objects with predefined meanings. Probably the least desirable method is having a command line which provides the name of the object and the x,y coordinates of the desired location for the object.

For the application involved, each symbol has a specific meaning. Therefore, selecting a symbol from a menu, dragging it to the desired location, and releasing it appears to be the most useful method. This does not require knowledge of any commands, but only the buttons on the mouse or the keys needed to move, pick up, and set down.

Connecting the objects on the screen also offers a variety of options. In the Gambit tool (Braegger, 1985), a dialogue is used to name the objects involved in a relationship. Once the information has been provided, the tool decides what kind of connection should be used, where to draw it, and then draws it. The command-line option is available for any graphic action. In this case the user could enter something like "connect from_object_name to to_object_name". Another method is to enter a command indicating the first, intermediate, and end points for an arrow. The line could be drawn all at once after the end point is indicated, or section by section as each intermediate point is indicated.

Drawing arrows could reasonably be done using a mouse or a drawing pad, which would allow for greater flexibility in object placement and provide neater diagrams.

Side issues on line-drawing include using or not using "rubber-band" lines, lines which follow the cursor wherever it's moved, and allowing different line styles to provide different meanings. Rubber-band lines are user-friendly in that as the line is being drawn, the user doesn't have to guess if it is going to inappropriately cross other objects. Different line styles are useful for providing greater semantic meaning to the graph.

Once several objects have been placed on the screen, the need for rearrangement may become evident. Simply erasing and redrawing objects is possible, but brings up problems of whether or not all the text specification details would have to be re-entered. A more elegant method is to select an object and "drag" it to its new position. Similar but not quite as visual is to select an object, move a cursor to the desired position, and command the move. The object is then erased from its current position and redrawn at the cursor location. Other types of moves are possible. If the chosen model is tree-like, the user might desire to move an entire sub-tree, connecting it to a different leaf or even inserting it between two nodes. All of these moves may have great effects on the underlying data structure which must be taken into account.

Deleting objects is relatively simple, but again the effects on the specification must be consistent with the action. Issues such as the status of a sub-tree of a deleted node arise with such actions. It would be useful to be able to get to such a disconnected subtree through some means other than the non-existent node. In this area especially, but in other areas also, the ability to undo an action becomes very important.

Viewing comes in two different areas. These are viewing the graphic representation and viewing the specification details. For viewing the graphic representation, one method would break the graph into several diagrams hierarchically such as in SADT (Ross, 1985). The user could move from diagram to diagram through the logical contacts between the diagrams. A more powerful method would define the specification as a single graph through which the user could scan. The tool would provide a moving window on the entire graph to show a selected part of the graph. Added to this would be the ability to change the scale of the information, so that the entire graph could be viewed on the screen. Of course, the components of a large graph would be very small when viewed all at once.

Finally, the need to enter, view, and edit the detailed

information required such as inputs, outputs, functional specifications, non-functional specifications, and interface information must be satisfied. It is possible to allow all of this in one setting, much like the now-familiar full screen editors. However, this method could allow making changes that could disrupt the graph-text consistency. Another solution is to have separate modes for each action. When an object is first drawn, an initial window would appear allowing the interactive entry of the data needed by the chosen specification model. At any later point in time, the data could be viewed or edited. Data could be displayed in the viewing mode either in "raw" form such as VAR = var_name, or in some other syntax such as a high-level-language template. Editing of data could be done in the same way, but would best be done in raw form so the user knows precisely what variable is being changed. An important concept is to ensure either that the user cannot textually modify data that affects the graph, or that any modifications to such data automatically modifies the graph also.

4.2.2 Ada Language Issues

At least three issues confront the individual or tool that would specify system requirements using the Ada language. First is whether or not the use of only the Ada language

specification is sufficient to describe a software system. Second is the ability to handle all the possible variations of a specification declaration, which is not a small task. Third is the development of non-procedural packages - i.e. packages of user-defined data types.

The unfortunate answer to the first issue is no, an Ada language specification is not sufficient in itself to describe a system. This is born out by the work of Wolf (1985) and Rudmik (1982). The Ada language specification describes the interface of the specified component, but neither the functional or the non-functional requirements for the implementation are described in Ada language syntax. This makes it necessary to either revert to a text description in comment form, or add to the language as in Wolf (1985). An ideal response would be to add a menu-selectable choice of specification languages to be used in a design session for functional and non-functional requirements statements. The appropriate sequence of specification data collection could then take place in the same window as the Ada language data collection. The non-Ada information would be maintained in the same manner as Ada information. This would add the flexibility of using the data collected for further analysis by tools which use the specified data.

The complexity of the Ada language adds another dimension of

difficult issues. Nesting of packages, procedures, tasks, and functions to theoretically unlimited depth creates many headaches for designing a graphic representation and handling the data collection for every possible option. The most realistic, though somehow displeasing, response is to make certain "stylistic" limitations on the design of Ada language systems. The most effective of these limitations is eliminating the nesting of packages (Clark, 1980). Personal preferences of applying or not applying "use" clauses is another, less complex issue. Should a tool assume that all accessed packages be included in a use-clause, that none should, or that some combination should be allowed? A useful solution is to define for each user a "user profile", which would allow personal preferences to be maintained. When activating the tool, it would automatically set certain decision parameters based on the user's profile, or use defaults for those parameters unspecified. Interactively setting or resetting of these parameters should be available during the session as the situation requires.

An important use of Ada packages is development of a common pool of user-defined types. A specification tool needs to be able to develop such packages. Once developed, the user ought to be able to bring up a window concurrently with the specification entry window so that he or she can be reminded of what types have already been defined.

4.3 Output

The purpose of the design is to provide a graphic tool whereby a user can graphically decompose a problem, specifying details about the procedures, inputs, outputs, and accesses in such a way as to allow generation of Ada language specifications. As has been pointed out, this is insufficient to completely describe the intent of or requirements for the underlying implementations. Even if the designer makes excellent use of data naming, package naming, and procedure naming, added comments are required to describe the function of the designed system.

Many output possibilities exist including code generation, output produced for use as input to other specification analysis tools, or creation of program templates for various high-level languages. This depends on how much information is acquired and in what format during the actual specification process.

As current program-generation technology increases, the output possibilities of automated tools have already been improving. The HOS methodology, along with its support tool family called USE.IT, already does some automatic code generation directly from its specifications (Hamilton,

1983). Many formal specification languages and accompanying graphic documentations are created, as in the TAGS methodology (Sievert, 1985). Using the proposed graphic interface as a front-end to these or other methodologies would add the capability of beginning with a graphic specification instead of waiting for one to be generated from the text specification.

Not only could an implementation produce output suitable for other specification tools, it could be used to produce various program templates. The original implementation which instigated this research, although much less powerful than that suggested here, created C-Pascal templates from access-graphs of small programming assignments for an Operating Systems graduate-level class. The current implementation creates Ada language specifications from an access-graph model of specifications. This could also be used to gather more information or re-arrange the available information to produce Ada language package body templates.

4.4 Summary

The ideal tool would be something like the description that follows. It should have interactive editing of a graphic representation that closely corresponds to the application being specified (or the language to be used for coding). For example, an access-graph might be used to represent an

Ada language specification. A menu of available symbols pertinent to the model should be available from which the user would select and drag symbols to their desired location. At that point a window should appear, allowing a query-response dialogue which provides gathering of the detailed data required by the model in use. (The system should handle incompleteness in a satisfactory way when all details are not yet available.) The user should be able to navigate through the graphic model in a way that is logical to the model being used (down, up, and across trees; from diagram to diagram in refinement models, etc.). The user should be able to retrieve to a window the detailed information related to the symbol that the marker is at, edit or view the information as desired, and return to the graph at the point it was left. All modifications that take place in either graphic editing or text editing should cause the corresponding modifications in the other. Finally, the output created by the tool should be oriented toward the application being developed. A display file should be created which would allow retrieval and further editing at a later time. If other tools exist in the current environment, this tool should create output of use to those other tools.

CHAPTER 5. GTGALS - A PROTOTYPE

This chapter describes the prototype implementation of a Graphic Tool for Generating Ada Language Specifications. The prototype is written in Turbo Pascal using an abbreviated version (see appendix B) of the Turbo Graphix Toolbox. The prototype was developed and runs on a Zenith Z-150 micro- computer. It has 4000 lines of source code (approximately 1680 lines are Turbo Graphix Toolbox code), compiling to 52K bytes of object code. At the current limit of 20 graphic objects and 100 access arrows, it requires 57K bytes of data space. Some dynamic allocation of memory heap space is done. Therefore a minimum of 320K bytes of internal memory is suggested to avoid some difficulties experienced with Turbo Pascal's heap space management. The output of the program, if the user decides to request it, is a filename.gph file and a filename.ada file. The .gph file is the display file (see appendix C), and the .ada file is the Ada Language specification of the developed access-graph (see figure 5.2.7 at the end of this chapter). (The filename is supplied interactively at the end of the GTGALS session.)

After briefly reviewing the choice of the access-graph model for the formal model, the what's and how's of the actual program are detailed. The program allows drawing and

deleting of objects and directed arcs and naming and specifying procedures and their inputs and outputs for each object. It automatically modifies the underlying data structure corresponding to graphic actions. The program will create Ada language specifications from the graphic specification, and allows saving a display file on disk which can be retrieved and further edited.

5.1 - Formal model

The access-graph model was used to better conceptualize the building of software systems from existing programs such as in an Ada program library (DOD, 1983). It has been modified for graphic reasons; fitting a large system on one diagram would cause reading problems. Top-down, step-wise refinement is the recommended method of development using this implementation. However, a bottom-up, compositional method could be used.

5.2 - User Interface

The key concepts of GTGALS lie in its graphic interface. Its purpose is to allow the designer(s) to graphically lay out the software system, interactively providing as much or as little detail as available initially.

5.2.1 - Graphic Design and Specification

The user first moves a cursor to the location on the screen for drawing an object. Objects include packages, subprograms, generic packages, and generic subprograms. Pressing "p", "s", "gp", or "gs" will, respectively, draw the symbols for these objects. At any time that another window is not on the screen, pressing "h" will bring up a help window. This window contains the commands with a brief description of what they do. (see figure 5.2.1)

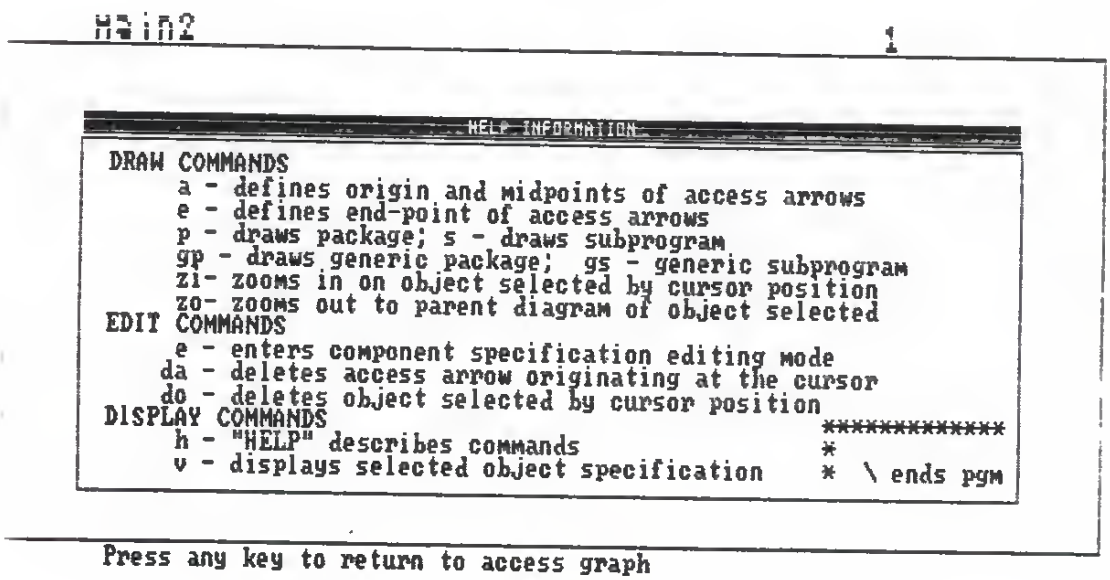


Figure 5.2.1 - The GTALS Help Window

Interactive prompt-response sequences then allow the designer to indicate for each component its name, procedure names, and the inputs and outputs for each procedure. The user can provide comments for the entire component as well as for each interface procedure or function. As little or

as much of this information as desired can be provided. After specifying several objects, access of object "B" by object "A" is accomplished by drawing an arrow from object A to object B. This is done by placing the cursor at the edge of object "A" and pressing "a" (for arrow). The cursor is then moved to the edge of object "B" and "e" (for end arrow) is pressed. If necessary, intermediate points can be established to draw around objects by pressing "a" at each intermediate point. Pressing "e" draws the last section of the arrow, plus the arrowhead. This automatically includes object B as an access parameter for object A. In fact, access parameters can only be identified in this manner. Therefore the data accurately reflects the graph, and the graph accurately pictures the data. (see figure 5.2.2)

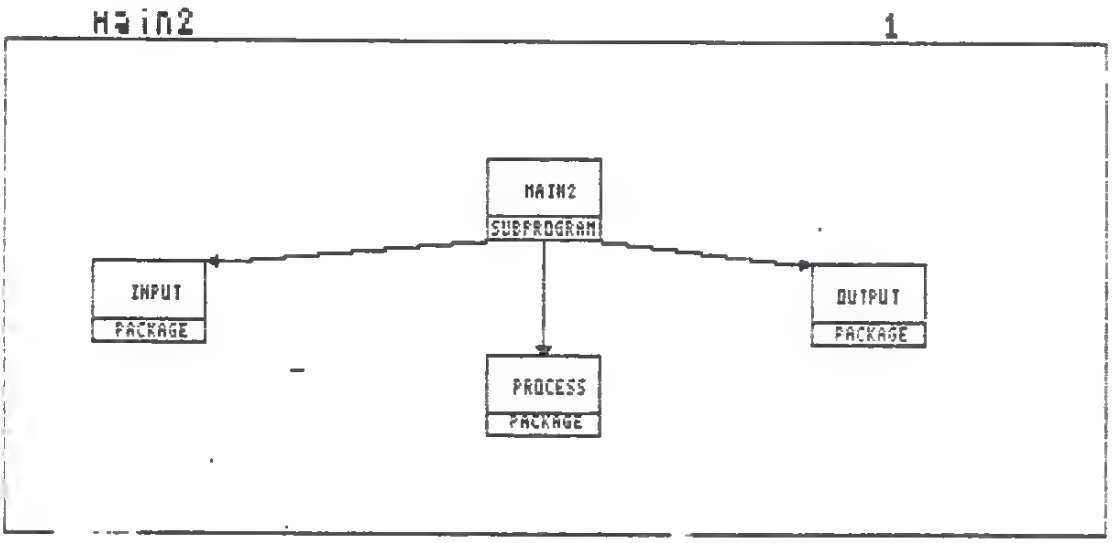


Figure 5.2.2 - GTGALS screen

5.2.2 - Specification Viewing

Another feature is that there is direct access to a component's specification from the graph. By moving the cursor to a component and pressing "v" (for view), the system creates a window and displays the data for that component. The data is displayed in Ada language specification syntax and is shown thirteen lines at a time. Only forward movement through a specification is currently supported. The designer can view the data and then return to the access-graph. (see figure 5.2.3)

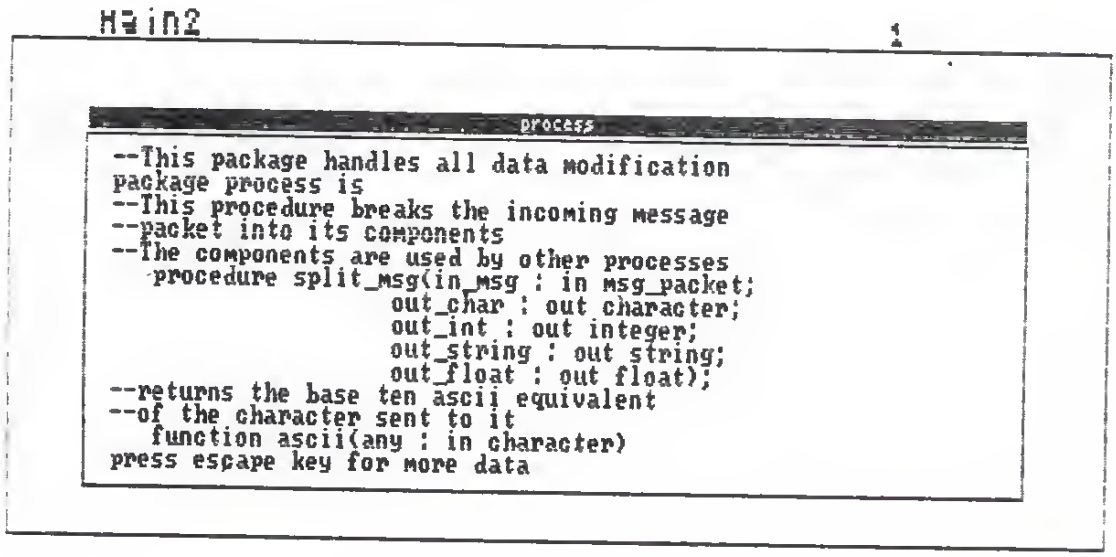


Figure 5.2.3 - GTGALS View Mode

5.2.3 - Graphic Editing

Deleting graphic objects or arrows results in an appropriately modified graph and data structure. For

example, deleting a package will also delete all arrows going to that package. Consequently, any component that has the deleted package in its access parameters will have the package's name removed. Deleting just an arrow ("da") removes access in the "from" object for the "to" object, but both objects remain in the structure and on the graph. The command "do" when the cursor is within a selected object will result in a verification request for deleting the object. A reply of "y" will result in the object being erased from the screen and its entire data structure re-initialized. This means that any graphs decomposed from that object will no longer be accessible.

5.2.4 - Specification Editing

Editing of component data is done on a simple basis. Each item of data for an object is shown one at a time. The user can either modify the item by typing "m" and then the new item, move to the next item by typing "n", or exit the editor by typing "e". As well as changing a comment, additional comments may be entered at the end of the current comment block. By typing "a" after the ? prompt at the end of a comment, the editor will allow the user to enter more comments. (see figure 5.2.4)

5.2.5 - Development Method

Based on a decompositional approach to design, GTGALS allows

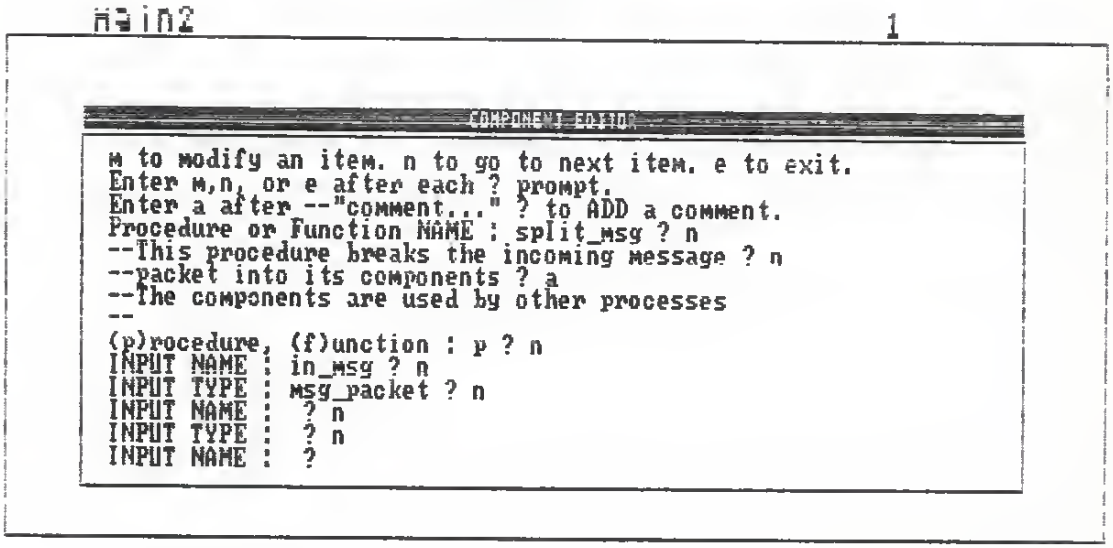


Figure 5.2.4 - GTGALS Specification Edit mode

multiple graphs. A typical example would be to divide a system into INPUT, PROCESS, and OUTPUT components, all under control of a main program. The next step would be to decompose the INPUT component. In GTGALS, this is done by "zooming in" on the INPUT component by moving the cursor to the component and pressing "zi". This moves to a new diagram. If INPUT has already been decomposed, the diagram will reflect the current design. If not, the designer chooses where to place the box representing INPUT. (see figure 5.2.5)

The designer can then draw, specify (see figure 5.2.6), and connect components as required. "Zooming out" by pressing "zo" from a component will bring on screen the diagram on which the component and its parent (the component from which

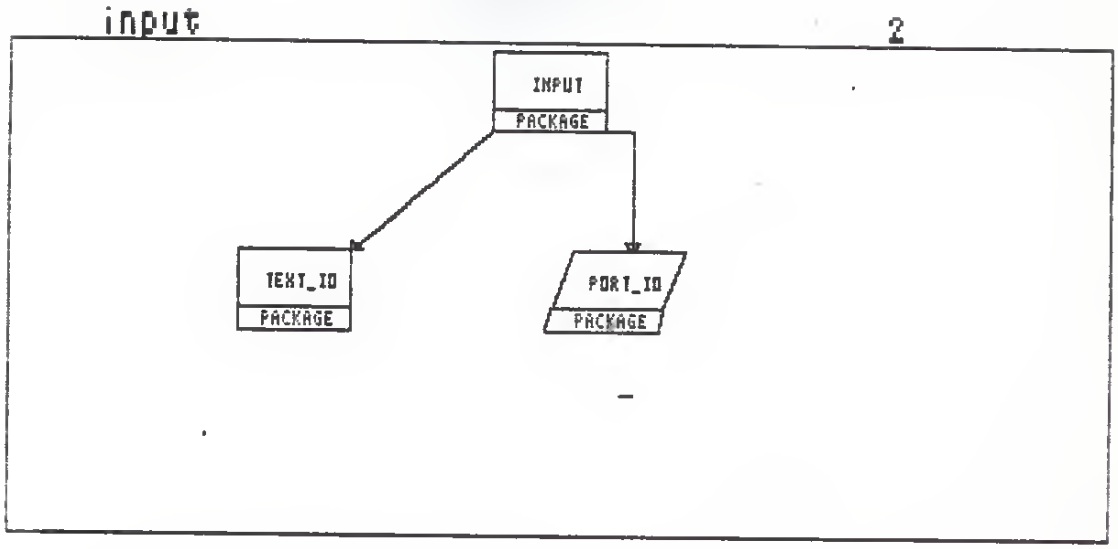


Figure 5.2.5 - Decomposition of INPUT from MAIN2

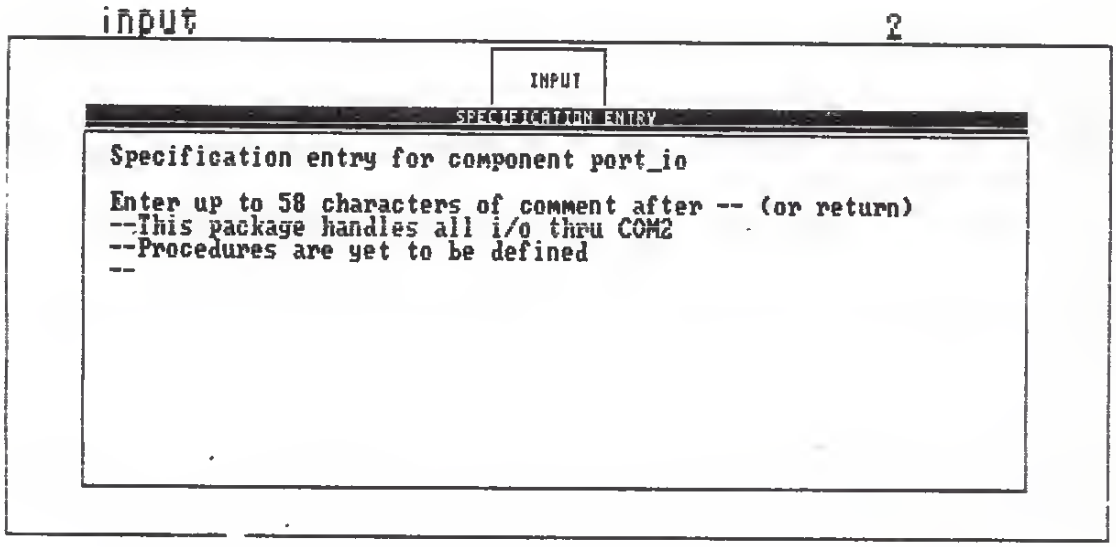


Figure 5.2.6 - Specification Entry for an object

it was decomposed) are both drawn.

Though the zoom in and zoom out commands are conceptually tied to functional decomposition, a bottom-up composition

could be accomplished by conceptually switching their roles. For example, draw several objects on the bottom of the screen and then one or more objects above them to represent the composition of the lower components. Next, "zoom in" on an upper level component. Place that component on the bottom of the new diagram. Draw several "sibling" components, and repeat the process of compose and zoom in.

5.3 Output

The implementation creates the Ada language specification part of a component (see figure 5.2.7 on page 71). The file would reside on disk as a filename.ada file, where filename is supplied by the user during the GTGALS session. For each component in the graph an Ada language specification part will be created based on the data entered during that design session. This will include the with-clause and the procedure specifications. Currently the tool allows package and generic package specification including their procedure interfaces, and subprogram and generic subprogram specification. Nesting of packages is not handled, and tasks are not handled. Individual tasks could be easily added to the implementation, but packages of tasks would be somewhat more difficult.

```
--This is the controller
with process,
     input,
     output;
procedure main2(in_msg : in msg_packet;
                out_msg : out msg_packet);


--This package handles all data modification
package process is
--This procedure breaks the incoming message
--packet into its components
--The components are used by other processes
   procedure split_msg(in_msg : in msg_packet;
                       out_char : out character;
                       out_int : out integer;
                       out_string : out string;
                       out_float : out float);
--returns the base ten ascii equivalent
--of the character sent to it
   function ascii(any : in character)
                        return integer;
end process;

--This packages interfaces to the "outside world"
with text_io;
package input is
   package DUMMY is new port_io;
--for reading entire message packets
   procedure read_msg(got_msg : out msg_packet);
end input;

--This handles ouput interface to environment
package output is
--Writes the message to the standard output file
   procedure write_msg(in_msg : in msg_packet);
end output;

--This is a predefined library program
package text_io is
end text_io;

--This package handles all i/o thru COM2
--Procedures are yet to be defined
generic
package port_io is
end port_io;
```

Figure 5.2.7 - Ada Language specification of MAIN2

CHAPTER 6. CONCLUSIONS

6.1 Usefulness

What has been learned from this research and design effort falls into the categories of the implementation, Ada language specifications, and tool output.

6.1.1 Implementation

Implementing a major project in Turbo Pascal, while it offers many advantages, suffers from two serious disadvantages. The advantages come from the language Pascal and the availability of the Turbo Graphix Toolbox. The structured nature of Pascal allowed procedural additions and incremental development of the project. The Turbo Graphix Toolbox eliminated the need to develop graphics and windowing procedures. The unfortunate disadvantages were the limitations on code space and data space. Though there are tools to circumvent these limitations, they were not accessible at the time of project development. The results of these limitations contributed to various decisions that detract from the usefulness of the final prototype. These decisions were the elimination of package nesting, the absence of handling packages of tasks, not handling generic type specification, the rather crude specification editor, and the number of objects which can be specified.

6.1.2 Ada language specifications

As research progressed, it became clear that Ada language specifications were never intended to be requirements specifications. Rather they are descriptions of the interfaces to their respective package bodies. (Their acceptability even for this is disputed by Wolf (1985)). Therefore, to adequately specify a software system, either additions to the language or use of some other specification language is necessary. This does not detract from the usefulness of this study. An access graph is still a good model for graphically describing Ada language software systems, and a graphic tool is by far the most enjoyable method for developing such a specification. However, to adequately and accurately specify the requirements for a software system in such a way as to promote correct results requires more than just the Ada language specification. Section 6.3 continues this issue.

6.1.3 Automatic Code Generation

The question is likely to arise, "Why bother with just specifying Ada language units instead of proceeding to automatic code generation?". With most code generation techniques now available, decomposition is required to a very detailed level and this level must be functionally primitive. It is the purpose of this paper to accomplish

the first level of this decomposition - specifying the separately compilable Ada language units. The main issue of this study has been the user interface to tools. What the tools can do once they have the information is "beyond the scope" of this paper. However, code generation systems probably require much more substantial computing power than is currently available on a 320K personal computer with one disk drive, which is the system used for development and running of the prototype.

6.2 Appropriateness of design

Does the formal model, user interface, and output of the design adequately display the capabilities of such a graphic tool as described in Chapter Four?

6.2.1 Formal Model

The access-graph model appears to accurately describe the interface specification for an Ada language system. Since the Ada language rules permit access to the whole component which is accessed and not just particular entry points of that package (DOD, 1983), the model clearly indicates this. An access graph can easily support all of the interface syntax inherent in Ada language specifications, even if the implementation does not. The weakness would come in graphically describing component bodies, since unfortunately

they can gain access to packages not already accessed in the specification.

6.2.2 User Interface

Much more could have been done in the implementation in regards to the interface design, given time and a tool to circumvent the limitations described in 6.1.1. However, even at its current level the prototype demonstrates the usefulness and desirability of such a tool. The fact that new tools are using such graphics, and older tools are adding them (e.g. HOS and USE.IT), gives support to the popularity of graphic interfaces.

6.2.3 Output

As already discussed, Ada language specifications are inadequate for accurately describing a software system. However, the output of the prototype does provide a collection of interface descriptions which would be helpful in designing the implementation of that system. If an implementor could access the interface specification through a workstation while developing the implementation, he or she could determine the necessary parameters for interfacing with the selected component. Additionally, the output from this tool could be run through an Ada language compiler to determine at least some amount of interface consistency.

6.3 Recommended extensions and modifications

At least two major areas require further development. Little mention has been made of the analyzability of the data produced by the design tool. This area needs to be examined. Though mentioned earlier, the idea of using this tool as a front end to other tools should be further studied.

6.3.1 Specification Analysis

The amount of analysis that can be done on a specification is a function of the amount and formality of the data produced by the tool (see 2.3). Since this design creates Ada language syntax specifications, the amount of analyzability is determined by the number of analysis tools present in the environment which use those specifications as input. At the very least, this would be the compiler. Unfortunately, the compiler will basically only tell you if the packages you have attempted to access in a with clause actually exist. Therefore, repeatedly recommended additions to the specifications in either the form of comments or additional language constructs and preprocessors are necessary. (See Wolf (1985) for one such language extension).

6.3.2 Front-end to Other Tools

Because of the inadequacy of the Ada language specification as a requirements specification on its own, the use of this design as a front end to other specification tools might be possible. Since many methodologies are now moving toward the addition of graphic interfaces to their tools, this is an unlikely proposition. However, it would be nice to see more of the tools being developed offer some version or implementation with a bent toward the Ada language, since like it or not Ada is going to be used in many areas.

6.4 The Needs

In attempting to develop this graphic interface, several needs have become evident. A need for cheaper, more accessible graphics workstations; more tools or additions to high-level-languages to take advantage of such workstations; and more emphasis in software design on graphic interfaces to development tools. Whether or not this need is a result of the environment under which this paper and project was developed is unknown.

The ultimate purpose of this paper is to encourage an increase in the number and varieties of graphic interfaces to software engineering tools.

REFERENCES

Alford, M. (1985). "SREM at the Age of Eight; The Distributed Computing Design System," IEEE Computer, April, pp. 36-46.

Balzer, R. and Goldman, N. (1979). "Principles of Good Software Specification and Their Implications for Specification Language," Proc. Specifications of Reliable Software Conf., September, pp. 58-67.

Booch, G. (1983). Software Engineering With Ada. Menlo Park, CA: Benjamin/Cummings Publishing Co.

Braegger, R. P., Dudler, A.M., Rebsamen, J., and Zehnder, C.A. (1985). "Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints, and Transactions," IEEE Transactions on Software Engineering, July, pp. 574-582.

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, August, pp. 27-35.

Buhr, R.J.A., Karam, G.M., and Woodside, C.M. (1985). "An Overview and Example of Application of CAEDE: A New, Experimental Design Environment for Ada," ADA Letters, September, pp. 173-184.

Buhr, R.J.A. (1984). System Design with Ada, Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Clark, L.A., Wileden, J.C., and Wolf, A.L. (1980). "Nesting in Ada Programs is for the Birds," Proc. ACM-Sigplan Symp. Ada Programming Language, in Sigplan Notices, November.

DeRemer, F. and Kron, H.K. (1976). "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, June, pp. 80-86.

DOD (1983). Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, Washington, D.C.: US Dept. of Defense, January.

Gilbert, P. (1983). Software Design and Development, Chicago, IL: Science Research Associates.

Grafton, R.B. and Ichikawa, T. (1985). "Visual Programming," IEEE Computer, August, pp. 6-9.

Hamilton, M., and Zeldin, S. (1983). "The Functional Life

Cycle Model and Its Automation: USE.IT," The Journal of Systems and Software, March, pp. 25-62.

Hamilton, M., and Zeldin, S. (1976). "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, March, pp. 9-31.

Hansen, P.B. (1977). The Architecture of Concurrent Programs, Englewood Cliffs, N.J.: Prentice-Hall.

Jacob, R.J.K. (1985). "A State Transition Diagram Language for Visual Programming," IEEE Computer, August, pp. 51-59.

Myers, W. (1978). "The Need for Software Engineering," IEEE Computer, February, pp. 12-25.

Peterson, J.L. (1981). Petri Net Theory and the Modeling of Systems, Englewood Cliffs, N.J.: Prentice- Hall.

Pressman, R.S. (1982). Software Engineering: A Practitioner's Approach, New York, NY: McGraw-Hill, Inc.

Roman, G. (1985). "A Taxonomy of Current Issues in Requirements Engineering," IEEE Computer, April, pp. 14-21.

Ross, D.T. (1985). "Applications and Extensions of SADT," IEEE Computer, April, pp. 25-34.

Rudmik, A. and Moore, B.G. (1982). "An Efficient Separate Compilation Strategy for Very Large Programs," Proc. Sigplan 82 Symp. Compiler Construction, in Sigplan Notices, June, pp. 301-307.

Schneiderman, B. (1983). "Direct Manipulation: A Step Beyond Programming Languages," IEEE Computer, July, pp. 57-69.

Sievert, G.E., and Mizell, T.A. (1985). "Specification-Based Software Engineering with TAGS," IEEE Computer, April, pp. 56-65.

Teichroew, D., and Hershey III, E.A. (1977). "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, January, pp. 41-48.

Ullman, J.D. (1982). Principles of Database Systems, Rockville, MD.: Computer Science Press.

Wolf, A., Clarke, L., and Wileden, J. (1985). "Ada-Based Support for Programming-in-the-Large," IEEE Software, March, pp. 58-71.

APPENDIX A - GTGALS Procedure Descriptions

Procedure Descriptions for GTGALS - A Graphic Tool for Generating Ada Language Specifications

These are all the procedures within the Graphic Tool for Generating Ada Language Specifications (GTGALS) system. Due to Turbo Pascal editor limitations, these are broken up into three files which, along with the type definition file, are needed to run GTGALS.

Brief comments follow each procedure to further describe its purpose.

File GTGALS1.PAS

```
procedure Adjust_name(var short_name : short_obj_name;  name
: object_name);
```

This procedure adjusts an incoming object name (of up to 20 characters) to a short name (up to 8 characters) for display withing the object symbol.

```
procedure Move_cursor_out;
```

This procedure moves the cursor-window outside of the main screen and turns it off so that when a save screen is done the cursor is not permanently displayed on one position on the screen.

```
procedure Move_cursor_in;
```

This procedure moves the cursor-window back to its previous position and turns it back on. It is used after Move_cursor_out and a save screen.

File GTGALS2.PAS

```
procedure Init_arrow(i : integer);
```

This procedure initializes one arrow, setting all the values of the indexed arrow to a known state. It is used on program start-up and whenever an arrow is erased from the graph.

```
procedure Init_object(i:integer);
```

This procedure initializes an object as above. (see Init_arrow)

```
procedure Init_structure;
```

This procedure is used to initialize all data structures at the start of the program.

```
procedure Left_justify(var name : object_name);
```

This procedure corrects for occasional right-justification of data being read in from a display file.

```
procedure Move_cursor;
```

This procedure reads the arrow keys corresponding to cursor movement on the main screen.

```
procedure New_screen(name : object_name; screen_no : integer);
```

This procedure sets up a new screen for further drawing, labeling the screen with the diagram number and the name of the object from which the screen was drawn. (If startup from a file, name is the file name, if zoom-in or zoom-out, name is the object name on which the command was given)

```
procedure Draw_arrow(x1,y1,x2,y2:real);
```

These procedures handle drawing of the last section of an access arrow and the appropriate arrow-point.

```
  procedure DrawArrow45(x1,y1,x2,y2:real);

  procedure DrawArrowHor(x1,y1,x2,y2 : real);

  procedure DrawArrowVer(x1,y1,x2,y2 : real);
```

```
procedure Draw_name(x1,y1:real; name : object_name);
```

This procedure draws the object name in the object located at x1, y1.

```
procedure Draw_object(which : char; x, y : real);
```

These procedures draw the object symbols based on an approximate center of x,y.

```
procedure Draw_std_object(x,y : real);

procedure Draw_generic(x, y : real);
```

```
procedure Draw_diagram(diag_index : integer; name : object_name);
```

This procedure selects the objects and arrows to be drawn on the diagram requested by diag_index, and uses the Draw procedures to draw them.

```
procedure Help;
```

Displays the system commands in a window. This window is accessible only from the main screen, not from within other windows.

```
procedure Remove_access(from_ind, to_ind : integer);
```

This procedure is used to remove access of the "to object" from the "from object" when either the access arrow or the accessed object has been deleted.

```
procedure Select_arrow(findx,findy : real; var found : boolean;
                       var index : integer);
```

This procedure determines which, if any, arrow begins at or near the given findx, findy coordinates.

```
procedure Select(findx, findy : real; var found : boolean;
                 var out_object : char; var index : integer);
```

This procedure determines which, if any, object surrounds the given findx, findy coordinates.

```
procedure Erase_arrow(object : char; index : integer);
```

This procedure erases the arrow indicated by index.

---
procedure Add_access(from_obj, to_obj : char; from_ind, to_ind : integer);

This procedure is used to add access when an access arrow has been drawn.

---
procedure Read_arrow;

This procedure allows the drawing of arrows and puts the data into the arrow array.

---
procedure Delete;

This procedure begins the deletion of either arrows or objects.

---
procedure Read_object(obj_type : char);

These procedures read the initial information when an object is drawn.

```
procedure get_comments(var in_ptr : comment_ptr);

procedure spec_entry;
```

---
procedure Zoom_in;

This procedure creates or accesses the screen on which the selected object is decomposed.

---
procedure Zoom_out;

This procedure moves the user back to the diagram on which the selected object is not decomposed.

File GTGALS.PAS

---
procedure Gen_Ada(index : integer; var head : spec_ptr);

These procedures build the Ada language specification from the data in the object array for the selected object.

```
procedure build_comments(in_ptr : comment_ptr);
```

```
  procedure build_parms(index, i : integer);
```

```
procedure View_text;
```

This procedure brings up the viewing window and calls Gen_Ada for the selected object.

```
procedure Edit;
```

These procedures allow for editing a selected components internal details such as name, procedures, inputs and outputs, and comments.

```
  procedure clear_window;

  procedure edit_comments(var in_ptr : comment_ptr);
```

```
procedure Read_display(filename : filenames);
```

This procedure reads a display file and puts the information into the data structure for use by GTGALS.

```
  procedure read_comments(var in_ptr : comment_ptr);
```

```
procedure Write_display;
```

This procedure writes out the data from the data structures to a uniquely formatted .gph display file.

```
  procedure write_comments(in_ptr : comment_ptr);
```

```
procedure Gen_specs;
```

This procedure uses Gen_Ada for each object in the data structure and writes it out to a .ada file.

APPENDIX B - Turbo Graphix Toolbox Modifications

The following procedures were removed from the Turbo Graphix Toolbox of Boreland International to make it possible to increase the amount of code in the Graphic Tool for Generating Ada Language Specifications (GTGALS).

The following were removed from Kernel.Sys

```
function GetErrorCode:byte;
procedure SetHeaderToBottom;
function GetWindow:integer;
function GetColor:integer;
procedure SetScreenAspect(aspect:real);
function GetScreenAspect:real;
function GetAspect:real;
procedure SetLinestyle(ls:integer);
function GetLinestyle:integer;
procedure SetVStep(vs:integer);
function GetVStep:integer;
function GetScreen:byte;
procedure DrawPoint(xr,yr:real);
function PointDrawn(xr,yr:real):boolean;
```

The following were removed from Windows.Sys

```
procedure CopyWindow(from,tu:byte;
                     x1,y1:integer);
procedure SaveWindow(n:integer;
                     FileName:wrkstring);
procedure LoadWindow(n,xpos,ypos:integer;
                     FileName:wrkstring);
procedure SaveWindowStack(FileName:wrkstring);
procedure LoadWindowStack(FileName:wrkstring);
procedure ResetWindowStack;
```

APPENDIX C - Display file for MAIN2 (see fig. 5.2.7)

This file would reside on disk as MAIN2.GPH. This is an annotated display file. The text in {} is not in the actual display file, but is used here to describe it. There would be no blank lines in the display file.

```
{The first line of an object record is its type,
s-subprogram, p-package, g-generic package,
h-generic subprogram; its array index, and its
x,y coordinates on its original diagram and its
refinement (zeros if not refined)}

s   1  500.0  320.0    0.0    0.0

{The second line is the diagram numbers on which
it is located, original then refinement}

 1  0

{The next line is the object's name}

main2

{A line preceeded by c is a comment}

c--This is the controller

{A * indicates a procedure or function}
{If followed by the word KEY, this data
is for the subprogram rather than an
internally named procedure or function}
{Otherwise, it will be followed by the
procedure or function name}

*pKEY

{? indicates input. It is immediately
followed by the input name.  The next
line will be the input type.}

?in_msg
msg_packet

{! is output. Same as input}
{If there were in out variables,
they would be indicated by a +}

!out_msg
msg_packet
```

```
{@ indicates that the number following
is an index to an accessed object}

@ 2
@ 3
@ 4

{Only different information will be
noted}

p   2  500.0  660.0    0.0    0.0
 1  0
process
c--This package handles all data modification
*psplit_msg
c--This procedure breaks the incoming message
c--packet into its components
c--The components are used by other processes
?in_msg
msg_packet
!out_char
character
!out_int
integer
!out_string
string
!out_float
float
*fascii

{if the * is a function, the next
line is the data type of the function}

integer
c--returns the base ten ascii equivalent
c--of the character sent to it
?any
character

{Notice that the following package
has been refined on diagram 2}

p   3  150.0  500.0  500.0  130.0
 1  2
input
c--This packages interfaces to the "outside world"
*pread_msg
c--for reading entire message packets
!got_msg
msg_packet
@ 5
```

```
@ 6
p   4  787.5  500.0    0.0    0.0
 1  0
output
c--This handles ouput interface to environment
*pwrite_msg
c--Writes the message to the standard output file
?in_msg
msg_packet
p   5  275.0  480.0    0.0    0.0
 2  0
text_io
c--This is a predefined library program
g   6  562.5  480.0    0.0    0.0
 2  0
port_io
c--This package handles all i/o thru COM2
c--Procedures are yet to be defined

{The first encounter of an 'a'
in column one indicates the start
of the access arrow data.}
{The first a is the originating point,
subsequent a's are intermediate points,
and the e is the end point.  This is
followed by the indices of the originating
object and then the accessed object}

a  500.0  400.0 1
e  500.0  600.0 1
   1   2
a  450.0  400.0 1
e  200.0  440.0 1
   1   3
a  550.0  400.0 1
e  737.5  440.0 1
   1   4
a  450.0  210.0 2
e  325.0  420.0 2
   3   5
a  550.0  210.0 2
a  575.0  210.0 2
e  575.0  420.0 2
   3   6
```

APPENDIX D - Source Code for A Graphic Tool for Generating Ada Language Specifications

{This program is a modification of a project done for CS736 (Computer Graphics) in the summer semester of 1985. The original program was written by :
Ernest G. Smith
Donald E. Bodle, Jr.

It's purpose was to demonstrate the use of a graphic interface to an underlying data structure. The graphic model chosen was the access graph as taught in CS720 (Operating Systems II) by Dr. Richard McBride for documenting C-Pascal programs.

The modifications that follow have been done by Donald E. Bodle, Jr. as part of his master's thesis implementation.

The main data structure has been modified, multiple levels of graphs have been added, the file format of the display file has changed slightly, and the program template is now for the Ada language rather than C-pascal. }

{These are the declarations necessary to the GTGALS program}

```
const
  max_accesses = 5;
  max_arrows = 100;      { max_objects * max_accesses }
  max_arrow_points = 5; { includes origin and end pt }
  max_inputs = 5;
  max_inouts = 5;
  max_objects = 20;
  max_outputs = 5;
  max_procedures = 5;

type
   data_name = string[10];
   filenames = string[14];
   object_name = string[20];
   output_line = string[70];
   procedure_name = string[20];
   short_obj_name = string[8];
   spec_ptr = ^spec_line_record;
   comment_ptr = ^ comment_record;

   access_record = record
     index : integer;      { array index of object accessed }
   end;
```

```
comment_record = record
  line : string[60];
  next : comment_ptr;
end;

input_record = record
  name : data_name;
  in_type : data_name;
end;

inout_record = record
  name : data_name;
  inout_type : data_name;
end;

output_record = record
  name : data_name;
  out_type : data_name;
end;

point_label = record
  object_type : char;  { for arrows, a = origin or }
  x : real;            { mid_pt, e = end. for objects }
  y : real;            { p, s, g, or h for pkg, subpgm }
end;                   { generic pkg, generic subpgm }

spec_line_record = record   { for linked list of lines }
     line : output_line;
     next : spec_ptr;
end;

arrow_record = record
  diagram : integer;
  point : array[1..max_arrow_points] of point_label;
  from_index : integer;      { originating object }
  to_index : integer;        { accessed object }
end;

procedure_record = record
  comment : comment_ptr;
  proc_type : char;     { p = procedure, f = function }
  f_returns : data_name;
  name : procedure_name;
  input : array[1..max_inputs] of input_record;
  output : array[1..max_outputs] of output_record;
  inout : array[1..max_inouts] of inout_record;
end;

object_record = record
  access : array[1..max_accesses] of access_record;
  child_diag : integer;  { if object decomposed }
```

```
      child_pt : point_label;
      comment : comment_ptr;
      diagram : integer;      { diagram where 1st drawn }
      id : integer;
      name : object_name;
      point : point_label;
      proc : array[1..max_procedures] of procedure_record;
    end;


var arrow : array[1..max_arrows] of arrow_record;
    Ch: char;                 { for keyboard input }
    filename : filenames;
    temp_file : filenames;
    i : integer;              { loops }
    in_file : text;           { read in display file }
    in_file_name : filenames;
    long_file_name : object_name;
    next_arrow,               { next empty slot ptrs for }
    next_diagram,             { arrays and diagram # }
    next_object : integer;
    object : array[1..max_objects] of object_record;
    screen_num : integer;     { screen is now this diagram }
    short_name : short_obj_name;
    tempx : integer;
    x, y : real;              { track the cursor }

{ Adjust an incoming object name from up to 20 letters
to a short name of up to 8 letters for display within
the object symbol}

procedure Adjust_name(var short_name : short_obj_name;
                      name : object_name);

begin
  short_name := name;
  i := length(name);
  case i of
    7,6 : short_name := ' ' + short_name;
    5,4 : short_name := '  ' + short_name;
    3,2 : short_name := '   ' + short_name;
  end;
  for i := 1 to 8 do short_name[i] := upcase(short_name[i]);

end; { adjust name }
{----------------------------------------}
{ Moves the cursor outside of the main screen and turns
it off so that when a save screen is done the cursor
is not permanently display at one position on the screen }

procedure Move_cursor_out;
```

```
begin
          SelectWindow(2);
          InvertWindow;
          tempx := trunc(x/12.6);
          MoveHor(-tempx,true);
          SelectWorld(1);
          SelectWindow(1);
end; { move cursor out }
{----------------------------------------}
{ Moves the cursor back to its previous position and turns
it back on.  Used after Move_cursor_out }

procedure Move_cursor_in;

begin
          CopyScreen;
          SelectWorld(2);
          SelectWindow(2);
          MoveHor(tempx,true);
          InvertWindow;
end;  { move cursor in }
{----------------------------------------}
{ File gtgals2.pas }

{ Sets one arrow to a know state.  Used at program
start-up and when an arrow is erased from the
graph }

procedure Init_arrow(i : integer);

var  index : integer;

begin
     with arrow[i] do
     begin
       diagram := 0;
       for index := 1 to max_arrow_points do
       begin
          point[index].object_type := ' ';
          point[index].x := 0;  point[index].y := 0;
       end;
       from_index := 0; to_index := 0;
     end; { with and for }
end; { Init_arrow }
{----------------------------------------}
{ Initializes an object. Used as Init_arrow is }

procedure Init_object(i:integer);

var index, k : integer;
```

```
begin
     with object[i] do
     begin
       diagram := 0;
       child_diag := 0;
       name := '';
       id := 0;
       point.object_type := ' ';
       point.x := 0; point.y := 0;
       child_pt.object_type := ' ';
       child_pt.x := 0; child_pt.y := 0;
       comment := nil;
       for index := 1 to max_procedures do
       begin
          proc[index].proc_type := ' ';
          proc[index].f_returns := '';
          proc[index].name := '';
          proc[index].comment := nil;
          for k := 1 to max_inputs do
          begin
            proc[index].input[k].name := '';
            proc[index].input[k].in_type := '';
          end;
          for k := 1 to max_outputs do
          begin
            proc[index].output[k].name := '';
            proc[index].output[k].out_type := '';
          end;
          for k := 1 to max_inouts do
          begin
            proc[index].inout[k].name := '';
            proc[index].inout[k].inout_type := '';
          end;
       end;
       for index := 1 to max_accesses do
          access[index].index := 0;
     end; { with and for }
end; { Init_class }                    object
{-----------------------------------------}
{ Uses init_arrow and init_object at program
start-up }

procedure Init_structure;

var
    i : integer;

begin
     for i := 1 to max_objects do Init_object(i);
     for i := 1 to max_arrows do Init_arrow(i);
end; { Init_structure }
```

```
{----------------------------------------}
{ Corrects for occasional right-justification
of data that has been written to a text file
using the var_name : ## format }

procedure Left_justify(var name : object_name);

var i,max : integer;

begin
  if name[1] = ' ' then
  begin
    max := length(name);
    for i := 2 to max do
      name[i-1] := name[i];
    name[max] := ' ';
  end; { if not left justified }
end; { procedure left_justify }
{----------------------------------------}
{ Reads the arrow keys corresponding to cursor
movement on the screen }

procedure Move_cursor;

begin
  case ord(Ch) of

      72 : if y >= 140 then
           begin
             MoveVer(-2,true); {up arrow?}
             y := y - 10;
             gotoxy(1,25);
           end;
      75 : if x >= 82.5 then
           begin
             MoveHor(-1,true); {left arrow?}
             x := x - 12.5;
             gotoxy(1,25);
           end;
      77 : if x <= 926.0 then
           begin
             MoveHor(1,true);  {right arrow?}
             x := x + 12.5;
             gotoxy(1,25);
           end;
      80 : if y <= 820 then
           begin
             MoveVer(2,true);  {down arrow?}
             y := y + 10;
             gotoxy(1,25);
           end;
```

```
    end; { case }
end; { move_cursor }

{-----------------------------------------}
{ Sets up a new screen for further drawing,
labeling hte screen with the diagram number and the name of
the object from which the screen was drawn.  (If startup
from a file, name is the file name, if zoom-in or zoom-out,
name is the objeot name on which the command was given) }

procedure New_screen(name : object_name;
                     screen_no : integer);

var screen_char : char;

begin
  screen_char := char(screen_no + 48);
  ClearScreen;
  SelectWorld(1);
  SelectWindow(1);              {select screen window}
  SetBackground(0);             {give it a black background}
  DrawSquare(20,55,1000,915,false);  {draw the border}
  DrawTextW(100,12,2,name);
  DrawTextW(800,12,2,screen_char);
  CopyScreen;

  SelectWindow(2);              {select cursor}
  SelectWorld(2);               {select it's world}
  SetBackground(0);             {give it a black background}
  InvertWindow;                 {turn the cursor on}
end; { New_screen }
{-----------------------------------------}
{ Draws the access arrows }

procedure Draw_arrow(x1,y1,x2,y2:real);

var
  slope : real;

{ These procedures handle drawing of the last section of an
access arow and the appropriate arrow-point }

procedure DrawArrow45(x1,y1,x2,y2:real);

begin
  if (x1 > x2) and (y1 > y2) then
  begin
    DrawLine(x1,y1,x2+5,y2+7.5);
    DrawLine(x2,y2+15,x2,y2);
    DrawLine(x2+10,y2,x2,y2);
    DrawLine(x2,y2+15,x2+10,y2);
```

```
        DrawLine(x2+5,y2+7.5,x2,y2);
      end else

      if (x1 < x2) and (y1 < y2) then
      begin
        DrawLine(x1,y1,x2-5,y2-7.5);
        DrawLine(x2,y2-15,x2,y2);
        DrawLine(x2-10,y2,x2,y2);
        DrawLine(x2,y2-15,x2-10,y2);
        DrawLine(x2-5,y2-7.5,x2,y2);
      end else

      if (x1 > x2) and (y1 < y2) then
      begin
DrawLine(x1,y1,x2+5,y2-7.5);
        DrawLine(x2,y2-15,x2,y2);
        DrawLine(x2+10,y2,x2,y2);
        DrawLine(x2,y2-15,x2+10,y2);
        DrawLine(x2+5,y2-7.5,x2,y2);
      end else

      if (x1 < x2) and (y1 > y2) then
      begin
        DrawLine(x1,y1,x2-5,y2+7.5);
        DrawLine(x2,y2+15,x2,y2);
        DrawLine(x2-10,y2,x2,y2);
        DrawLine(x2,y2+15,x2-10,y2);
        DrawLine(x2-5,y2+7.5,x2,y2);
      end;
    end; { DrawArrow45 }

      procedure DrawArrowHor(x1,y1,x2,y2 : real);

      begin
       if x2 > x1 then
       begin
         DrawLine(x1,y1,x2-10,y2);
         DrawLine(x2-10,y2-10,x2,y2);
         DrawLine(x2-10,y2+10,x2,y2);
         DrawLine(x2-10,y2-10,x2-10,y2+10);
         DrawLine(x2-10,y2,x2,y2);
       end
       else
       begin
         DrawLine(x1,y1,x2+10,y2);
         DrawLine(x2+10,y2-10,x2,y2);
         DrawLine(x2+10,y2+10,x2,y2);
         DrawLine(x2+10,y2-10,x2+10,y2+10);
         DrawLine(x2+10,y2,x2,y2);
       end;
      end;      { DrawArrowHor }
```

```
  procedure DrawArrowVer(x1,y1,x2,y2 : real);

  begin
   if y2 > y1 then
   begin
     DrawLine(x1,y1,x2,y2-15);
     DrawLine(x2-7,y2-15,x2,y2);
     DrawLine(x2+7,y2-15,x2,y2);
     DrawLine(x2-7,y2-15,x2+7,y2-15);
     DrawLine(x2,y2-15,x2,y2);
   end
   else
   begin
     DrawLine(x1,y1,x2,y2+15);
     DrawLine(x2-7,y2+15,x2,y2);
     DrawLine(x2+7,y2+15,x2,y2);
     DrawLine(x2-7,y2+15,x2+7,y2+15);
     DrawLine(x2,y2+15,x2,y2);
   end;
  end;   { DrawArrowVer }

begin { Draw_arrow }
  Move_cursor_out;
  if x2 = x1 then slope := 10.0
  else slope := abs((y2 - y1)/(x2 - x1));
  if slope <= 0.5 then DrawArrowHor(x1,y1,x2,y2)
  else if slope >= 2.0 then DrawArrowVer(x1,y1,x2,y2)
  else DrawArrow45(x1,y1,x2,y2);
  Move_cursor_in;
end; {Draw_arrow }

{-----------------------------------------}
{ Draws the object name in the object located
at x1, y1 }

procedure Draw_name(x1,y1:real; name : object_name);

var
short_name : short_obj_neme;

begin
  x1 := x1 - 35;
  y1 := y1 - 10;
  adjust_name(short_name,name);
  Move_cursor_out;
  DrawTextW(x1,y1,1,short_name);
  Move_cursor_in;
end; { Draw name }
{-----------------------------------------}
{ Draws the object symbols based on an approximate
center of x,y}
```

```
procedure Draw_object(which : char; x, y : real);

procedure Draw_std_object(x,y : real);

begin
  Move_cursor_out;
  DrawSquare(x-50,y-60,x+50,y+40,false);
  DrawSquare(x-50,y+40,x+50,y+80,false);
  Move_cursor_in;
end;  { Draw Std Object }

procedure Draw_generic(x, y : real);

begin
  Move_cursor_out;
  DrawLine(x-40,y-60,x+60,y-60);
  DrawLine(x+60,y-60,x+40,y+40);
  DrawLine(x+40,y+40,x-60,y+40);
  DrawLine(x-60,y+40,x-40,y-60);
  DrawLine(x-60,y+40,x-65,y+80);
  DrawLine(x-65,y+80,x+35,y+80);
  DrawLine(x+35,y+80,x+40,y+40);
  Move_cursor_in;
end;  { draw generic }

begin { draw object }
  case which of
    'g' : begin                { generic package }
            Draw_generic(x,y);
            Move_cursor_out;
            DrawTextW(x-38,y+53,1,'PACKAGE');
            Move_cursor_in;
          end;
    'h' : begin                { generic subprogram }
            Draw_generic(x,y);
            Move_cursor_out;
            DrawTextW(x-58,y+53,1,'SUBPROGRAM');
            Move_cursor_in;
          end;
    'p' : begin                { package }
            Draw_std_object(x,y);
            Move_cursor_out;
            DrawTextW(x-28,y+53,1,'PACKAGE');
            Move_cursor_in;
          end;
    's' : begin                { subprogram }
            Draw_std_object(x,y);
            Move_cursor_out;
            DrawTextW(x-45,y+53,1,'SUBPROGRAM');
            Move_cursor_in;
          end;
```

```
  end; { case }
end; { draw object }
{-----------------------------------------}
{ Selects the objects and arrows to be drawn
on the diagram indicated by diag_index and
uses Draw_object and Draw_arrow to draw them}

procedure Draw_diagram(diag_index : integer;
                       name : object_name);

var
  i,j : integer;
  x1, y1, x2, y2 : real;

begin
  for i := 1 to next_object - 1 do
  with object[i] do
  if diag_index = diagram then
  begin
    Draw_object(point.object_type, point.x, point.y);
    Draw_name(point.x, point.y, name);
  end
  else if diag_index = child_diag then
  begin
    Draw_object(point.object_type, child_pt.x, child_pt.y);
    draw_name(child_pt.x, child_pt.y, name);
  end;

  for i := 1 to next_arrow - 1 do
  with arrow[i] do
  if diag_index = diagram then
  begin
    x1 := point[1].x;
    y1 := point[1].y;
    j := 2;
    Move_cursor_out;
    while point[j].object_type = 'a' do
    begin
      x2 := point[j].x;
      y2 := point[j].y;
      DrawLine(x1,y1,x2,y2);
      j := j + 1;
      x1 := x2;
      y1 := y2;
    end;  { while }
    Move_cursor_in;
    x2 := point[j].x;
    y2 := point[j].y;
    Draw_arrow(x1,y1,x2,y2);
  end; { for with if }
```

```
end; { Draw_diagram }
{-----------------------------------------}
{ Displays system commands in a window }

procedure Help;

begin
  Move_cursor_out;
  StoreWindow(1);
  SelectWorld(4);
  SelectWindow(4);
  SetBackground(0);
  DefineHeader(4,'HELP INFORMATION');
  SetHeaderOn;
  DrawBorder;
  gotoxy(10,7);  writeln('DRAW COMMANDS');
  gotoxy(10,8);
  writeln('     a - defines origin and midpoints of',
          ' access arrows');
  gotoxy(10,9);
  writeln('     e - defines end-point of access arrows');
  gotoxy(10,10);
  Writeln('     p - draws package; s - draws subprogram');
  gotoxy(10,11);
  writeln('     gp - draws generic package;',
          '  gs - generic subprogram');
  gotoxy(10,12);
  writeln('     zi- zooms in on object selected by',
          ' cursor position');
  gotoxy(10,13);
  writeln('     zo- zooms out to parent diagram of',
          ' object selected');
  gotoxy(10,14);
  writeln('EDIT COMMANDS');
  gotoxy(10,15);
  writeln('     e - enters component specification',
          ' editing mode');
  gotoxy(10,16);
  writeln('    da - deletes access arrow originating at',
          ' the cursor');
  gotoxy(10,17);
  writeln('    do - deletes object selected by',
          ' cursor position');
  gotoxy(10,18);
  writeln('DISPLAY COMMANDS           ',
          '                        **************');
  gotoxy(10,19);
  writeln('     h - "HELP" describes',
          ' commands                  *');
  gotoxy(10,20);
  writeln('     v - displays selected object',
```

```
          ' specification    *   ends pgm');
   gotoxy(10,24);
   writeln('Press any key to return to access grapb');
   repeat until keypressed;
   gotoxy(1,24); writeln(' ':80);
   ClearScreen;
   RestoreWindow(1,0,0);
   Move_oursor_in;
end;  { Help }
{----------------------------------------}
{ Removes access from object[from_ind] to
object[to_ind] wben either an object[to_ind] is
deleted or the access arrow is deleted. }

procedure Remove_access(from_ind, to_ind : integer);

var i : integer;

begin
   i := 0;
   repeat
     i := i + 1;
   until object[from_ind].access[i].index = to_ind;
   object[from_ind].access[i].index := 0;
end; { Remove_access }
{----------------------------------------}
{ Determines which, if any, arrow begins at or
near coordinates findx, findy }

procedure Select_arrow(findx,findy : real;
                       var found : boolean;
                       var index : integer);

var i : integer;

begin
  found := false;
  i := 1;
  repeat
    with arrow[i] do
    begin
      if (point[1].x-10 <= findx) and
         (point[1].x+10 >= findx) and
         (point[1].y-10 <= findy) and
         (point[1].y+10 >= findy) then
      begin
        found := true;
        index := i;
      end; { if }
    end; { witb }
    i := i + 1;
```

```
  until found or (i >= next_arrow);

end; ( Select_arrow }
{-----------------------------------------}
{ Determines which, if any, object begins at or
near coordinates findx, findy }

procedure Select(findx, findy : real; var found : boolean;
                 var out_object : char;
                 var index : integer);

var i,j : integer;

begin
  found := false;
  i := 1;
  repeat
    with object[i] do
    begin
      if ((point.x-60 <= findx) and
          (point.x+70 >= findx) and
          (point.y-60 <= findy) and
          (point.y+90 >= findy) and
          (diagram = screen_num)) or
         ((child_pt.x-60 <= findx) and
          (child_pt.x+70 >= findx) and
          (child_pt.y-60 <= findy) and
          (child_pt.y+90 >= findy) and
          (child_diag = screen_num)) then
      begin
        found := true;
        out_object := point.object_type;
        index := i;
      end; ( if }
    end; { with }
    i := i + 1;
  until found or (i >= next_object);
end; ( procedure select }
{-----------------------------------------}
{ Erases the arrow indicated by index }

procedure Erase_arrow(object : char; index : integer);

var i,j : integer;
    x1,y1,x2,y2 : real;

begin
  for i := 1 to next_arrow do
  begin
    if ((arrow[i].from_index =  index) and (object <> 'a'))
       or ((arrow[i].to_index = index) and (object <> 'a'))
```

```
        or ((object = 'a') and (index = i)) then
      with arrow[i] do
      begin
        SetColorBlack;
        x1 := point[1].x;
        y1 := point[1].y;
        j := 2;
          Move_cursor_out;
        while point[j].object_type = 'a' do
        begin
          x2 := point[j].x;
          y2 := point[j].y;
          DrawLine(x1,y1,x2,y2);
          j := j + 1;
          x1 := x2;
          y1 := y2;
        end;  { while }
          Move_cursor_in;
        x2 := point[j].x;
        y2 := point[j].y;
        Draw_arrow(x1,y1,x2,y2);
        if (to_index = index) or (object = 'a') then
          Remove_access(from_index, to_index);
        Init_arrow(i);
      end; { with and if }
    end;{ for }
    SetColorWhite;
end;  { erase_arrow }
{-----------------------------------------}
{ Adds access of object[to_ind] to object[from_ind] }

procedure Add_access(from_obj, to_obj : ohar;
                     from_ind, to_ind : integer);

var i : integer;
    name : object_name;

begin
   name := object[to_ind].name;
   i := 0;
   repeat
     i := i + 1;
   until object[from_ind].access[i].index = 0;
   object[from_ind].access[i].index := to_ind;
end; { Add_access }
{-----------------------------------------}
{ Draws new arrows and puts data into arrow array,
calls Add_access }

procedure Read_arrow;
```

```
var
  x1, y1,
  x2, y2 : real;
  object : char;
  found : boolean;
  valid : boolean;
  index : integer;
  i : integer;
  from_object : char;
  from_index : integer;

begin    { Read_Arrow }
gotoxy(1,24); writeln(' ':80); writeln(' ':80);
x1:= x;
y1:= y;
i := 1;
valid := true;
select(x1,y1,found,object,index);

if found then
begin
from_ohject := object;
from_index := index;
arrow[next_arrow].diagram := screen_num;
arrow[next_arrow].from_index := index;
arrow[next_arrow].point[i].object_type := 'a';
arrow[next_arrow].point[i].x := x1;
arrow[next_arrow].point[i].y := y1;
i := 1 + 1;

repeat
  read(Kbd,Ch);          {read the keystroke}
  case ord(Ch) of
      97 : begin       { a }
             gotoxy(1,24);
             writeln(' ':80); writeln(' ':80);
             if i = max_arrow_points then
             begin
               gotoxy(3,24);
               write('This is the last point.',
                     ' Move cursor to end of arrow');
               writeln(' and press e');
             end else
             begin
               Move_cursor_out;
               x2 := x;
               y2 := y;
               arrow[next_arrow].point[i].object_type := 'a';
               arrow[next_arrow].point[i].x := x2;
               arrow[next_arrow].point[i].y := y2;
               DrawLine(x1,y1,x2,y2);
```

```
                x1 := x2;
                y1 := y2;
                i := i + 1;
                Move_cursor_in;
              end;
            end;

      101 : begin   { e }
             gotoxy(1,24); writeln(' ':80);
             select(x,y,found,object,index);
             if not found then
             begin
              gotoxy(3,24);
              writeln('Arrow does not end at an object. ',
              'Press a or move closer to object and press e');
              Ch := ' ';
             end;
            end;

       72,
       75,
       77,
       80 : Move_cursor;
    end;
until Ch = 'e';                {e ends arrow}
    x2 := x;
    y2 := y;
    Draw_arrow(x1,y1,x2,y2);
    arrow[next_arrow].to_index := index;
    arrow[next_arrow].point[i].object_type := 'e';
    arrow[next_arrow].point[i].x := x2;
    arrow[next_arrow].point[i].y := y2;
    Add_access(from_object, object, from_index, index);
    next_arrow := next_arrow + 1;
end else
begin
    gotoxy(3,24);
    writeln('Arrow does not start at an object.',
            ' Move closer to the object and press a');
end; { if object is found }
end; { Read_arrow }
{----------------------------------------}
{ Initiates deletion of an object or arrow }

procedure Delete;

var more : char;
    choice : char;
    x1,y1,x2,y2 : real;
    j,i : integer;
    found : boolean;
```

```
    in_object : cbar;
    index : integer;

begin  { delete }
  read(Kbd,more);
  case more of
    'o' : begin    { delete object }
            select(x,y,found,in_object,index);
            if found then
            begin                        { if found }
              gotoxy(3,24);
              write('Do you want to delete object ',
                     object[index].name,
                         '     y/n ?');
              read(Kbd, choice);
              gotoxy(1,24); writeln(' ':80);
              if choice = 'y' then
              begin
                SetColorBlack;
                Draw_object(in_object,
                            object[index].point.x,
                            object[index].point.y);
                Draw_name(object[index].point.x,
                          object[index].point.y,
                          object[index].name);
                Erase_arrow(in_object, index);
                SetColorWhite;
                Init_object(index);
              end;
              SetColorWhite;
            end; { if found }
          end;   { end delete object }

    'a' : begin    { delete arrow}
            Select_arrow(x, y, found, index);
            if found then with arrow[index] do
            begin          { if found }
                gotoxy(3,24);
                write('Do you want to delete this arrow',
                      '    y/n ?');
                for i := 1 to 2 do
                begin  { for  - blink arrow }
                  SetColorBlack;
                  x1 := point[1].x;
                  y1 := point[1].y;
                  j := 2;
                  Move_cursor_out;
                  while point[j].object_type = 'a' do
                  begin         { while a }
                    x2 := point[j].x;
                    y2 := point[j].y;
```

```
                    DrawLine(x1,y1,x2,y2);
                    j := j + 1;
                    x1 := x2;
                    y1 := y2;
                  end;  { while a, draw line segments }
                  Move_cursor_in;
                  x2 := point[j].x;
                  y2 := point[j].y;
                  Draw_arrow(x1,y1,x2,y2);
                  SetColorWhite;
                  x1 := point[1].x;
                  y1 := point[1].y;
                  j := 2;
                  Move_cursor_out;
                  while point[j].object_type = 'a' do
                  begin          { while a }
                    x2 := point[j].x;
                    y2 := point[j].y;
                    DrawLine(x1,y1,x2,y2);
                    j := j + 1;
                    x1 := x2;
                    y1 := y2;
                  end;  { while a, draw line segments }
                  Move_cursor_in;
                  x2 := point[j].x;
                  y2 := point[j].y;
                  Draw_arrow(x1,y1,x2,y2);
                end;      { for  - blink arrow }
                read(Kbd, choice);
                if choice = 'y' then
                   Erase_arrow('a', index);
                gotoxy(1,24);
                writeln(' ':80); writeln(' ':80);
              end; { if found }
  end; { case }
  end;
end; { Delete }
{----------------------------------------}
{ Reads in initial specification when a new object
is drawn. (Does the drawing too.) }

procedure Read_object(obj_type : char);

var
    name : object_name;
    entry : procedure_name;
    line_no : integer;
    next_entry : integer;
    type_proc : char;

procedure get_comments(var in_ptr : comment_ptr);
```

```
var current_com : comment_ptr;
    comment : comment_ptr;
    command : char;
    in_comment : string[60];

begin
  if line_no > 17 then
  begin
    for i := 11 to 20 do  { blank out information }
    begin
      gotoxy(10,i); writeln(' ':60);
    end;
    line_no := 11;
  end;
  gotoxy(10,line_no); writeln(' ':60);
  gotoxy(10,line_no);
  in_comment := '';
  writeln('Enter up to 58 characters of comment after',
          ' -- (or return)');
  line_no := line_no + 1;
  gotoxy(10,line_no); write('--'); readln(in_comment);
  line_no := line_no + 1;
  if in_comment <> '' then
  begin
    New(comment);
    comment^.line := '--' + in_comment;
    comment^.next := nil;
    current_com := comment;
    in_ptr := comment;
    repeat
      if line_no > 17 then
      begin
        for i := 11 to 20 do  { blank out information }
        begin
          gotoxy(10,i); writeln(' ':60);
        end;
        line_no := 11;
      end;
      gotoxy(10,line_no);
      write('--');
      in_comment := '';
      readln(in_comment);
      line_no := line_no + 1;
      if in_comment <> '' then
      begin
        New(comment);
        current_com^.next := comment;
        comment^.line := '--' + in_comment;
        comment^.next := nil;
        current_com := comment;
      end;
```

```
    until (in_comment = '');
  end; { if first comment <> '' }
end;   { get_comments }

procedure spec_entry;

var
    temp_in : string[10];
    i, j : integer;

begin
   Move_cursor_out;
   StoreWindow(1);
   SelectWorld(4);
   SelectWindow(4);
   SetBackground(0);
   DefineHeader(4,'SPECIFICATION ENTRY');
   SetHeaderOn;
   DrawBorder;
   gotoxy(10,7);
   writeln('Specification entry for component ',name);
   line_no := 8;
   gotoxy(10,line_no); line_no := line_no + 1;
   get_comments(object[next_object].comment);
   repeat
     for i := 8 to 20 do
     begin
        gotoxy(10,i); writeln(' ':60);
     end;
     line_no := 8;
     gotoxy(10,line_no); line_no := line_no + 1;
     gotoxy(10, line_no);
     write('procedure or function (p or f) ?',
           ' (return to bypass) : ');
     type_proc := ' ';
     readln(type_proc);
     line_no := line_no + 1;
     if (type_proc = 'p') or (type_proc = 'f') then
     begin
       object[next_object].proc[next_entry].proc_type
             := type_proc;
       if (object[next_object].point.object_type = 'p') or
          (object[next_object].point.object_type = 'g') then
       begin
         gotoxy(10,line_no);  write('Enter name : ');
         readln(entry);
       end else entry := 'KEY';
       { to indicate a subprogram so write }
       { and read display will access the  }
       { data for the subprogram }
       if type_proc = 'f' then
```

```
begin
  gotoxy(40,line_no); write('Returns ? : ');
  readln(object[next_object].
                proc[next_entry].f_returns);
end;
line_no := line_no + 1;
object[next_object].proc[next_entry].name := entry;
j := 1;
get_comments(object[next_object].
             proc[next_entry].comment);
repeat
  temp_in := '          ';
  gotoxy(13, line_no); write('Input : ');
  read(temp_in);
  if (temp_in[1] <> ' ') or (temp_in[2] <> ' ') then
  begin
    object[next_object].proc[next_entry].
          input[j].name := temp_in;
    gotoxy(33, line_no);
    write(' Type : ');  temp_in := '          ';
    readln(temp_in);
    object[next_object].proc[next_entry].
          input[j].in_type := temp_in;
  end;
  line_no := line_no + 1; j := j + 1;
  if line_no > 17 then
  begin
    for i := 11 to 20 do  { blank out information }
    begin
      gotoxy(10,i); writeln(' ':60);
    end;
    line_no := 11;
  end;
until ((temp_in[1] = ' ') and (temp_in[2] = ' ')) or
       ( j > max_inputs);
j := 1;
if type_proc <> 'f' then
repeat
  temp_in := '          ';
  gotoxy(13, line_no); write('Output : ');
  read(temp_in);
  if (temp_in[1] <> ' ') or (temp_in[2] <> ' ') then
  begin
    object[next_object].proc[next_entry].
          output[j].name := temp_in;
    gotoxy(33, line_no);
    write(' Type : ');  temp_in := '          ';
    readln(temp_in);
    object[next_object].proc[next_entry].
          output[j].out_type := temp_in;
  end;
  line_no := line_no + 1; j := j + 1;
  if line_no > 17 then
```

```
          begin
            for i := 11 to 20 do  { blank out information }
            begin
              gotoxy(10,i); writeln(' ':60);
            end;
            line_no := 11;
          end;
        until ((temp_in[1] = ' ') and (temp_in[2] = ' ')) or
              ( j > max_outputs);
        j := 1;
        if type_proc <> 'f' then
        repeat
          temp_in := '          ';
          gotoxy(13, line_no); write('In out : ');
          read(temp_in);
          if (temp_in[1] <> ' ') or (temp_in[2] <> ' ') then
          begin
            object[next_object].proc[next_entry].
                  inout[j].name := temp_in;
            gotoxy(33, line_no);
            write(' Type : ');  temp_in := '          ';
            readln(temp_in);
            object[next_object].proc[next_entry].
                  inout[j].inout_type := temp_in;
          end;
          line_no := line_no + 1; j := j + 1;
          if line_no > 17 then
          begin
            for i := 11 to 20 do  { blank out information }
            begin
              gotoxy(10,i); writeln(' ':60);
            end;
            line_no := 11;
          end;
        until ((temp_in[1] = ' ') and (temp_in[2] = ' ')) or
              ( j > max_inouts);
      end; { if a valid procedure name }
      next_entry := next_entry + 1;
    until  { procedures are bypessed }
       ((type_proc <> 'p') and (type_proc <> 'f'))
          { or maximum procedures have been specified }
       or (next_entry > max_procedures) or
       { or object is  subprogram - (procs not specified) }
       (object[next_object].point.object_type = 's') or
       (object[next_object].point.object_type = 'h');
    ClearScreen;
    RestoreWindow(1,0,0);
    Move_cursor_in;
end; { procedure spec_entry }

begin
```

```
   next_entry := 1;
   SelectWorld(1);
   SelectWindow(1);
   gotoxy(1,24); writeln(' ':80); writeln(' ':80);
   Draw_object(obj_type, x, y);
   gotoxy(3,24);
   write('Enter name : ');
   readln(name);
   adjust_name(short_name,name);
   Draw_name(x,y,short_name);
   object[next_object].point.object_type := obj_type;
   object[next_object].point.x := x;
   object[next_object].point.y := y;
   object[next_object].name := name;
   object[next_object].diagram := screen_num;
   object[next_object].id := next_object;
   gotoxy(1,24); writeln(' ':80); writeln(' ':80);
   {procedures and functions are only specified
       for packages, not subprograms}
   spec_entry;
   next_object := succ(next_object);

end; { Read_object }
{-----------------------------------------}
{ Creates or accesses the screen on which the
selected object is decomposed }

procedure Zoom_in;

var
    found : boolean;
    out_object : char;
    index : integer;
    new_diagram : boolean;

begin
 Select(x, y, found, out_object, index);
 if found then
 with object[index] do
 begin
   new_diagram := false;
   if child_diag = 0 then
   begin
     new_diagram := true;
     child_diag := next_diagram;
     next_diagram := succ(next_diagram);
   end;
   screen_num := child_diag;
   New_screen(name, screen_num);
   if new_diagram then
   begin
```

```
      gotoxy(3,24);
      writeln('Place cursor at location for ',name,
              ' and press h');
      repeat
        read(Kbd, Ch);
        Move_cursor;
      until Ch = 'h';
      Draw_object(point.object_type, x, y);
      child_pt.object_type := point.object_type;
      child_pt.x := x;
      child_pt.y := y;
      gotoxy(1,24); writeln(' ':80);
    end;
    if diagram = 0 then
      diagram := screen_num;
    Draw_diagram(child_diag,name);
  end
  else begin
     gotoxy(3,24); writeln('Object not found');
     repeat until keypressed;
     gotoxy(1,24); writeln(' ':80);
  end;
end; { Zoom_in }
{---------------------------}
{ Draws the diagram on which the selected
object was 1st drawn }

procedure Zoom_out;

var
    found : boolean;
    out_object : char;
    index : integer;
    new_diagram : boolean;
begin
 Select(x, y, found, out_object, index);
 if found then
   if object[index].diagram <> 0 then
   begin
     screen_num := object[index].diagram;
     New_screen(object[index].name, object[index].diagram);
     Draw_diagram(object[index].diagram,object[index].name);
   end
   else begin
     gotoxy(3,24);
     writeln(object[index].name,' has no parent');
     repeat until keypressed;
     gotoxy(1,24); writeln(' ':80);
   end
 else begin
    gotoxy(3,24); writeln('Object not found');
```

```
    repeat until keypressed;
    gotoxy(1,24); writeln(' ':80);
  end;
end; { Zoom_out }

program gtgalsgraph;

 {$I typedef.sys}              {these files must be}
 {$I graphix.sys}              {included and in this order}
 {$I kernel.sys}
 {$I windows.sys}
{$I gtgals.def}
{$I gtgals1.pas}
{$I gtgals2.pas}
var
  heaptop : ^integer;

{-------------------------------------}
{ Builds the Ada language specification from
the data in the object array for the selected object. }

procedure Gen_Ada(index : integer; var head : spec_ptr);

const
  gen_sub : string[26] = '   procedure DUMMY is new ';
  gen_pkg : string[24] = '   package DUMMY is new ';


var
   count, i, j, k : integer;
   current_line : spec_ptr;
   spec_line : spec_ptr;
   build_line : output_line;
   gen_line : array[1..max_accesses] of output_line;

procedure build_comments(in_ptr : comment_ptr);

var next : comment_ptr;

begin
  next := in_ptr;
  repeat
    spec_line^.line := next^.line;
    New(spec_line);
    spec_line^.next := nil;
    current_line^.next := spec_line;
    current_line := spec_line;
    next := next^.next;
  until next = nil;
end;    { build comments }
```

```
procedure build_parms(index, i : integer);

var j : integer;
    count : integer;

begin
 count := 0;
 with object[index] do
 begin
  for j := 1 to max_inputs do
  if proc[i].input[j].name <> '' then
  begin
    count := count + 1;
    if count = 1 then build_line := build_line + '('
    else begin
      build_line := build_line +'; ';
      spec_line^.line := build_line;
      New(spec_line);
      spec_line^.next := nil;
      current_line^.next := spec_line;
      current_line := spec_line;
      build_line := '                    ';
    end;
    build_line := build_line + proc[i].input[j].name;
    build_line := build_line +' : in ';
    build_line := build_line + proc[i].input[j].in_type;
  end;

  for j := 1 to max_outputs do
  if proc[i].output[j].name <> '' then
  begin
    count := count + 1;
    if count = 1 then build_line := build_line + '('
    else begin
      build_line := build_line +'; ';
      spec_line^.line := build_line;
      New(spec_line);
      spec_line^.next := nil;
      current_line^.next := spec_line;
      current_line := spec_line;
      build_line := '                    ';
    end;
    build_line := build_line + proc[i].output[j].name;
    build_line := build_line +' : out ';
    build_line := build_line + proc[i].output[j].out_type;
  end;

  for j := 1 to max_inouts do
  if proc[i].inout[j].name <> '' then
  begin
    count := count + 1;
```

```
      if count = 1 then build_line := build_line + '('
      else begin
        build_line := build_line +'; ';
        spec_line^.line := build_line;
        New(spec_line);
        spec_line^.next := nil;
        current_line^.next := spec_line;
        current_line := spec_line;
        build_line := '                    ';
      end;
      build_line := build_line + proc[i].inout[j].name;
      build_line := build_line +' : in out ';
      build_line := build_line + proc[i].inout[j].inout_type;
    end;

    if (proc[i].proc_type <> 'f') then
      if count > 0 then build_line := build_line + ');'
      else build_line := build_line +'; '
    else begin
      if count > 0 then build_line := build_line + ')';
      spec_line^.line := build_line;
      New(spec_line);
      spec_line^.next := nil;
      current_line^.next := spec_line;
      current_line := spec_line;
      build_line := '                    ';
      build_line := build_line +' return ';
      build_line := build_line + proc[i].f_returns;
      build_line := build_line + ';';
    end;
    spec_line^.line := build_line;
    New(spec_line);
    spec_line^.next := nil;
    current_line^.next := spec_line;
    current_line := spec_line;
  end; { with object [index] }
end; { build_parms }

begin
  New(spec_line);
  spec_line^.next := nil;
  head := spec_line;
  current_line := spec_line;
  build_line := 'with ';

  with object[index] do
  begin
    count := 0;
    j := 1;
    if comment <> nil then build_comments(comment);
    for i := 1 to max_accesses do
```

```
if access[i].index <> 0 then   { valid access }
begin
  case object[access[i].index].point.object_type of
    'p', 's' : begin           { build with clause }
                 count := count + 1;
                 if count > 1 then
                 begin
                   build_line := build_line + ', ';
                   spec_line^.line := build_line;
                   New(spec_line);
                   spec_line^.next := nil;
                   current_line^.next := spec_line;
                   current_line := spec_line;
                   build_line := '     ';
                 end;
                 build_line := build_line +
                   object[access[i].index].name;
               end;
        'g' : begin  { build package instantiations }
                 gen_line[j] := gen_pkg;
                 gen_line[j] := gen_line[j] +
                   object[access[i].index].name;
                 gen_line[j] := gen_line[j] + ';';
                 j := j + 1;
               end;
        'b' : begin {build subprogram instantiations }
                 gen_line[j] := gen_sub;
                 gen_line[j] := gen_line[j] +
                   object[access[i].index].name;
                 gen_line[j] := gen_line[j] + ';';
                 j := j + 1;
               end;
  end; { case }
end; { for accesses }
if lengtb(build_line) > 5 then
begin                          { link "witb" clause }
  build_line := build_line + ';';
  spec_line^.line := build_line;
  New(spec_line);
  spec_line^.next := nil;
  current_line^.next := spec_line;
  current_line := spec_line;
end;
build_line := '';
case point.object_type of  { build declaration line }
  'p' : begin
          build_line := 'package ';
          build_line := build_line + name;
          build_line := build_line + ' is';
          spec_line^.line := build_line;
          New(spec_line);
```

```
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
          end;
    's' : begin
            build_line := 'procedure ';
            build_line := build_line + name;
            build_parms(index, 1);
            New(spec_line);
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
          end;
    'g' : begin
            spec_line^.line := 'generic ';
            New(spec_line);
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
            build_line := 'package ';
            build_line := build_line + name;
            build_line := build_line + ' is';
            spec_line^.line := build_line;
            New(spec_line);
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
          end;
    'h' : begin
            spec_line^.line := 'generic ';
            New(spec_line);
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
            build_line := 'procedure ';
            build_line := build_line + name;
            build_parms(index, 1);
            New(spec_line);
            spec_line^.next := nil;
            current_line^.next := spec_line;
            current_line := spec_line;
          end;
  end; { case }             { end build declaration }
  build_line := '';
  for i := 1 to j - 1 do
  begin                     { link generic instantiations }
    spec_line^.line := gen_line[i];
    New(spec_line);
    spec_line^.next := nil;
    current_line^.next := spec_line;
    current_line := spec_line;
```

```
      end;

      if (point.object_type = 'p') or
         (point.object_type = 'g') then
      for i := 1 to max_procedures do
      if proc[i].name <> '' then
      begin          { valid procedure }
        if proc[i].comment <> nil then
           build_comments(proc[i].comment);
        if proc[i].proc_type = 'p' then
          build_line :=  '   procedure '
        else build_line :=  '   funotion ';
        build_line := build_line + proc[i].name;
        build_parms(index,i);
      end;  { if valid procedure }
      if (point.object_type = 'p') or
         (point.object_type = 'g') tben
      begin
        build_line :=  'end ';
        build_line := build_line + name;
        build_line := build_line + ';';
        spec_line^.line := build_line;
      end;
      spec_line^.next := nil;

    end; { with object }
end; { procedure Gen_Ada }
{--------------------------------}
{ Brings up the viewing window and cals Gen_ada
for the selected object }

procedure View_text;

const col = 10;

var
     current : spec_ptr;
     found : boolean;
     in_object : cbar;
     index, loop : integer;
     line_no : integer;
     head : spec_ptr;
     more : char;

begin
    line_no := 7;
    select(x,y,found,in_object,index);
    if found then
    begin
      Gen_ada(index, head);
      Move_cursor_out;
```

```
      StoreWindow(1);
      SelectWindow(4);
      DefineHeader(4,object[index].name);
      SetBackground(0);
      SetHeaderOn;
      DrawBorder;
      gotoxy(col,line_no);
      current := head;
      repeat
        if line_no > 19 then
        begin
          writeln('press escape key for more data');
          repeat
            read(Kbd, more);
          until ord(more) = 27;
          more := ' ';
          for loop := 7 to 20 do
          begin
            gotoxy(col, loop);
            writeln(' ':60);
          end;
          line_no := 7;
          gotoxy(col, line_no);
        end; { if information fills window }
        writeln(current^.line);
        line_no := line_no + 1;
        gotoxy(col,line_no);

        current := current^.next;
      until current = nil;
      gotoxy(10,24);
      writeln('Press any key to return to access graph');
      repeat until keypressed;
      gotoxy(1,24); writeln(' ':80);
      more := ' ';
      ClearScreen;
      RestoreWindow(1,0,0);
      Move_cursor_in;
    end {if object found }
    else begin
      gotoxy(3,24);
      writeln('Object not found.',
              '  Press any key to continue');
      repeat until keypressed;
      gotoxy(1,24); writeln(' ':80);
    end;
end;   { view text }
{---------------------------------------}
{ Allows editing a selected components specification }

procedure Edit;

const
```

```
    title_col = 10;

var
     command : char;
     comment : comment_ptr;
     exit : boolean;
     found : boolean;
     name_change : boolean;
     out_object : char;
     i, j, index : integer;
     new_diagram : boolean;
     line_no : integer;

procedure clear_window;
var i : integer;
begin
  for i := 10 to 20 do
  begin
    gotoxy(title_col,i);
    writeln(' ':60);
    line_no := 10;
  end;
end; { clear window }

procedure edit_comments(var in_ptr : comment_ptr);

var comment : comment_ptr;
    cur_comment : comment_ptr;
    in_comment : string[60];
    prev_comment : comment_ptr;

begin
  cur_comment := in_ptr;
  prev_comment := in_ptr;
  repeat
    gotoxy(title_col, line_no);
    if in_ptr = nil then
      write('-- ?')
    else
      write(cur_comment^.line,' ?');
    repeat
      read(Kbd, command);
    until (command = 'm') or (command = 'n') or
          (command = 'a') or (command = 'e');
    writeln(' ', command);
    line_no := line_no + 1;
    if (command = 'm') and (in_ptr <> nil) then
    begin
      in_comment := '';
      gotoxy(title_col, line_no);
      write('--');
```

```
      readln(in_comment);
      line_no := line_no + 1;
      if in_comment = '' then
        prev_comment^.next := cur_comment^.next
      else
        cur_comment^.line := '--' + in_comment;
    end; { if command = 'm' }
    if command = 'e' then exit := true;
    if command = 'a' then
    begin
      if in_ptr <> nil then
      while cur_comment^.next <> nil do
        cur_comment := cur_comment^.next;
      repeat
        in_comment := '';
        gotoxy(title_col, line_no);
        write('--');
        readln(in_comment);
        line_no := line_no + 1;
        if in_comment <> '' then
        begin
          New(comment);
          comment^.line := '--' + in_comment;
          comment^.next := nil;
          if in_ptr = nil then
          begin
            in_ptr := comment;
            cur_comment := comment;
          end
          else begin
            cur_comment^.next := comment;
            cur_comment := comment;
          end;
        end;
      until in_comment = '';
    end; { if command = 'a' for add }
    prev_comment := cur_comment;
    cur_comment := cur_comment^.next;
  until (exit) or (cur_comment = nil);
end; { edit_comment }


begin
  name_change := false;
  exit := false;
  line_no := 10;
  Select(x, y, found, out_object, index);
  if found then
  with object[index] do
  begin
    Move_cursor_out;
```

```
StoreWindow(1);
SelectWorld(4);
SelectWindow(4);
SetBackground(0);
DefineHeader(4,'COMPONENT EDITOR');
SetHeaderOn;
DrawBorder;
gotoxy(10,7);
writeln('m to modify an item. n to go to next item.',
        ' e to exit.');
gotoxy(10,8);
writeln('Enter m,n, or e after each ? prompt.');
gotoxy(10,9);
writeln('Enter a after --"comment..." ?',
        ' to ADD a comment.');
gotoxy(title_col,line_no);
write('OBJECT NAME : ');
write(name, ' ?');
repeat
  read(Kbd, command);
until (command = 'm') or
  (command = 'n') or (command = 'e');
writeln(' ',command);
line_no := line_no + 1;
if command <> 'e' then
begin
  if command = 'm' then
  begin
    name_change := true;
    gotoxy(title_col, line_no);
    write('Enter new OBJECT NAME  : ');
    name := '';
    readln(name);
    line_no := line_no + 1;
  end;
  edit_comments(comment);
  for i := 1 to max_procedures do
  if not exit then
  with proc[i] do
  begin
    clear_window;
    if (point.object_type = 'p') or
       (point.object_type = 'g') then
    begin
      gotoxy(title_col, line_no);
      write('Procedure or Function NAME : ');
      write(name, ' ?');
      repeat
        read(Kbd, command);
      until (command = 'm') or
        (command = 'n') or (command = 'e');
```

```
    writeln(' ',command);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Enter new NAME : ');
      name := '';
      readln(name);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
    edit_comments(comment);
  end; { if package or generic package }
  gotoxy(title_col,line_no);
  if not exit then
  begin
    gotoxy(title_col, line_no);
    write('(p)rocedure, (f)unction : ');
    write(proc_type, ' ?');
    repeat
      read(Kbd, command);
    until (command = 'm') or
      (command = 'n') or (command = 'e');
    writeln(' ',command);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Enter new choice (p)rocedure or',
            ' (f)unction : ');
      proc_type := ' ';
      readln(proc_type);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
    if (proc_type = 'f') and (not exit) then
    begin
      gotoxy(title_col, line_no);
      write('Function returns TYPE : ');
      write(f_returns, ' ?');
      repeat
        read(Kbd, command);
      until (command = 'm') or
        (command = 'n') or (command = 'e');
      writeln(' ',command);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
```

```
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Function will return what TYPE : ');
      f_returns := '';
      readln(f_returns);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
  end; { if function }
  if not exit then
  for j := 1 to max_inputs do
  if not exit then
  with input[j] do
  begin
    gotoxy(title_col, line_no);
    write('INPUT NAME : ');
    write(name, ' ?');
    repeat
      read(Kbd, command);
    until (command = 'm') or
      (command = 'n') or (command = 'e');
    writeln(' ',command);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Enter new INPUT NAME  : ');
      name := '';
      readln(name);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
    if not exit then
    begin
      gotoxy(title_col, line_no);
      write('INPUT TYPE : ');
      write(in_type, ' ?');
      repeat
        read(Kbd, command);
      until (command = 'm') or
        (command = 'n') or (command = 'e');
      writeln(' ',command);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
      if command = 'e' then exit := true;
      if command = 'm' then
      begin
```

```
      gotoxy(title_col, line_no);
      write('Enter new INPUT TYPE : ');
      in_type := '';
      readln(in_type);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
  end; { if not exit }
end; { for inputs }
if (not exit) and (proc_type <> 'f') then
for j := 1 to max_outputs do
if not exit then
with output[j] do
begin
  gotoxy(title_col, line_no);
  write('OUTPUT NAME : ');
  write(name, ' ?');
  repeat
    read(Kbd, command);
  until (command = 'm') or
    (command = 'n') or (command = 'e');
  writeln(' ',command);
  line_no := line_no + 1;
  if line_no > 20 then clear_window;
  if command = 'e' then exit := true;
  if command = 'm' then
  begin
    gotoxy(title_col, line_no);
    write('Enter new OUTPUT NAME : ');
    name := '';
    readln(name);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
  end;
  if not exit then
  begin
    gotoxy(title_col, line_no);
    write('OUTPUT TYPE : ');
    write(out_type, ' ?');
    repeat
      read(Kbd, command);
    until (command = 'm') or
      (command = 'n') or (command = 'e');
    writeln(' ',command);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Enter new OUTPUT TYPE : ');
```

```
      out_type := '';
      readln(out_type);
      line_no := line_no + 1;
      if line_no > 20 then clear_window;
    end;
  end; { if not exit }
end; { for outputs }
if (not exit) and (proc_type <> 'f') then
for j := 1 to max_inouts do
if not exit then
with inout[j] do
begin
  gotoxy(title_col, line_no);
  write('IN OUT NAME : ');
  write(name, ' ?');
  repeat
    read(Kbd, command);
  until (command = 'm') or
    (command = 'n') or (command = 'e');
  writeln(' ',command);
  line_no := line_no + 1;
  if line_no > 20 then clear_window;
  if command = 'e' then exit := true;
  if command = 'm' then
  begin
    gotoxy(title_col, line_no);
    write('Enter new IN OUT NAME : ');
    name := '';
    readln(name);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
  end;
  if not exit then
  begin
    gotoxy(title_col, line_no);
    write('IN OUT TYPE : ');
    write(inout_type, ' ?');
    repeat
      read(Kbd, command);
    until (command = 'm') or
      (command = 'n') or (command = 'e');
    writeln(' ',command);
    line_no := line_no + 1;
    if line_no > 20 then clear_window;
    if command = 'e' then exit := true;
    if command = 'm' then
    begin
      gotoxy(title_col, line_no);
      write('Enter new IN OUT TYPE : ');
      inout_type := '';
      readln(inout_type);
```

```
                line_no := line_no + 1;
                if line_no > 20 then clear_window;
              end;
            end; { if not exit }
          end; { for inouts }
        end; { if not exit after procedure name change }
        if (point.object_type = 's') or
          (point.object_type = 'h') then
          exit := true;
      end;  { if not exit from procedures }
    end; { if initial command not exit }
    ClearScreen;
    RestoreWindow(1,0,0);
    Move_cursor_in;
    if name_change then Zoom_out;
          { to redraw screen with new names if any }
  end  { if object found }
  else begin
    gotoxy(3,24);
    writeln('Object not found.  Press any key to continue');
    repeat until keypressed;
    gotoxy(1,24); writeln(' ':80);
  end;
end; { edit procedure }
{----------------------------------------}
{ Reads a display file and puts the information
into the data structure for use by GTGALS }

procedure Read_display(filename : filenames);

var
   in_file : text;
   code : char;
   obj_ind, proc_ind, access_ind,
   arrow_ind, pt_ind : integer;
   i, j : integer;

procedure read_comments(var in_ptr : comment_ptr);

var current_comment : comment_ptr;
    comment : comment_ptr;

begin
  new(in_ptr);
  current_comment := in_ptr;
  readln(in_file, current_comment^.line);
  current_comment^.next := nil;
  read(in_file, code);
  while code = 'c' do
  begin
    new(comment);
```

```
    current_comment^.next := comment;
    current_comment := comment;
    readln(in_file, current_comment^.line);
    current_comment^.next := nil;
    read(in_file, code);
  end;
end; { if comment }

begin
  assign(in_file,filename);
  reset(in_file);
  read(in_file, code);

  while (code = 'p') or
        (code = 's') or
        (code = 'g') or
        (code = 'h') do   { read in objects }
  begin
   read(in_file, obj_ind);
   with object[obj_ind] do
   begin
    point.object_type := code;
    id := obj_ind;
    readln(in_file, point.x, point.y,
          child_pt.x, child_pt.y);
    readln(in_file,diagram,child_diag);
    readln(in_file, name);
    if (diagram = next_diagram) then
       next_diagram := diagram + 1;
    proc_ind := 1;
    read(in_file, code);
    if code = 'c' then read_comments(comment);
    while code = '*' do    { read in procedures }
    begin
      readln(in_file,proc[proc_ind].proc_type,
             proc[proc_ind].name);
      if proc[proc_ind].proc_type = 'f' then
        readln(in_file, proc[proc_ind].f_returns);
      Left_justify(proc[proc_ind].name);
      read(in_file, code);
      if code = 'c' then
         read_comments(proc[proc_ind].comment);
      j := 1;
      while code = '?' do   { read inputs }
      begin
        readln(in_file, proc[proc_ind].input[j].name);
        readln(in_file, proc[proc_ind].input[j].in_type);
        read(in_file, code);
        j := j + 1;
      end;
      j := 1;
```

```
      while code = '!' do    { read outputs }
      begin
        readln(in_file, proc[proc_ind].output[j].name);
        readln(in_file, proc[proc_ind].output[j].out_type);
        read(in_file, code);
        j := j + 1;
      end;
      j := 1;
      while code = '+' do   { read inouts }
      begin
        readln(in_file, proc[proc_ind].inout[j].name);
        readln(in_file, proc[proc_ind].inout[j].inout_type);
        read(in_file, code);
        j := j + 1;
      end;

      proc_ind := succ(proc_ind);
    end; { while procedures }

    access_ind := 1;
    while code = '@' do    { read in access parameters }
    begin
      readln(in_file, access[access_ind].index);
      read(in_file, code);
      access_ind := succ(access_ind);
    end; { while access parameters }

  end; { with object }
end; { while objects }

next_object := obj_ind + 1;

arrow_ind := 1;
while not EOF(in_file) do        { read in arrows }
with arrow[arrow_ind] do
begin
  pt_ind := 1;
  while (code = 'a') or (code = 'e') do
  begin
    if (code = 'e') then next_arrow := succ(next_arrow);
    point[pt_ind].object_type := code;
    readln(in_file, point[pt_ind].x, point[pt_ind].y,
           diagram);
    read(in_file, code);
    pt_ind := succ(pt_ind);
  end;
  readln(in_file, from_index, to_index);
  arrow_ind := succ(arrow_ind);
  if not EOF(in_file) then read(in_file, code);
end; { while arrows }
next_arrow := arrow_ind;
```

```
  gotoxy(1,24); writeln(' ':80);
  gotoxy(3,24); writeln(temp_file,' retrieved');
  close(in_file);
end; { read_display }
{----------------------------------------}
{ Writes out the data from the data structures
to a uniquely formatted .gph display file }

procedure Write_display;

var
    filename : filenames;
    i,j : integer;
    index : integer;
    display_file : text;
    template_file : text;
    open_paren : boolean;
    pad_name, pad_type : integer;

procedure write_comments(in_ptr : comment_ptr);

var next : comment_ptr;

begin
  next := in_ptr;
  repeat
    writeln(display_file,'c',next^.line);
    next := next^.next;
  until next = nil;
end;

begin
   gotoxy(3,24);
   write(' Enter file name to save display file',
        ' (or return) :  ');
   temp_file := '';
   readln(temp_file);
   if temp_file <> '' then
   begin   { write display file to disk }
     filename := temp_file + '.gph';
     assign(display_file,filename);
     rewrite(display_file);

     for i := 1 to next_object do
     with object[i] do
     if id <> 0 then
     begin
       writeln(display_file,point.object_type:1,' ',
               id:3,' ', point.x:6:1,' ',point.y:6:1,' ',
               child_pt.x:6:1,' ', child_pt.y:6:1);
       writeln(display_file,diagram:2,' ',
```

```
            child_diag:2);
    writeln(display_file, name);
    if comment <> nil then write_comments(comment);
    for index := 1 to max_procedures do
    if proc[index].name <> '' then
    begin
      writeln(display_file, '*',proc[index].proc_type,
              proc[index].name);
      if proc[index].proc_type = 'f' then
         writeln(display_file,proc[index].f_returns);
      if proc[index].comment <> nil then
            write_comments(proc[index].comment);
      for j := 1 to max_inputs do
      if proc[index].input[j].name <> '' then
      begin
        writeln(display_file,'?',
                proc[index].input[j].name);
        writeln(display_file,
                proc[index].input[j].in_type);
      end;
      for j := 1 to max_outputs do
      if proc[index].output[j].name <> '' then
      begin
        writeln(display_file,'!',
                proc[index].output[j].name);
        writeln(display_file,
                proc[index].output[j].out_type);
      end;
      for j := 1 to max_inouts do
      if proc[index].inout[j].name <> '' then
      begin
        writeln(display_file,'+',
                proc[index].inout[j].name);
        writeln(display_file,
                proc[index].inout[j].inout_type);
      end;
    end;
    for index := 1 to max_accesses do
       if access[index].index <> 0 then
       writeln(display_file, '@ ',access[index].index);
end; { with and for }

for i := 1 to next_arrow do
with arrow[i] do
begin
  for index := 1 to max_arrow_points do
     if point[index].object_type <> ' ' then
       writeln(display_file,
               point[index].object_type:1,
               ' ', point[index].x:6:1,
               ' ', point[index].y:6:1,
               ' ',diagram);
```

```
          if from_index <> 0 then
            writeln(display_file, from_index:4, to_index:4);
      end; { with and for }
      gotoxy(1,24); writeln(' ':80);
      gotoxy(3,24);
      writeln('Display file ',temp_file,' saved');
      close(display_file);
      Delay(600);
    end;  { if file name }

end; { Write_display }
{--------------------------------}
{ Uses Gen_Ada for each object in the data
structure and writes it out to a .ada file }

procedure Gen_specs;

var
  current : spec_ptr;
  head : spec_ptr;
  i : integer;
  outfile : text;
  response : char;

begin
  gotoxy(3,24);
  write(' Enter y to create Ada language specification',
        ' (or return) :  ');
  response := ' ';
  readln(response);
  if response = 'y' then
  begin
    if temp_file = '' then
    begin
      gotoxy(1,24); writeln(' ':80);
      gotoxy(3,24);
      write('Enter name of specification file : ');
      readln(temp_file);
    end;
    temp_file := temp_file + '.ada';
    assign(outfile, temp_file);
    rewrite(outfile);
    for i := 1 to max_objects do
    if object[i].id <> 0 then
    begin
      Gen_ada(i, head);
      current := head;
      repeat
        writeln(outfile, current^.line);
        current := current^.next;
      until current = nil;
```

```
      writeln(outfile);
    end;
    close(outfile);
    gotoxy(1,24); writeln(' ':80);
    gotoxy(10,24);
    writeln('Ada language specification written to file ',
            temp_file);
    delay(900);
    gotoxy(1,24); writeln(' ':80);
    ClearScreen;
  end; { if specification file requested }
end;  { generating specification file }

{-----------------------------------------}
begin      { main program }

Init_structure;
InitGraphic;           {initialize the graphics system}
x := 500;
y := 500;
next_arrow := 1;
next_diagram := 2;
next_object := 1;
screen_num := 1;

DefineWorld(1,0,1000,1000,0);
          {give it a world coordinate system}
DefineWindow(2,trunc(XMaxGlb/2),trunc(YMaxGlb/2),
               trunc(XMaxGlb/1.995),trunc(YMaxGlb/1.995));
DefineHeader(2,'THIS IS THE CURSOR'); {give it a header}
DefineWorld(2,0,1000,1000,0);
          {give it a world coordinate system}
DefineWindow(3,trunc(XMaxGlb/10),trunc(YMaxGlb/1.8),
               trunc(XmaxGlb*9.3/10),trunc(YMaxGlb*9/10));
DefineWindow(4,trunc(XMaxGlb/10),trunc(YMaxGlb/6),
               trunc(XMaxGlb*9.3/10),trunc(YMaxGlb*5/6));
DefineWorld(4,0,80,25,0);

temp_file := '';
write('Enter name of old specification or',
      ' return for new specification :');
readln(temp_file);
if temp_file <> '' then
begin
  in_file_name := temp_file + '.gph';
  Read_display(in_file_name);
  long_file_name := temp_file;
  New_screen(temp_file,1);
  Draw_diagram(1,long_file_name);
end
else New_screen('GTGALS',1);
```

```
repeat
  read(Kbd,Ch);                {read the keystroke}
  case ord(Ch) of
      97 : Read_arrow;  { 'a' for arrow }
     103 : begin        { 'gp' for generic package }
                        { 'gs' for generic subprogram }
             read(Kbd,Ch);
             if Ch = 'p' then Read_object('g');
             if Ch = 's' then Read_object('h');
           end;
     112 : Read_object('p'); { 'p' for package }
     115 : Read_object('s'); { for subprogram }
     118 : View_text;    { 'v' for view }
     122 : begin
             read(Kbd, Ch);
             case Ch of
               'i' : Zoom_in;
               'o' : Zoom_out;
             end; { case }
           end; { Zoom }
     100 : Delete;       { 'd' for delete }
     101 : Edit;         { 'e' for edit }
     104 : Help;         { 'h' for help }
      72,
      75,
      77,
      80 : Move_cursor;
  end;
until Ch = '';          {  ohar exits program}
Write_display;
Gen_specs;
LeaveGraphic;                {leave the graphics system}
end.
```

A GRAPHIC TOOL FOR GENERATING ADA LANGUAGE SPECIFICATIONS

by

DONALD E. BODLE, JR.

B.S., Kansas State University, 1984

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1985

A Graphic Tool for Generating Ada Language Specifications

by Don Bodle

Abstract

Methods for specifying software systems have gained increasing attention as the size and complexity of computer applications has grown. The purpose of this paper is to present the current state of software specification techniques and to propose improvements in one component of these techniques, the user interface.

The use of automated tools for specification is described, with particular emphasis on their user interfaces. Many features of these tools are highlighted. From this study, a proposal for a graphic interface for software system specification is developed, describing the desirable features of such an interface. Finally, a prototype of the proposal is examined.