National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

Ottawa-Carleton Institute for Computer Science

Knowledge-Based Program Understanding & Testing Assistant (KBPUTA)

Computer Science Masters (M.C.S.) Thesis

J.D.Brett Hodges,
University of Ottawa,
Ottawa, Ontario,
Canada

© Brett Hodges 1993

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-95925-8

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGMENTS

The author wishes to thank his thesis advisor, Dr. Robert Probert of the University of Ottawa, for his valuable guidance in conducting this research. The author also thanks Hareton Leung, of Bell Northern Research, for his assistance in developing rules for the regression testing of global variables.

The author expresses his gratitude to the following people from Revenue Canada Taxation;

Richard Manicom
Assistant Deputy Minister, EDP Systems and Technology Branch

David Green
Director General, Human Resources Systems and Administration
(Formerly Director General, Individual Tax Systems)

Raymond Martinuk
Director, Advanced Technology Research Division

Valerie Wutti
Manager, Client Assistance Sector

Ken Vachon
Project Leader, Expert Systems Support Group

Without their support this research would never have seen fruition.

The author also thanks Trinzic Corporation and Revenue Canada Taxation for their permission to use the Aion Development System (ADS). ADS was used to build the proof-of-concept Knowledge-Based Program Understanding and Testing Assistant.

ABSTRACT

The Knowledge-Based Program Understanding and Testing Assistant (KBPUTA) is presented in this thesis. This assistant was designed and implemented as a proof-of-concept system. The purpose in developing this assistant is threefold; 1) to provide a generic design for a tool to aid in the maintenance task of program understanding, 2) to illustrate how regression testing assistance for a program modification can be provided, and 3) to show how such a tool can easily be built using a commercially available knowledge-based system development shell.

Maintenance is the most costly phase of the project life cycle and program understanding consumes more than half of the time spent on maintenance. Any reduction in this cost will free resources for software development.

The development of this proof-of-concept system passed through the following stages; 1) a examination of the research into program understanding, 2) examination of the research into regression testing, 3) selection of a knowledge-based representation to store the required knowledge, 4) development of rules to aid in regression testing, and 5) design and implementation of the proof-of-concept system. A summary of the results of each stage is provided in the body of this thesis paper.

A program can be viewed at many levels of abstraction, from high level specifications to source code. Our system design allows the user to view a program from three different views: from the domain view utilizing program goals, to an intermediate view utilizing program plans to a

lower-level view of module interdependencies and variable definition and use (data flow).

The major contributions of this thesis are:

1) the inclusion of domain knowledge in the specification of program goals. This will allow the maintenance assistant to provide three views of a program, a domain knowledge view, a program knowledge view, and a view that shows the mapping between the two.

2) development of a knowledge-based development shell independent design for a maintenance assistant.

3) providing regression testing assistance before the actual program modification is made. This is accomplished by advising the user what must be tested for a given change. This may cause the user to create another modification design that will have less impact on the program than the original change.

4) providing assistance on the regression testing of global variables. This is based upon the work by Leung and White (1990A). The development of the rules for the regression testing of global variables was assisted by Hareton Leung.

KBPUTA is implemented using the Aion Application Development Expert System Shell (ADS/WIN) version 6.2. ADS/WIN runs under Windows 3.0, or greater, on any PC with at least 4MB of RAM. The Aion Execution System (AES/WIN), vs 6.2, is required in order to run the proof-of-concept system.

TABLE OF CONTENTS

CHAPTER 1.0

INTRODUCTION TO SOFTWARE MAINTENANCE AND MOTIVATION FOR THESIS . . . 1
1.1 DEFINITION AND COST OF SOFTWARE MAINTENANCE . . . 1
1.2 PROGRAM UNDERSTANDING FUNCTION OF SOFTWARE MAINTENANCE . . . 3
1.3 THE MAINTENANCE PROBLEM . . . 4
1.4 LACK OF INTEREST IN DEVELOPING SOLUTIONS TO THE SOFTWARE MAINTENANCE PROBLEM . . . 5
1.5 CONTRIBUTIONS OF THESIS . . . 7
1.6 ORGANIZATION OF THESIS . . . 13

CHAPTER 2.0

BACKGROUND: PROGRAM UNDERSTANDING . . . 14
2.1 DEFINITION OF PROGRAM UNDERSTANDING . . . 14
2.2 LEVELS OF PROGRAM UNDERSTANDING . . . 15
2.3 STRATEGIES FOR PROGRAM UNDERSTANDING . . . 17
2.4 THEORIES OF PROGRAM ANALYSIS . . . 18
2.4.1 THE CODE - DRIVEN APPROACH . . . 19
2.4.2 THE PROBLEM-DRIVEN APPROACH . . . 19
2.5 A MENTAL MODEL FOR PROGRAM UNDERSTANDING . . . 20
2.5.1 INTRODUCTION TO COGNITIVE SCIENCE . . . 20
2.5.2 A MENTAL MODEL FOR PROGRAM UNDERSTANDING . . . 21

CHAPTER 3.0

BACKGROUND: OVERVIEW OF KNOWLEDGE-BASED PROGRAM UNDERSTANDING SYSTEMS . . . 24
3.1 INTRODUCTION TO KNOWLEDGE-BASED SYSTEMS . . . 24
3.2 USE OF KNOWLEDGE-BASED SYSTEMS AS ASSISTANTS . . . 26

CHAPTER 4.0

THE APPROACH TO PROGRAM UNDERSTANDING: PROGRAM PLANS AND GOALS 28
4.1 RESEARCH INTO TEXT COMPREHENSION 28
4.2 RESEARCH INTO PROGRAM PLANS AND GOALS 30
4.3 DEFINITIONS OF PLANS AND GOALS IN PROUST 32
4.4 DEFINITION OF PROGRAM PLANS IN THE PROGRAMMER'S APPRENTICE 36
4.5 FORMALIZATION OF PLANS AND GOALS IN KBPUTA 37
4.6 COMPARISON OF PLANS IN PROUST vs KBPUTA 38
4.7 COMPARISON OF PLANS IN PA VS KBPUTA 40
4.8 NAMING CONVENTIONS FOR PLANS AND GOALS IN KBPUTA 41

CHAPTER 5.0

MEANS OF PROVIDING STATIC PROGRAM MODIFICATION ASSISTANCE IN KBPUTA 42
5.1 BENEFITS OF PROVIDING STATIC ASSISTANCE 42
5.2 DEFINITION OF REGRESSION TESTING 43
5.3 THE REGRESSION TESTING MODEL 46
5.4 REGRESSION TESTING ASSISTANCE PROVIDED BY KBPUTA . . . 47

CHAPTER 6.0

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT DESIGN 51
6.1 OVERVIEW 51
6.2 THE MENTAL MODEL CREATED AND MAINTAINED BY THIS DESIGN 56
6.3 OVERVIEW OF FEATURES PROVIDED BY DESIGN OF THE ASSISTANT 59
6.4 OBJECTS AND ATTRIBUTES OF THE KBPUTA KNOWLEDGE BASE 60
6.5 RULES FOR REGRESSION TESTING 64

CHAPTER 7.0
EVALUATION OF THE APPROACH AND THE TOOL 69
7.1 SPECIFICATION OF TARGET PROGRAM 69
7.2 EXAMPLE PROGRAM USED IN THE KBPUTA PROTOTYPE 72
7.3 IMPLEMENTATION OF KBPUTA IN ADS 80
7.3.1 INTRODUCTION TO ADS 80
7.3.2 ARCHITECTURE OF KBPUTA AS IMPLEMENTED IN ADS 81
7.3.3 ASSESSMENT OF ADS AS THE APPLICATION DEVELOPMENT PLATFORM 87
7.4 ASSESSMENT OF USEFULNESS AND APPLICABILITY OF KBPUTA 87

CHAPTER 8.0

COMPARISON OF KBPUTA TO OTHER PROGRAM UNDERSTANDING SYSTEMS 90
8.1 PUDSY 90
8.2 PROUST 91
8.3 LASSIE 94
8.4 OTHER KNOWLEDGE REPRESENTATION SCHEMAS 96

CHAPTER 9.0

CONCLUSIONS 98
9.1 SUMMARY OF RESULTS 98
9.2 CURRENT VIEWS ON PROGRAM UNDERSTANDING SYSTEMS 99
9.3 OUR VIEW OF PROGRAM UNDERSTANDING SYSTEMS 100
9.4 UNIQUE CHARACTERISTICS OF OUR WORK 101
9.5 SUGGESTIONS FOR FUTURE RESEARCH 103

APPENDIX A: BIBLIOGRAPHY A-1

APPENDIX B: KBPUTA SAMPLE SESSION B-1

FIGURE LIST

Figure 4.1 Goal Translation 33

Figure 4.2 Actual Example of a Goal Translation 34

Figure 6.1: Program Specific Knowledge Of KBPUTA 53

Figure 6.2: Domain Knowledge of KBPUTA 54

Figure 6.3 High Level Conceptual Knowledge Model of KBPUTA 55

Figure 6.4: Mental Model Development Process 58

Figure 7.1: Architecture of KBPUTA 83

Figure 7.2: Example of an ADS Class 85

Figure 7.3: Example of an ADS Rule 86

<u>KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT</u>

CHAPTER 1.0

INTRODUCTION TO SOFTWARE MAINTENANCE AND MOTIVATION FOR THESIS

Software Maintenance is the most time consuming and costly phases of the software life cycle. In this chapter we define software maintenance and provide some statistics on its cost to an organization. We also discuss the role of program understanding in the maintenance process. We then present the maintenance problem and the lack of interest in developing tools to solve this problem. This is followed by a summary of the contributions of this thesis. Finally, we conclude the chapter with a section on the organization of this paper.

1.1 DEFINITION AND COST OF SOFTWARE MAINTENANCE

The U.S. National Bureau of Standards has defined ***software maintenance*** as "the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production." (NBS 1985, pg 2). There are three types of software maintenance, perfective, adaptive, and corrective as identified by Lientz and Swanson (Lientz and Swanson 1980). ***Perfective maintenance*** involves changes to software that are the result of new or modified requirements. ***Adaptive maintenance*** refers to maintenance that is required as a result of a change to the environment under which the software runs. ***Corrective maintenance*** refers to maintenance that is required as a result of errors or "bugs" which must

be corrected.

Maintenance is composed of three main functions, namely; understanding, modification, and testing. A maintainer must develop an understanding of software. This understanding comprises the functional and nonfunctional requirements, the interaction of components, as well as the mapping of functions to an end-user application. The modification must be then defined, designed, and coded. The resulting modified software must then be tested to ensure that the change is correct and that the software still performs according to specifications (Verification and Validation).

Maintenance consumes a large part of the data processing effort of most organizations. Lientz and Swanson performed a survey of data processing managers in 487 data processing organizations (Lientz & Swanson, Comm ACM, Nov, 1981) and found that software maintenance can consume up to seventy percent of the total costs expended during the life cycle of a system.

These statistics show that maintenance consumes a significant portion of an organization's scarce resources. Any improvements in software maintenance can free resources for use in development of new software.

1.2 PROGRAM UNDERSTANDING FUNCTION OF SOFTWARE MAINTENANCE

Program understanding has been defined as "the process of recovering high-level, functionality-oriented information from the source code" (Kozaczynski et al, 1991, pg 203). Understanding a program requires knowledge of the problem domain, knowledge of programming techniques, and knowledge of the system's environment. *Program understanding* also "involves detecting or inferring different kinds of relations between program parts" (Pennington 1987, p 295). The maintainer must be able to determine static connections such as those between modules, as well as knowledge connections such as that between the problem domain and its implementation.

The important role that *program understanding* plays in the maintenance process is shown by the fact that it has been estimated (Standish 1984), that 50 to 90 percent of maintenance time is devoted to program comprehension. Another study estimated that maintenance programmers spend between 47 to 62 percent of their time trying to comprehend code (Parikh, 1983).

In examining the percentage of time spent studying the documentation versus the source code, one survey, Fjelstrad & Hamlen 1983, found that the time spent on studying the code was three and a half times greater than the time spent on studying the documentation. As well this survey found that the amount of time spent studying the program was as long as that spent implementing the enhancement.

From the above statistics we can see that in order to have a big impact on reducing maintenance costs, the amount of time spent on *program understanding* should be reduced.

1.3 THE MAINTENANCE PROBLEM

In Section 1.1 it was shown that too much of an organization's valuable resources are spent on software maintenance. The problem for a significant number of maintainers who are required to maintain existing software is that not only do they have to work with poorly documented software, but a significant portion of the software does not follow the principles of structured programming. This makes it difficult for the maintainer to relate specific programming actions to specific code (Freedman 1980). Standard program listings are not designed to support reading for comprehension.

To compound the problem, additions to the original code have often been made without updating the limited original documentation. Sometimes these additions take the form of "quick fixes". These additions add to the difficulty of understanding the code and they often lead to the presence of "spaghetti code". This refers to code where it is almost impossible to follow the logic flow. Often, the maintainers who have performed these previous modifications did not have an adequate understanding of the program. This would result in a degradation of the program's organization.

Another problem occurs after the modification has been made. The modification to fix one bug often causes the propagation of other bugs (side effects). It is difficult to determine whether a change in code will affect the original code (Freedman 1980). This is because one of the major causes of errors in making modifications to a program occurs when an analysis of the program and its associated documentation do not reveal the sections of a program that are related even though they are physically far apart (Schneidewind 1987).

Frequently, maintainers are called upon to perform emergency fixes. Every minute that the system is inoperable costs an organization. These maintainers are under a high degree of pressure. Consequently, program modifications are made without gaining a proper understanding of the program. This leads to further periods when the system is "down".

1.4 LACK OF INTEREST IN DEVELOPING SOLUTIONS TO THE SOFTWARE MAINTENANCE PROBLEM

There has not been very much interest, until recently, in developing systems to aid in maintenance. Robert Glass (1992) notes there are two major reasons for this. Development is thought of being more interesting and fun and is thus more popular than maintenance. As well, most EDP organizations feel that development is more important than maintenance. This is also true of universities who present maintenance as an unimportant byproduct of the development process.

Glass reaches two conclusions:

1) the focus of managers, vendors, and academics must be changed so that software maintenance becomes a priority

2) the educational system should spend as much time on the reading of software as on the writing of it

The currently commercially available maintenance tools can be classified into one of the following groups (Wilde 1989):

* code analyzers
statically analyze the control structure and data flow of a program. Examples include Fastbol from TASC and the Cobol Structuring Facility from IBM.

* documentation aids
graphically analyze the program logic through the user of flow charts and data flow diagrams. Examples include Flobol from Cosmic and Cobol Glossary from MacKinney Systems.

* cross referencers
trace the use of data elements and named paragraphs or procedures through a program. Examples include CA-OPTIMIZER from Computer Associates and Scan/Cobol from Group Operations.

* restructurers
transform the an unstructured version of a program into a structured one. The structured version of the program is functionally equivalent to the unstructured one and will produce the same output. One example is Recoder from Language Technology.

* reformatters
intelligent editors that add indentation, line spacing and paging to a program. Recoder will also provide reformatting.

* Execution Monitors/Debuggers
tools that allow the programmer to interactively control the execution of the program. The values of data items may be displayed at any time during the execution. One example is XPEDITER from Application Development Systems Inc.

* Test Coverage Monitors
tools that allow the user to identify which lines of a program are executed for a given set of test data. CA-OPTIMIZER provides this type of assistance.

* Source Comparators
tools that allow the programmer to easily compare a modified version of a program with the original. A report showing lines that were changed is produced. Comparex from Sterling Software offers this type of assistance.

In our research, we did not come across any commercially available tools that were devoted to aiding in *program understanding*.

1.5 CONTRIBUTIONS OF THESIS

We have chosen to address the *maintenance problem* (Section 1.3). As *program understanding* consumes a large part of the maintenance task, it is in this area where we have decided to concentrate our efforts in providing a solution to the maintenance problem.

The major contribution of this thesis is the design and implementation of a proof-of-concept knowledge-based assistant, KBPUTA, to aid in *program understanding* and *regression testing* (defined on page 10). The design was based on recent research in *program understanding* (Letvosky 1986, Letvosky & Soloway 1986, Littman et al 1986, Soloway & Ehrlich 1984 and

Soloway 1986) and psychology (Sowa 1984, Schank 1983, and Schank 1984). Our assistant is designed to be as usable as possible. The knowledge provided will complement the maintainer's expertise.

The design of the assistant provides a framework that will allow any organization which is using knowledge-based system technology to easily build a software maintenance assistant. Once the basic knowledge is captured, it will be easy to add other knowledge such as a history of program abends (abnormal terminations), design decisions, names of people responsible for modules and rules for modifying modules. It is important to record what has been understood or else the effort spent on understanding a program will be lost.

The maintenance assistant is designed to handle programming languages that are procedure-oriented rather than object-oriented. This is because "Most business applications are written in Cobol and most of the ongoing maintenance in industry is performed on Cobol Systems" (Gibson & Senn 1989, pg 347). Cobol is a third generation language. However, the design can easily be modified to incorporate object-oriented constructs.

In order to focus on the main issue of modelling programming knowledge, we built KBPUTA using a knowledge-based development tool (ADS/WIN). This allowed us to pay less attention to the implementational aspects of the system that have nothing to do with the domain being modeled. We were able to concentrate on how to represent program knowledge and not on

building an inference engine.

Our contribution to aid in *program understanding* involves our modification of the traditional concepts of program goals and plans. We provide three different views of a program to aid in the understanding process. These views incorporate recent research results on the use of program goals and plans in *program understanding*. We have developed our own specification for program plans and goals that incorporate more domain knowledge than traditional specifications of program plans and goals (Soloway & Ehrlich 1984, Johnson and Soloway 1985, and Waters 1982). ***Domain knowledge*** refers to "knowledge about the world in which the target software is to operate" (Rich and Waters, 1986, p. xx) . Providing an increased level of domain knowledge it will enable the maintainer to obtain a better grasp of how the source code is mapped to a specific end-user application. Therefore, the effect of programming change can be mapped to its effect on the users of the program.

In order to load the KBPUTA knowledge base, the plans and goals must be entered manually. However, as noted by the developers of the LASSIE knowledge-based software information system (Devanbu et al 1991) concerning program requirements and design specifications, the cost of manually entering this knowledge is more than offset by the benefits of building the assistant.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Our research contributes a method of providing static *regression testing* assistance. After studying *program understanding*, we decided to expand the scope of our system to provide assistance in making the program modification after an adequate understanding of the program has been reached. We provide limited assistance in the *regression testing* of global variables.

Regression testing refers to the testing that occurs to a software system after a code modification has been made. According to the Gartner Group (1991), implementing a *regression testing* program can reduce, by up to 70 percent, the number of defects or problems reported with a system in the first 60 to 90 days of operation. The goal of *regression testing* is to insure that the change does not introduce undesirable side effects into the program. Our purpose was to illustrate how this assistance can be provided before the actual program modification is implemented. The potential of providing a regression testing assistant tool has not been explored by either the academic or the commercial world. Current regression testing tools only record "snap shots" of test transactions. These transactions are then be run at a later date in order to determine if the system still functions according to specifications.

We have reviewed the work done by Leung and White (1989, 1990, 1990A) on *regression testing*. We also worked with Leung in order to develop a set of rules. These rules are used to provide regression testing assistance for global variables to the user.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

One of the problems with building a knowledge-based system is the maintenance of the knowledge base by the developers and/or the users. We have addressed this issue in KBPUTA by providing a way for the maintainer to easily update the knowledge base. This serves to eliminate the need for a trained knowledge engineer to perform the knowledge acquisition.

The assistant is only a proof-of-concept system. However, with increased resources it is feasible to extend the system to understand large and/or complex systems that are currently too expensive to re-engineer or re-implement. It would be easy to modify the assistant to handle object-oriented programs. Knowledge on object-oriented constructs such as methods and classes could be easily represented.

Our assistant will help in training maintainers by providing them with access to a library of standard program plans. These maintainers will then be able to use the standard plans when implementing changes. This will allow for easier understanding of the programs by future maintainers.

This system should form part of a collection of tools that would make up a maintenance system. Other tools would include, graphical control and data flow displays, source code viewer, source editor, and a program change comparator.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Our research will hopefully shift the emphasis from providing development tools to providing maintenance tools. This is much more realistic since large systems are developed incrementally. As well, it will help shift the emphasis that CASE tools seem to have on a database repository to a knowledge-based repository.

To summarize, our research contributions revolve around the design of a knowledge-based program understanding and testing assistant (KBPUTA). The design is knowledge-based development shell independent. We have modified the traditional concepts of program goals and plans in order that their emphasis be shifted from programming knowledge to domain knowledge. This has allowed us to provide three different views of a program. We provide static *regression testing* assistance by using the assistant to advise the maintainer what must be tested for a given change for a global variable. The proof-of-concept system contains rules on the *regression testing* of global variables which is based upon the work of Leung and White (1989, 1990, 1990A). If the advice provided indicates to the maintainer that the potential change will have a wider impact than originally thought, the maintainer has the option of designing another change that will have less of an impact on the program.

1.6 ORGANIZATION OF THESIS

This document is organized in the following way; we first present relevant background information on *program understanding* and knowledge-based program understanding (Chapters 2 and 3). We then discuss our approach of providing program knowledge through the use of program plans and goals (Chapter 4). Relevant background information on *regression testing* and how this will be provided by KBPUTA is presented in Chapter 5. This is followed by a description of our generic design in Chapter 6. Chapter 7 presents an evaluation of our approach. This is followed by an illustration of KBPUTA working with an example program (Chapter 8). The next chapter deals with a discussion of our research. This is followed by a description of other program understanding systems. Finally we present our conclusions in Chapter 9.

Appendix A contains the Bibliography. Appendix B contains the screens from a sample KBPUTA session.

CHAPTER 2.0

BACKGROUND: PROGRAM UNDERSTANDING

In this chapter we introduce the concept of *program understanding*. We provide a definition of *program understanding* and outline the different levels of knowledge required to understand a program. We discuss the theories behind *program understanding* and program analysis. Finally, we introduce the cognitive science concept of a mental model and show how it has been applied to *program understanding*.

2.1 DEFINITION OF PROGRAM UNDERSTANDING

Recall from Section 1.2: ***Program understanding*** has been defined as "the process of recovering high-level, functionality-oriented information from the source code" (Kozaczynski et al, 1991, pg 203). *Program understanding* also "involves detecting or inferring different kinds of relations between program parts" (Pennington 1987, p 295).

A maintainer must develop an understanding of the software system, its internal structure, and its operational requirements in order to perform a maintenance change. Consequently, there is a direct correlation between the degree of comprehension of the program by the maintainer and

the chance that the modifications will be correct. If the program is not well understood, then there is a good chance that new program errors will be introduced if a program fix or an enhancement is made.

Programs contain a wealth of knowledge. A program contains different types of knowledge such as knowledge of the application domain, knowledge of a particular problem being solved, general programming knowledge, constraints imposed by the hardware and software environment, a history of modifications, and design decisions.

It is hard for a maintainer who has never been exposed to a program before to quickly and accurately obtain all this knowledge. In order to thoroughly understand a program a maintainer needs to examine different views of the program at any one time. These views range from the view of the end user to a view of the source code.

2.2 LEVELS OF PROGRAM UNDERSTANDING

Different levels of understanding are involved in fully understanding a program. From a high level view of how the program functions in the application domain, to a low level view of the source code. This can be illustrated in examining the following Pascal code segment:

```
sum :=0;
Read(X);
While X <> 999 Do
      sum := sum + X;
Write(sum);
```

Three levels of program understanding can be identified for this segment.

1) The understanding required to manually execute the code.

2) The understanding required to recognize that the segment sums an unknown number of values and prints the result when 999 is encountered.

3) The understanding required to recognize that the segment determines the number of students with overdue library books at the University of Ottawa.

The first type of understanding requires knowledge of the programming language. The second type of understanding represents a mapping from the source code to a standard programming problem-solving technique, a methodology, or an algorithm. The third type of understanding represents a mapping between the problem-solving technique and a very specific end-user application. The mapping is between the application domain and the programming domain.

Different types of understanding are required depending upon the maintenance task to be performed.

2.3 STRATEGIES FOR PROGRAM UNDERSTANDING

In their article "Mental Models and Software Maintenance" Littman et al (1986) determined from their research that there are two strategies for *program understanding*; the systematic strategy and as-needed strategy. The strategy that a maintainer uses to study a program will strongly influence the knowledge obtained about that program.

The ***systematic strategy*** involves tracing data flow and control flow through the program in order to understand the global behaviour of the program. This strategy is obviously not feasible for large programs as it would consume too much time.

The ***as-needed strategy*** involves an attempt to "localize parts of the program for which changes can be made that will implement the modification" (Littman et al 1986, pg 91). An attempt is made to understand the area of the program where the change is to be made. It can be easily seen that the disadvantage of this strategy is that the effects of the change on other parts of the program are not identified. This was also noted by Littman et al (1986). This study showed that maintainers who used the systematic approach to study a program performed more successful modifications then those who used the as-needed strategy. Those using the as-needed strategy were "unlikely to detect interactions in the program that might affect or be affected by the modification" (Littman et al, 1986, p. 81).

2.4 THEORIES OF PROGRAM ANALYSIS

There are two approaches to program analysis: the code-driven approach and the problem-driven approach.

Using the ***code-driven approach*** the maintainer begins with the code and "forms a more abstract description of what the individual parts of the program do and then what the entire program does." (Sevoira 1987, pg 21). This abstract model is constantly modified as the maintainer examines the code. This approach has also been labelled the bottom-up approach. The initial proponents of this model were Shneiderman and Mayer (1979) and Basili and Mills (1982).

Under the ***problem-driven approach***, the maintainer examines the specifications for the program and forms a hypothesis of what the program is supposed to do. This hypothesis is then verified against the code. This approach has been labelled the top-down approach by its original proponent Brooks (Brooks, 1983).

2.4.1 THE CODE - DRIVEN APPROACH

Empirical studies have shown that when involved in reading a program, maintainers tend to "chunk" or group parts of the program together. The chunks consist of program segments and the chunks are mentally organized based on the functionality of the code (Adelson, 1981). These chunks have been labelled plans by Soloway (1984). ***Beacons*** (Brooks, 1983) refer to the indicators that serve to indicate the presence of a particular structure or operation. Another study by Weiser (Weiser, 1982), showed that programmers often break up the program into "program slices". ***Program slices*** are sets of statements that are related by their flow of data.

The maintainer uses these "chunks" to form a hypothesis on the function of parts of the program. This leads to an overall hypothesis on the function and purpose of the entire program.

2.4.2 THE PROBLEM-DRIVEN APPROACH

Under the problem-driven approach, the maintainer begins by making an overall hypothesis concerning the function of the program. This hypothesis can be formed by reading the comments at the start of the program or by examining documentation, the program name and any input or output files. The hypothesis will lead the maintainers to expect particular structures or operations to appear in the program. The maintainer will then attempt to validate the

hypothesis by searching the program for features that designate the presence of these structures. These structures have been labelled beacons by Brooks (1983). For example, if a function is performed on a variable called index, then the maintainer will expect hashing to be present.

2.5 A MENTAL MODEL FOR PROGRAM UNDERSTANDING

In this subchapter, we provide a brief introduction to cognitive science, specifically to the concept of mental models. We then discuss the proposed mental model for program understanding.

2.5.1 INTRODUCTION TO COGNITIVE SCIENCE

Cognitive psychology can be considered as the study of problem solving. Basili and Musa (1991) state that cognitive psychology can be applied to the study of the intellectual activities involved in the software engineering. The authors further state that "To date, very little research based on cognitive psychology has been performed in software engineering, but there is substantial evidence of its promise" (pg 93). This statement is based on the fact that software engineering can be considered as a problem-solving process. We have used the aspects of cognitive psychology as applied to *program understanding* to design our maintenance assistant.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Sowa (1984) has defined a ***mental model*** as an abstraction of reality that accurately represents the thing in question as well as actions that can be performed by it and on it. According to Sowa, one of the basic premises of A.I. is that knowledge of something is the ability to form a mental model that accurately represents the thing as well as the actions that can be performed by it and on it (Sowa 1984). Then by testing actions on the model, a person can predict what is likely to happen in the real world. A mental model can be considered an abstraction of reality.

In the next section, we will examine the mental model that has been proposed by the program understanding process.

2.5.2 A MENTAL MODEL FOR PROGRAM UNDERSTANDING

A ***mental model for program comprehension*** was developed from a study on the process of program comprehension performed by Letvosky (1986). A maintainer can be considered as a ***knowledge-based understander***. This understander consists of three components:

knowledge base: encodes the expertise and program knowledge that a programmer brings to the understanding task

mental model: encodes the programmers current understanding of the target program

assimilation process: interacts with the program code and documentation and the knowledge base to construct the mental model

The knowledge base consists of program language semantics, goals of the program, and program plans (the implementation of the goals), efficiency knowledge, domain knowledge, and discourse rules (stylistic program conventions). Program goals and plans will be discussed in the next chapter (Chapter 4).

The mental model component should consist of the following:

specification: an explicit description of the goals of the program

implementation: an explicit, complete description of the actions and data structures in the program

annotation: (mapping) an explanation of how the each goal in the specification is accomplished.

In their article on Mental Models and Software Maintenance, Littman et al (Littman et al, 1986) discuss the knowledge required to understand a program and state that:

> [I]n order to understand a program, a programmer must know about the objects the program manipulates and the actions the program performs. In addition, the programmer must have knowledge about the functional components, comprised of functionally-related actions, which accomplish the tasks in the program (Littman et al, 1986, pg 83).

They refer to the above-mentioned knowledge as ***static knowledge*** because those aspects of a program do not change as the program executes. The knowledge that the programmer must

possess concerning the casual connections between functional components as the program executes is called ***casual knowledge***. Casual knowledge allows the programmer to reason about how the program's functional components interact during execution.

Littman et al define ***weak mental models*** of a program as those models containing only static knowledge of the program. ***Strong mental models*** are those containing both casual and static knowledge. Their study found that programmers who used solely the as-needed strategy obtained static knowledge and hence formed only weak mental models.

As mentioned previously (Chapter 3.3), the systematic approach obtained better results than the as-needed approach. A method of assisting maintainers working with large programs in constructing strong mental models must be developed because the systematic approach is not feasible for large programs. Our assistant, detailed later on, connects program goals to plans and plans to program source code. This will allow the programmer to fully understand the area of the program that is being changed. The causal knowledge provided by the assistant will allow the maintainer to note any interactions this area has with other areas of the code.

CHAPTER 3.0

BACKGROUND: OVERVIEW OF KNOWLEDGE-BASED PROGRAM UNDERSTANDING SYSTEMS

In this chapter, we introduce knowledge-based program understanding systems. We also discuss the role of knowledge-based systems acting as human assistants.

3.1 INTRODUCTION TO KNOWLEDGE-BASED SYSTEMS

We have chosen to use Artificial Intelligence techniques in developing our system. Software engineering researchers, such as Richard Waters, have discovered that "AI techniques make it possible to represent a great deal of knowledge about programming in general and then use this knowledge to understand particular programs" (Waters, 1985).

Feigenbaum (1993), had stated that ***knowledge*** is not synonymous with information, rather knowledge is information that has been interpreted, categorized, applied and revised. Knowledge consists of descriptions of definitions, symbolic descriptions of relationships, and procedures to manipulate both types of descriptions (McGraw & Harbision-Briggs 1989, pg 13). The use of an object hierarchy is one method of representing this knowledge and the method that is used in our generic design of KBPUTA.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

A ***knowledge-based system*** is the term applied to information systems in which some symbolic representation of human knowledge is applied, usually in a way which resembles human reasoning.

As early as 1978, Osterweil and Taylor had discovered the "central nature and importance of a data base of total, up-to-the minute information about a program under development or maintenance" (Osterweil & Taylor 1978, pg 36). The advent of knowledge-based technology has meant that not only can the above be accomplished, but the data can be represented as knowledge. The inferencing system of the knowledge-based system can be used to provide advice to a user, based on the content of the knowledge base and the rules available.

The expertise provided by the original developers and maintainers of a software system can be stored in the knowledge-based system for distribution to future maintainers who can benefit from this knowledge.

The Rome Laboratory has been working on a knowledge-based system that would address the entire software life cycle. This is the Knowledge-Based Software Assistant (KBSA) project (White, 1991). The goal is to build a knowledge base that will "capture the history of the life cycle processes and support automated reasoning about the software under development" (White, 1991, p. vi).

3.2 USE OF KNOWLEDGE-BASED SYSTEMS AS ASSISTANTS

We have decided to build an assistant rather than a fully automated system. The research into knowledge-based systems has recently shifted its emphasis from automatic programming to human augmentation (Fischer et al 1992). We can say that 1 human + 1 knowledge-based assistant is > 2 or in other words, a person and a machine can do more than either can do alone.

A system to replace maintainers would have a knowledge base containing domain knowledge, program knowledge, programming practices, program language information, etc. As we can see, this knowledge base would be quite large. The system would take a significant amount of time to respond to a query. A system that assists maintainers will combine the general knowledge of programming constructs and programming language syntax of the maintainer with the specific knowledge of a piece of software contained in the assistant.

Our maintenance tool will follow the trend toward human augmentation. It is designed to help maintainers, not replace them. The assistant and the human should complement one another. The decision on whether to use an assistant is up to the maintainer. If the programmer is familiar with the piece of software under study or the software is easily understood, then it would be unnecessary to employ the assistant. However, if the software under study is complex or is new to the maintainer, then the assistant will save the maintainer time in understanding the

program and analyzing the effects of a modification.

The interaction with the assistant will all the user to learn about the software under study while they are maintaining it. This will aid the user in making future changes. As well, in pressure situations where the maintenance change is critical, the fact that the system is in place will act as a calming influence on the maintainer. They will not look at an unfamiliar program and become flustered and make changes haphazardly with predictably costly results.

A ***knowledge-based program understanding system*** is described as a "knowledge-based approach to supporting understanding-intensive tasks in software maintenance and re-engineering" (Kozacyzynski et al, 1991, p. 203). These systems contain a representation of a program and other knowledge that is deemed important in the understanding process.

Recent knowledge-based systems which work with programming knowledge include: Knowledge-Based Software Assistant (White 1991) which is being developed by the Rome Laboratory, and BAL/SRW (Kozacynski et al 1991) which is a knowledge-based software re-engineering workbench developed by Anderson Consulting. Knowledge-based systems devoted to *program understanding* include PROUST (Johnson), LASSIE (Devanbu & Brachman 1991) and PUDSY (Lukey 1980). In a subsequent chapter we compare these systems to KBPUTA.

CHAPTER 4.0
THE APPROACH TO PROGRAM UNDERSTANDING: PROGRAM PLANS AND GOALS

In Chapter 2.0 we introduced the concept of *program understanding*. There is a substantial body of psychological research that supports the goal and plan concepts. We begin this chapter by briefly examining the research into text comprehension. We then examine the concept of program plans and goals as applied to *program understanding*. Further, we discuss how program plans and goals are specified in KBPUTA. We then compare their specification in KBPUTA with that of other knowledge-based program understanding systems.

4.1 RESEARCH INTO TEXT COMPREHENSION

The research into *program understanding* has its basis in the research that has been performed into text comprehension. In studying how people know what to expect in a given situation, researchers have found that people use background knowledge (Schank, 1984). This background knowledge is in the form of ***knowledge structures***. These structures include ***scripts, plans and goals***.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Scripts are a set of expectations that are applied in common situations (Schank, 1982). They are used to tell us what might happen next in a chain of events. Scripts direct our inference process and they are built up over time.

In reading a story, we must find the goals of the characters in the story, we then determine what plans are being used to realize each goal. In order to understand a story fully, it is necessary to infer a set of possible plans in which the characters in the story are likely to use to achieve their goals. In order to infer these plans, we must be able to generate them. In other words we must make guesses about the intentions behind actions in an unfolding story (Schank 1984).

To use the knowledge that we have about goals and plans, the reader must recognize the implicit connections between the two in what is being read. In order to understand a character's plans, it is necessary to understand the goal that a given plan is intended to achieve. If a reader lacks knowledge of specific plans, then they will have trouble understanding stories that implicitly use them.

From the above background information on text comprehension, we can easily see then how the concept of plans and goals has been applied to *program understanding*. In understanding a program, the reader of the code is attempting to understand how the specifications are implemented in the code. In terms of plans and goals, the goals would be the specifications and the plans would be their implementation.

4.2 RESEARCH INTO PROGRAM PLANS AND GOALS

Letvosky and Soloway (1986, pg 41) state as follows:

> the "goal of program understanding is to recover the intentions behind the code. We use the term "goal" to denote intentions and the term "plan" to denote techniques for realizing intentions".

As in text comprehension, Letvosky has defined a ***goal*** as being the intentions behind the code, and a plan as a technique for realizing these intentions. Letvosky (1986), defines ***goals*** a being recurring and computational and include search, sort, and delete. Goals are described independently of the algorithms that compute them. Goals can be decomposed into subgoals. Letvosky also notes that plans can range from low level components such as a counter to more domain specific plans. Plans and goals are not language dependent.

A ***plan*** is a program fragment that represents a stereotypical action sequence in programming. For example there are running total loop plans, and item search loop plans. Plans can be considered to be composed of events (Harandi & Qing, IEEE, 1988). Rich (1981), has defined plans as reusable patterns of data flow and control flow.

For example (Letvosky 1986), the goal of swapping two variables would be translated into the following plan:

Swap (VAR1, VAR2)

Event1 : move VAR1 into TEMP

Event 2: move VAR2 into VAR1

Event 3: move TEMP into VAR3

Algorithms are composed of one or more programming plans. For example, a mergesort algorithm is composed of:

- a plan for recursion on a binary tree;
- a plan for splitting a sequence in two; and
- a plan for merging sorted lists.

Plans are implemented using ***rules of discourse*** that represent programming conventions. Using variable names that represent their function, only putting in code that will to be used by the program and avoiding the use of "go to's" are examples of programming conventions. These rules of discourse set up expectations in the minds of the programmers about what will be contained in the program.

A study performed by Letvosky and Soloway (1986) showed that maintainers had difficulty understanding code in programming plans they call delocalized plans. ***Delocalized plans*** are programming plans realized by lines scattered throughout different parts of the program.

A maintainer when writing a program, will translate a program goal or subgoal into a set of plans in order to implement that goal or subgoal. This can be considered as ***program synthesis***. The reversal of this process can be used to achieve *program understanding*. This is illustrated in Figure 4.1. Figure 4.2 shows an actual example. As programmers gain experience they have been shown to learn a repertoire of plans.

4.3 DEFINITIONS OF PLANS AND GOALS IN PROUST

A programming tutor, PROUST, developed by Johnson (1990) makes use of the concept of plans and goals. PROUST analyzes a program, looking for bugs and bad programming style. The use of goals and plans follows the definition by Letvosky and Soloway above.

Goals are developed from the problem statement of the program and are decomposed into plans. In other words a ***goal decomposition*** relates a goal that a program is supposed to achieve with the plans that achieve it. For every statement in a program analyzed by PROUST, the goal decomposition indicates what goal that statement serves to implement and how the goal fits into

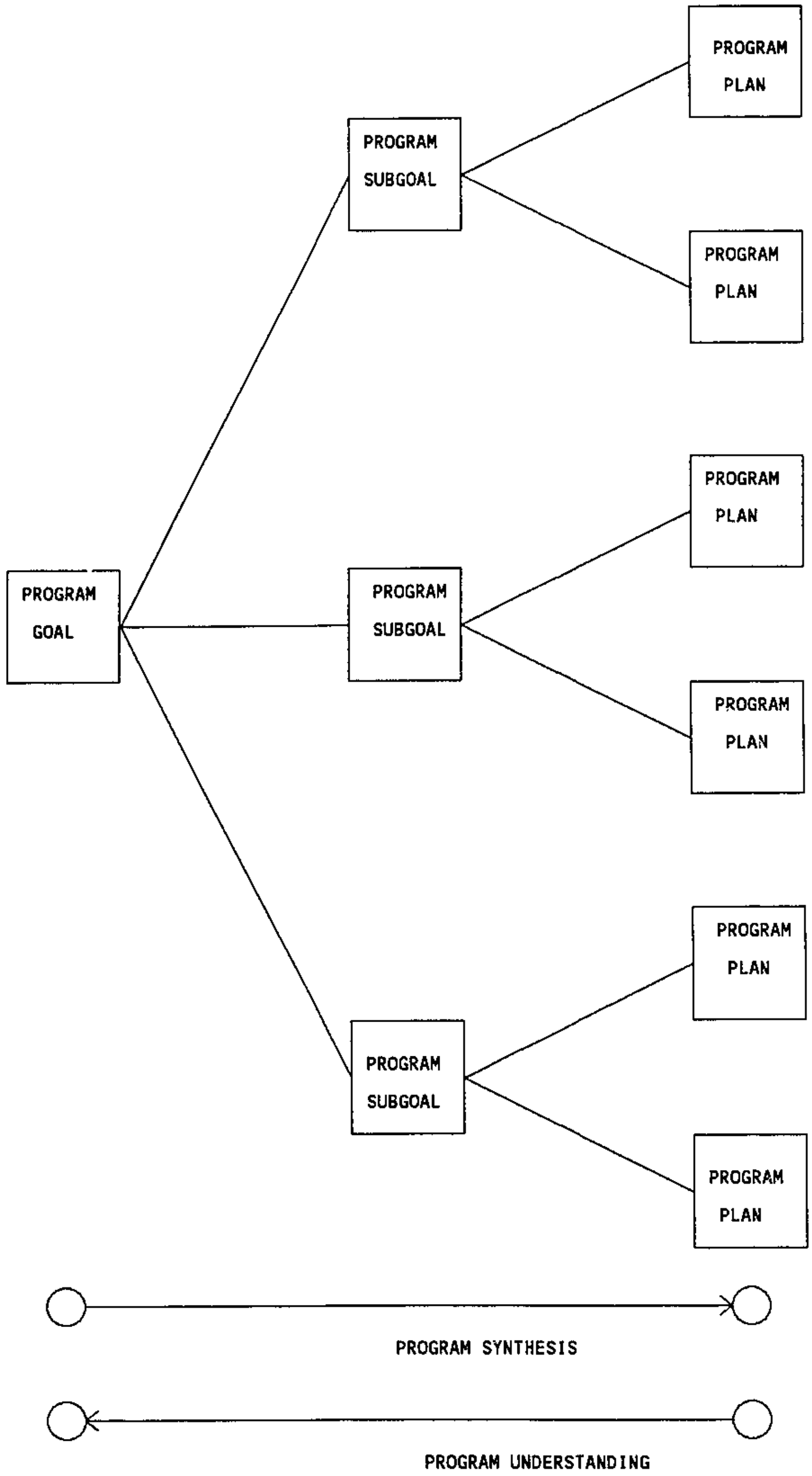


Figure 4.1: Goal Translation

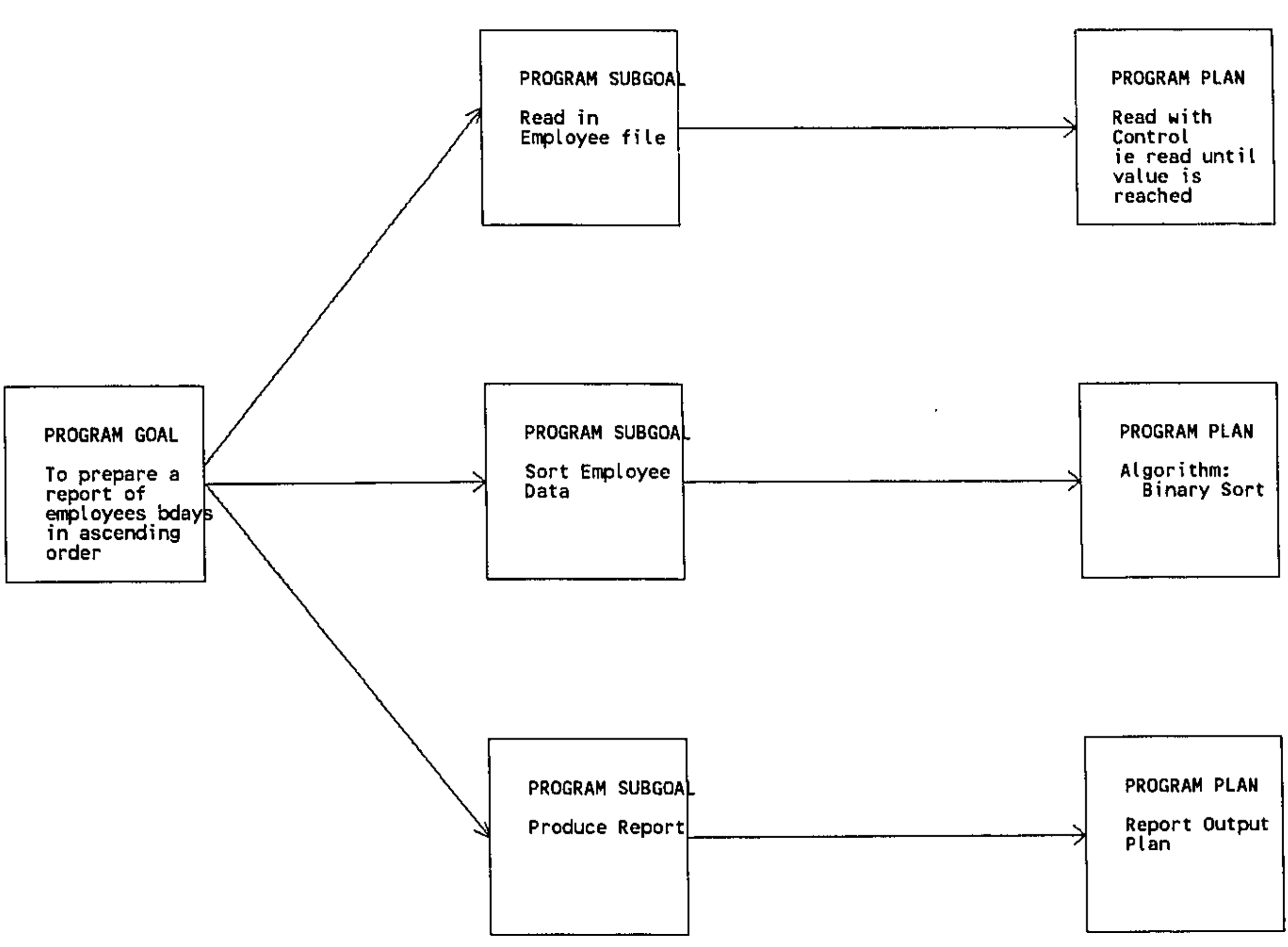


Figure 4.2: Actual Example of Goal Translation

goal fits into the overall procedure for solving the problem.

Johnson (1990), provides an example of how goals and plans are specified in PROUST. The example program is the Rainfall Problem. This program will compute the average rainfall per day, the number of rainy days, and the number of valid inputs. The problem requires that the students write a program that reads in a series of numbers which represent the average rainfall on a given day.

A specification of the Sentinel-Controlled Input Sequence goal is as follows (Johnson 1990, pg 64-65):

Name: Sentinel-Controlled Input Sequence

Description: read data and process it until a sentinel value is input

Implementations: Sentinel Process-Read While Plan
Sentinel Read-Process While Plan
Sentinel Read-Process Repeat Plan
Sentinel Process-Read Repeat Plan
Bogus Yes-No Loop
Bogus Counter-Controlled Loop

Each of the above implementations can be used to implement the Sentinel-Controlled Input sequence goal.

A specification of the Sentinel Read-Process Repeat Plan is as follows:

Name: Sentinel Read-Process Repeat Plan

Description: a repeat loop in which an Input subgoal, a Sentinel Guard subgoal, and a set of computations on the input are found

This specification includes subgoals that must be decomposed into their corresponding plans.

4.4 DEFINITION OF PROGRAM PLANS IN THE PROGRAMMER'S APPRENTICE

The Programmer's Apprentice (Waters 1982) also makes use of a structure called a plan. In Waters system a plan is describe as a "representation for programs which abstracts away the from the unessential features of a program and represents the basic logical properties of the algorithm explicitly" (pg 2). Plans are common algorithmic fragments. The basic unit of a plan is a segment, which is a unit of computation with both input and output ports. These ports specify the input values and the output values that the segment produces. It also includes a set of specifications which specify preconditions for the inputs that are required in order for the segment to execute correctly. Postconditions specify what will be true after the segment is executed.

4.5 FORMALIZATION OF PLANS AND GOALS IN KBPUTA

Let us return to the example Pascal code segment first presented in Section 3.2:

```
sum :=0;
Read(X);
While X <> 999 Do
      sum := sum + X;
Write(sum);
```

To recap: Three levels of *program understanding* can be identified for this segment.

1) The understanding required to manually execute the code.

2) The understanding required to recognize that the segment sums an unknown number of values and prints the result when 999 is encountered.

3) The understanding required to recognize that the segment determines the number of students with overdue library books at the University of Ottawa.

The second type of understanding represents a mapping from the source code to a "standard" programming problem-solving technique or methodology. We consider these techniques or methodologies to be *"programming plans"*.

The third type of understanding represents a mapping between the programming methodologies and a very specific end-user application. We consider these end-user applications to be the program's *"goals"*.

<u>KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT</u>

We want to shift the emphasis on goals and plans from program knowledge to domain knowledge. By doing this, we can provide the third type of understanding.

4.6 COMPARISON OF PLANS IN PROUST vs KBPUTA

We can see from the definition of a goal in PROUST, that the third type of understanding is not provided. The Sentinel-Controlled Input Sequence Goal is not related to the domain knowledge implicitly found in the Rainfall Program. It does not provide the maintainer with a connection between the end-user application and the program.

The purpose of our KBPUTA is to provide aid to the maintainer in performing *program understanding*. To this end, we feel that it is necessary to provide a link between the domain knowledge and the program.

The specification of goals and plans in PROUST and KBPUTA will also be different since PROUST is designed to help novice programmers learn to write programs. The goals and plan representations in PROUST are designed to be machine readable. Our plan and goal representations are designed to be read by human maintainers.

Another difference from PROUST is that in KBPUTA, plans will not be composed of subplans. If plans were composed of subplans, the maintainer would be required to review a description of the top level plans as well as the composing goals. This would complicate the understanding process. As well, since we are providing information on only one program at a time, each subgoal will only be implemented by one plan. We will not store alternate implementations as is done in PROUST. Again, this is an attempt to reduce the complexity of the understanding process.

In order to further contrast the definition of plans and goals in PROUST from that in KBPUTA we can use the example of the rainfall problem. A goal in KBPUTA would be specified as:

Name: Input Rainfall Data

Description: To input all of the rainfall data

Implemented By: Sentinel Controlled Input Sequence

A plan in KBPUTA would be specified as:

Name: Sentinel Controlled Input Sequence

Used in: Module Input_Rain_Fall

Related Goal: Input Rainfall Data

If the maintainer is not familiar with a given plan or goal, the maintainer will be able to review an explanation of each one. The maintainers can be trained to recognize plans in the code in

order to use them to record their understanding of a program.

In summary it is our intention to make the goals in our system more domain specific than those specified in PROUST. We will be using the goals to link domain specific knowledge to how that knowledge is implemented. Therefore, the concept of goals as expressed in PROUST is analogous to the concept of plans in the KBPUTA. Plans in PROUST are analogous to program concepts in KBPUTA.

4.7 COMPARISON OF PLANS IN PA VS KBPUTA

The plans used in the Programmer's Apprentice are used differently then the ones in KBPUTA. The plans in KBPUTA are used for *program understanding*, while the plans in PA are used for program building. Since the plans in PA are used for program building, they must be explicitly defined. Some of these plans include, successive approximation, square root, and sequence of summation.

4.8 NAMING CONVENTIONS FOR PLANS AND GOALS IN KBPUTA

The naming of plans and goals in KBPUTA will be user-dependent. We will provide a methodology for choosing their names but since they are many types of end-use applications, a standard set of plans and goals would be impossible. Soloway (1986) has noted that plan names can be awkward.

The goal of a program should be labelled by a brief phrase on the program's purpose. For subgoals, we recommend using the functions of modules as the basis for determining the goals of a program. Decide what each module actually does in terms of the end-user application. For example, if the module sorts values, what values are they? What do the values mean to the end-user?

For plans, we recommend using the names of algorithms or other well-known methods of manipulating data. Every environment has their own unique names for these kinds of methods. Explanation for the plans can be easily provided to the maintainer if they require an explicit definition of the plan. However, we recommend, as does Soloway (1986), that explicit labels be used for frequently used plans.

CHAPTER 5.0

MEANS OF PROVIDING STATIC PROGRAM MODIFICATION ASSISTANCE IN KBPUTA

In the previous chapter we discussed how we intend to use KBPUTA to provide program understanding assistance. In this chapter we show how KBPUTA will help in the next step of the maintenance task, namely, designing a maintenance change. An introduction to *regression testing* is provided along with the knowledge used by KBPUTA for the *regression testing* of global variables. This knowledge is in the form of rules. It will be used to illustrate how KBPUTA can reduce the testing effort by providing relevant knowledge before the actual implementation is performed.

5.1 BENEFITS OF PROVIDING STATIC ASSISTANCE

We believe that for a program understanding tool to be useful, it must be designed to support the process of maintenance, i.e. program modification. Towards this aim, we have decided to incorporate the work of Leung and White regarding *regression testing* (Leung & White 1989, 1990, 1990A). A summary of their work is presented in Section 5.2.

Basically, what we want to provide is assistance in making a change. This assistance will be provided by having the assistant advise the user on what should be tested for a given change.

In this way, the user may decide not to make his chosen change because of the impact that it will have. The assistant may advise him to test fifty modules. The maintainer might then decide to look at another possible modification that would require the testing of a fewer number of modules.

For the purposes of this proof-of-concept system. This information will be provided for the *regression testing* of global variables. However, it can be easily extended to handle other types of testing.

5.2 DEFINITION OF REGRESSION TESTING

Regression testing has been defined as;

> the testing that is performed after making a functional improvement or repair to the program. Its purpose is to determine if the change has regressed other aspects of the program. It is usually performed by running some subset of the program's test cases. Regression testing is important because changes and error corrections tend to be much more error prone than the original code in the program (Myers, pg 122, 1979).

Leung and White (1989) have identified two types of *regression testing*, namely progressive and corrective. ***Progressive regression testing*** involves testing a modified specification while ***corrective regression testing*** involves testing a modification in which the specification has not changed.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

In deciding upon the type of regression testing assistance to provide, we reviewed the paper by Hartman and Robson (1990) which summarized the work on *regression testing* to date.

Hartman and Robson (1990) have classified regression testing techniques into two groups; namely structural and functional. ***Structural regression testing*** uses knowledge of the program's construction, while ***functional regression testing*** uses knowledge of the system's requirements. They further identify two main problems in *regression testing*:

1) Maintaining the test suite -- this refers to maintaining the test suite after a system modification. The problem lies in identifying test cases which are not required to test the modification.
2) Selection of the test suite

Leung and White (1989) have approached the second problem by recommending that only those structural tests that execute the changed structures or routines containing those structures should be reexecuted and analyzed. They say that this reveals if the old tests that executed the modified programs sections are now not required.

Leung and White also recommend the use of function tables which provides a mapping between each functions and the functional test cases that execute them. This mapping can be provided for in our assistant by adding an attribute to the module object (see Chapter 8).

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Fischer's (as described by Hartman and Robson 1990) work involves linear programming. It requires information on connectivity and reachability of test case coverage of program segments and their data dependencies. This information is obtained from a general data flow analysis for the program. This information is stored in matrices. We have decided that this information is at too low of a level for our assistant.

Leung and White have also done work on the *regression testing* of global variables (1990A). They define a ***global variable*** as "a variable referenced by a module other than the one containing its declaration" (pg 209). As the authors note, the use of global variables lead to an increase in the effort required to understand a program. As well they create "data flow dependencies between modules which are not directly callable" (pg 209). It is hard to determine the complete effect of a module without seeing how the global variables in that module affect other modules. The author's recommend, not surprisingly, that global variables should be tested during unit and integration *regression testing*.

5.3 THE REGRESSION TESTING MODEL

Leung and White (1990, 1990A) have developed a testing model. For the purposes of this model, the testing process is composed of three phases: unit testing, integration testing, and systems testing. Each testing phase should concentrate on detecting errors unlikely to be detected by the previous testing phase. The following definitions are used in this model:

Use of a variable
-- an instruction in which the variable is referenced

Definition of a variable
-- an instruction which assigns a value to the variable

Definition Clear Path of a variable
-- a path in the control flow graph of a program which contains no definition of v

Directly used in module M
-- a global variable used by an instruction other than a call instruction in M. M is termed the using module.

Directly defined in module M
-- the global variable is defined by an instruction rather than a call instruction. M is called the defining module.

Reachability
-- a definition of v at module M will reach another module n if
i) there is a definition clear path from the definition of v to the exit point of A
ii) there is a module invocation sequence from M to N such that v is not redefined on at least one program path between the definition of v at M and the use of V in N

Killing
-- a definition of a variable v, which reaches a module M is killed by M if M redefines v on all paths through M.

Transferability
-- a module M is denoted as a transferring module for a given variable v if
i) v is an input parameter of M
ii) M is not a using or defining module of v.

Define-use pair
-- occurs for a pair of modules M and N where M is a defining module and B is a using module of variable v such that the definition of v in A may reach B.

We will incorporate this information into our knowledge base. It is our supposition that making this knowledge available to the maintainer will increase their understanding of a program. With this knowledge, the chance that the modification to a global variable or a parameter will create inconsistencies in other parts of the program will be significantly reduced.

5.4 REGRESSION TESTING ASSISTANCE PROVIDED BY KBPUTA

We have chosen to provide assistance in the *regression testing* of global variables. Global variables are widely used. If they are not tested during integration or unit testing than their testing will not be performed other than by coincidence (Leung & White 1990A).

The authors present a strategy for the *regression testing* of global variables in each of these phases. Global variables can be treated as parameters for testing purposes. Our system will provide this information to the user when a modification requires that one or more global variables be tested. This knowledge is in the forms of rules and is presented in Section 6.5.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

The knowledge to be provided includes;

* where a given global variable is directly used
* where a given global variable is directly defined
* using modules of a given global variable
* defining modules of a given global variable
* transferring modules of a given global variable
* killing modules of a given global variable

Our assistant will aid in controlling the need for *regression testing* by aiding the maintainer in the determination of how pieces of code are affected by others. Thus the maintainer can perform modifications without causing negative side effects. Linkages to other components will be identified.

For example, suppose a maintainer has a modification to perform. They already have developed a design for the modification of a global variable, X. They can use the assistant to determine what effect the modification of a global variable X will have on the rest of the program. The maintainer will be able to use the assistant to determine the other locations in which the definition of this global variable is used. This same information could have been provided through execution of the program with test cases. However, it is much more efficient to change the modification at the design stage then after it has been implemented in the program. The ability to perform this type of change consequence analysis before actual implementation will

reduce the chances of an error being found during *regression testing*.

The current maintenance process model is one of:

1) Review program code and its associated documentation in order to understand the program
2) Design and implement the required modification
3) Use Regression Testing to test the modification
4) Analyze the results of 3 ...
    a) results could point to a correct modification
    or
    b) results could lead maintainer back to step 1 or step 2.

In this process model, potential inconsistencies are pinpointed after the changes have been made. This problem has been shown to be NP-hard (Horowitz 1988)

Our aim to reduce the chance that the maintainer would have to go back to either step 1 or step 2 in the above process model. We want to provide the maintainer with knowledge so that they can better design their modification.

Each define-use module pair can be determined from intra-procedural data flow analysis. In fact a tool to provide this kind of information is available (Harrold & Sofa 1988). There is not currently a tool to automatically provide information on killing, transferring, and defining modules. However, such a tool should not be difficult to build.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

KBPUTA will also have the ability to guide a maintainer step by step through the unit and integration *regression testing* of a program. Our proof-of-concept system will perform this valuable assistance for global variables (see Section 6.5).

CHAPTER 6.0

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT DESIGN

In this chapter we present the design of our knowledge-based maintenance assistant. As discussed previously, it is our intention to make the design shell-independent. The mental model that will be created by the maintainer as a result of their interaction with the assistant is also described.

6.1 OVERVIEW AND DESIGN RATIONALE

From the current research, we decided what we wanted our design to provide. We provide the maintainer with the ability to use either the *systematic strategy* or the *as-needed strategy* or a combination of both (See Section 3.3). In order to make a modification, the maintainer is able to quickly find out where the change should be made and also to determine where testing is required.

The maintainer has the capability to either perform a *code-driven approach* or a *problem-driven approach*. This is accomplished by providing knowledge in the form of program plans, goals and subgoals. The maintainer is able to review a list of program goals and view how and where

these are implemented. The maintainer is also able to look at the information for a module and then determine what program plans, goals or subgoals are implemented by this particular module by using the assistant.

Our system provides static analysis. Static analysis has been defined as "the process of examining program source text in order to infer characteristics of the execution behaviour of the program specified by the text. It is also used to detect errors in programs" (Osterweil, page 406).

The assistant also provides different views of the program. From an application domain view to a code level view. This is also accomplished by the use program plans and goals.
Code level views of a program are provided using program specific knowledge. Figure 6.1 shows the inputs required in producing program specific knowledge.

Domain knowledge is contained in program goals. Figure 6.2 illustrates the inputs required in developing the goals of the program. A high level conceptual knowledge model of the program is shown in Figure 6.3. This shows the three types of knowledge that can be provided by KBPUTA. The mapping between the domain knowledge and the program knowledge is provided through the use of program plans.

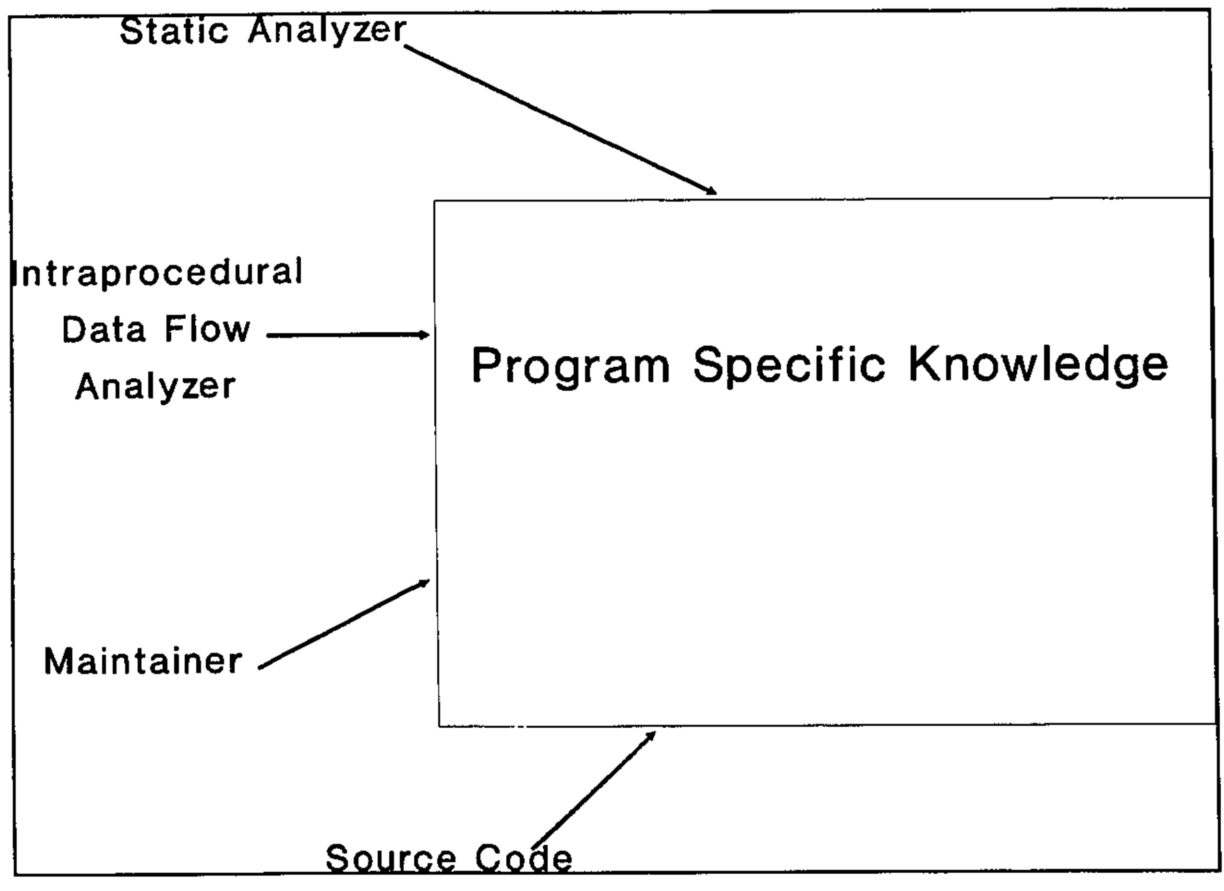


Figure 6.1: Program Specific Knowledge Of KBPUTA

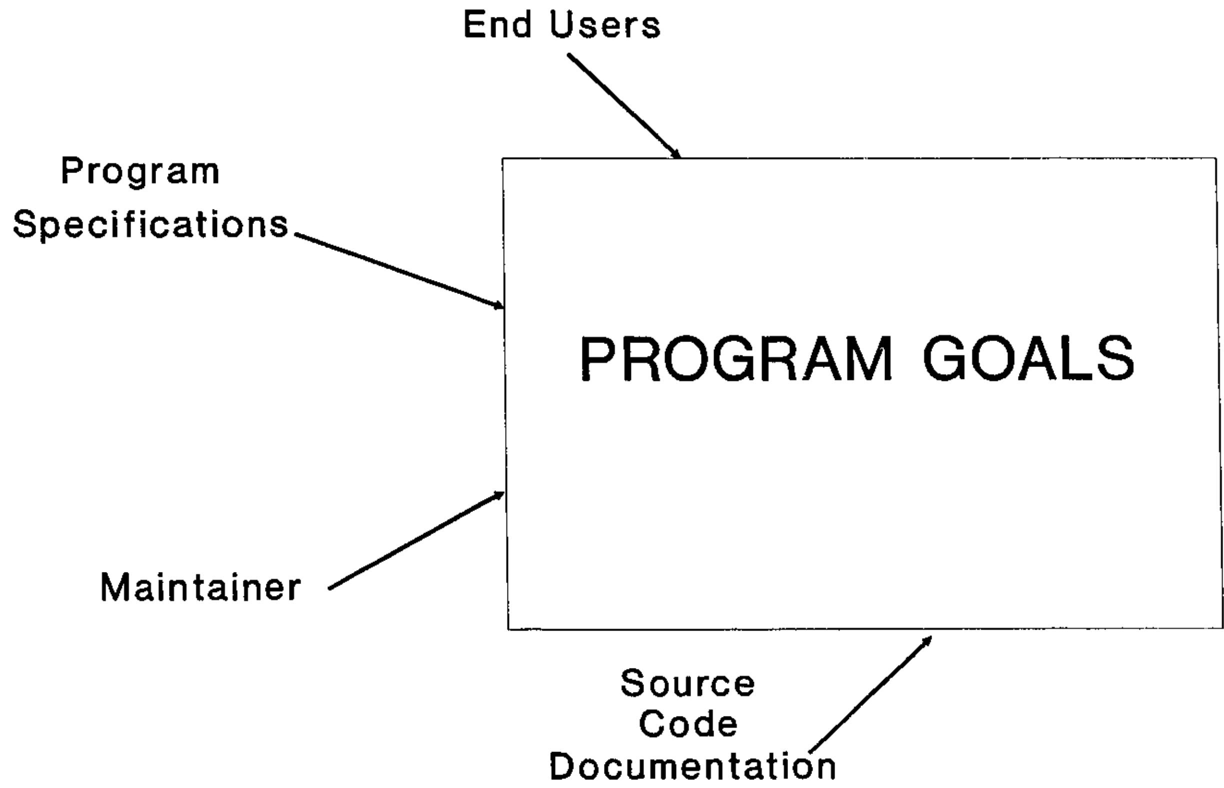


Figure 6.2: Domain Knowledge of KBPUTA

High Level Conceptual Knowledge Model

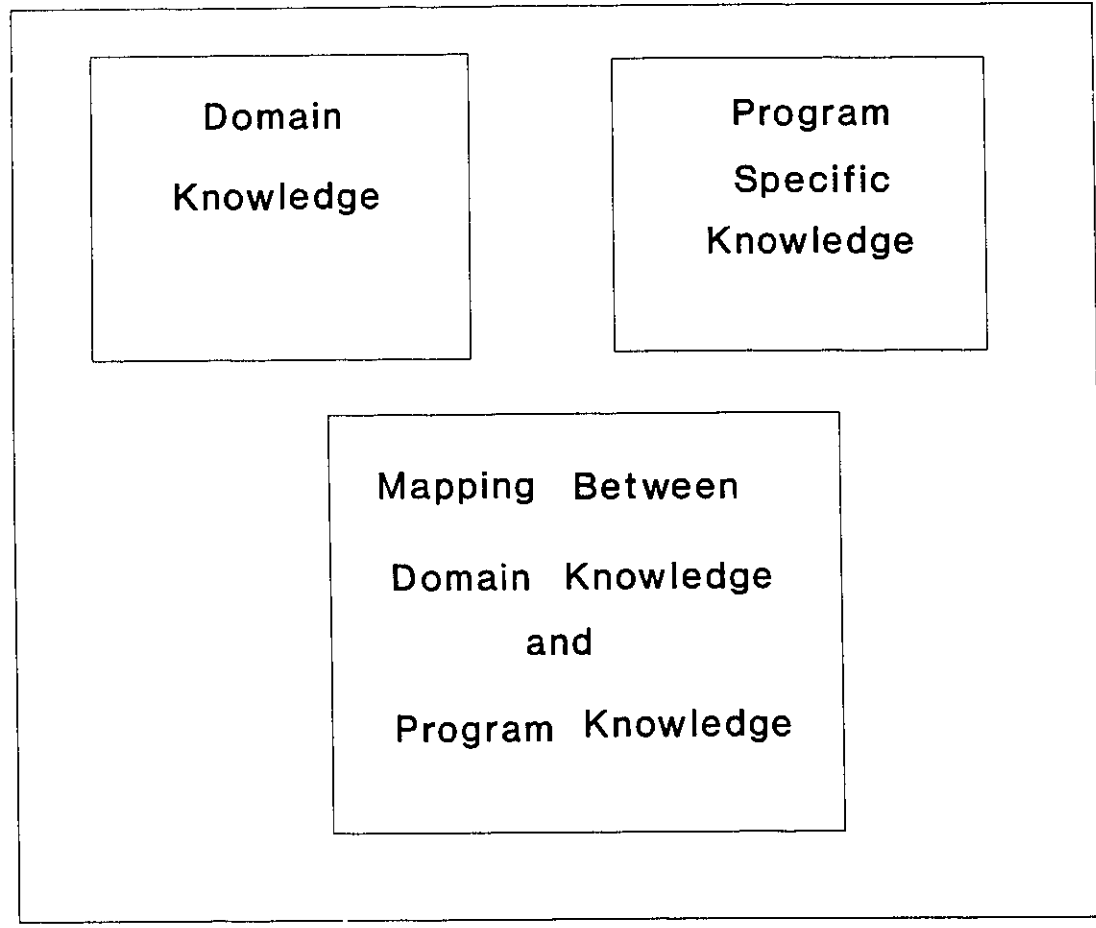


Figure 6.3 High Level Conceptual Knowledge Model of KBPUTA

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

As mentioned in Chapter 5, we have chosen to incorporate elements of the testing model of Leung and White, specifically the part which deals with global variables. This additional information should also increase understandability. We have only included the knowledge on the *regression testing* of global variables due to the limited scope of this proof-of-concept maintenance assistant.

6.2 THE MENTAL MODEL CREATED AND MAINTAINED BY THIS DESIGN

Viewing the cognitive model of program comprehension, as put forward by Letvosky (presented in 2.5.2), we can consider the assistant as an expansion of the knowledge base of the programmer. The degree of expansion depends on the maintainer's experience with the code. The assistant will enhance the knowledge of the maintainer to the degree necessary to provide a good understanding of the code. Using the work done by Littman et al (1986), it can be seen that this enhancement to the programmer's knowledge base should contain both static and casual knowledge in order that the mental model developed by the programmer will be a *strong one*. A maintainer, can start out with a program goal and review how it is implemented, or vice versa. The two directions are labelled ***program synthesis*** and *program understanding* respectively or *problem-driven or code-driven analysis* depending upon which terminology is used.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Once the maintainer has understood the program to their satisfaction, they will then try to construct a mental model of the modified program. To verify this mental model they can query the assistant as to what effect a change to a global variable, will have on the program. The assistant will, using the knowledge base and various rules that will be incorporated into it, let the maintainer know what other parts of the program are effected by this change. This may cause them to redevelop their original mental model because, perhaps, the program was not understood clearly. On the other hand, it may cause them to try and develop a new mental model (i.e. design a new modification) for the program. This process is illustrated in Figure 6.4.

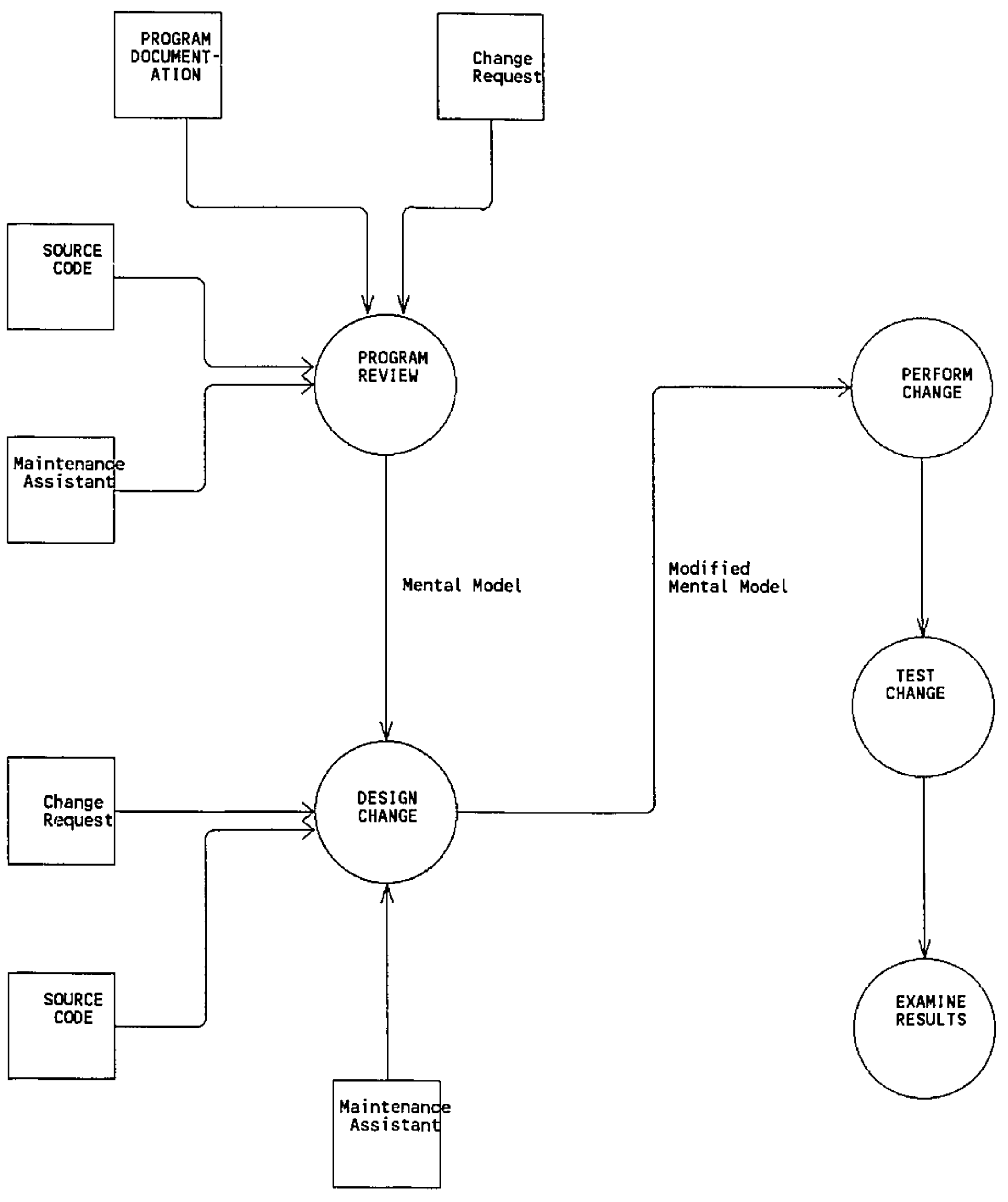


Figure 6.4: Mental Model Development Process

6.3 OVERVIEW OF FEATURES PROVIDED BY DESIGN OF THE ASSISTANT

The following is a summary of the features provided by this system design:

1) provides the user with a mapping between the domain knowledge and its implementation in the program through the use of the concept of a program plans and goals in order to aid in task of *program understanding*.

2) serves to control the need for *regression testing* by informing the user which modules should be tested in order to examine the impact on a global variable for a given change. The modification can then be designed so that it has the least possible impact on a global variable.

3) presents information on demand, thus allowing the user to pursue their own strategies for program comprehension.

   The user can choose to perform top-down (problem-driven) or bottom-up (code-driven) understanding or a combination of the two.

4) provides a form of change-consequence analysis.

   When trying to understand a program, the user constructs a mental model of the program and a mental model of the modification. By querying the system, this mental model is continually updated. For example, the programmer might have a model where the modification is designed around variable X, by querying the system he finds out that X is meant to be constant throughout the program, therefore he will change his mental model to incorporate another variable and the process is repeated.

5) provides a method for the maintainer to easily modify the knowledge base

6) the knowledge base is easily extended to include additional information such as the name of the module designer

6.4 OBJECTS AND ATTRIBUTES OF THE KBPUTA KNOWLEDGE BASE

A suitable knowledge formalism should have the following characteristics (Lucas 1991 and Van Der Gaag, pg 8):

1) sufficient power for encoding the particular domain knowledge

2) possess a clean semantic basis such that the meaning of the knowledge present in the knowledge base is easy to grasp

3) permit efficient algorithmic interpretation

4) allow for explanation and justification of the solutions obtained by the user by showing why certain questions were asked and certain conclusions drawn

Lucas and Van Der Gaag note that not a single knowledge representation formalism meets all of these requirements.

We have chosen to use objects to represent programming knowledge. As most of the knowledge-base development tools, (ADS, Nexpert, KBMS), incorporate objects into their knowledge representation schemes. Objects can easily encode our domain and program knowledge and present them to the maintainer. The following presents the types of knowledge to be stored and the attributes associated with each.

Name:	program
Slots:	name modules called global variables defined global variables used global constants defined global constants used input files output files user defined types goals of program subgoals of program programming plans used modules used but not called by main program

Name:	module
Slots:	name modules called modules called by local variables defined local variables used global variables defined global variables used local constants defined local constants used global constants defined global constants used parameters defined parameters used related goal programming plans used module type (function or procedure) function type

Name: Global Constant

Slots:

constant name
value
constant type
modules defined in
modules accessed in
program plans used in
description
purpose

Name: Global Variable

Slots:

variable name
variable type
modules define-use pairs
using modules
defining modules
program plans use in
killing modules
defining modules
transferring modules
description
purpose

Name: goal

Attributes:

Identification Phrase
description
Goal Type (Main Goal or Subgoal)
component program plans

Name: algorithm

Attributes:

inputs
outputs
purpose
description
components

Name: plan

Attributes:

Identification Phrase
description
name of related goal
purpose

Actual knowledge will be contained in instances of the above objects.

This design is for a program written in Pascal. Other types of languages would have different objects and attributes. It may be beneficial to add attributes not specified above. For example; design decisions, and bug history, could be added to program and module. Local variable and constant objects can also be included. The objects and attributes chosen depend on the needs of the maintainers as well as the programming language be used.

6.5 RULES FOR REGRESSION TESTING

ASSUMPTIONS:

A) For each global variable that is used in the given program, P,the use of each global variable is explicitly documented. The knowledge-based maintenance assistant contains "knowledge" on:

Define-use module pairs

Using modules

Defining modules

Killing Modules

Transferring Modules

B) For each module that is used in the given program, P, the knowledge-based maintenance assistant contains "knowledge" on:

Global variables defined in the module

Global variables used in the module

Each global variable which occurs in the module to be changed is examined separately for testing purposes.

C) Explanations will be provided for each of the questions and answers. For example, the maintainer can view a definition of corrective regression testing.

System Sourcing refers to automatic inferencing. User input is not required.

Each session with the knowledge-based maintenance assistant is called a consultation. We assume, for illustration purposes, that the global variable to be analyzed is in fact used or defined in the module being changed. For each consultation the maintainer should respond to the following questions:

1) Does the modification involve a change to the program specification.

Yes --> Progressive Regression Testing
Advise maintainer that new test cases may be required

No --> Corrective Regression Testing

2) What is the name of the module in which the change is to be made?

Information required for knowledge-base access.

3) What global variable is being analyzed for testing at this time (for a list of all global variables used in this module, see the main menu)?

Information required for knowledge-base access.

UNIT TESTING CONSULTATION

1) In the module where the modification is to be made, will the modified code be executed before global variable G is referenced?

YES -->	**Advise maintainer to Run Test Cases which traverse both the modified code and the instruction which references the global variable**
NO --->	**Advise maintainer that no unit regression testing is required**

System Sourcing

2) Access KB to determine if M is a defining module of G.

TRUE -->	**Produce listing of all def-use module pairs for global variable G** **Advise maintainer to repeat global variable tests and check whether previous values of G are replicated**
FALSE -->	**Determine use-def combinations** **Advise maintainer to run global variable tests to check if the output of the module indicates any error.**

In both cases, the maintainer should be advised that if the tests produce different results, then either the specification should be modified or the changes are incorrect.

TEST SELECTION STRATEGY FOR UNIT TESTING

Advise maintainer that the global variable can be treated as an extra input/output parameter to the module.

System Sourcing

Module M is a defining module of G

TRUE --> Advise maintainer:
- **i) To treat G as an output variable**
- **ii) That the test set should contain every functional condition of M under which G might change**
- **iii) That the test set should be drawn from functional and structural tests which affect G**

FALSE --> M is a using module of G
Advise maintainer:
- **i) the test sets much range over sufficient values of G so that all effects on B are represented by these tests.**
- **ii) External values of G should be used whenever possible**

INTEGRATION TESTING

System Sourcing

M is a defining module of G

TRUE -->	**Present to the maintainer a list of all using modules of global variable G** **Advise the maintainer that integration testing should be performed on all these modules.**
FALSE -->	**M is a using module of G** **Present the maintainer a list of all defining modules of global variable G** **Advise maintainer that integration testing should be performed on all these modules to ensure that the def-use assumption has not been violated.**

TEST SELECTION STRATEGY FOR INTEGRATION TESTING

Advise user that for each def-use pair or use-def pair, testing of G should be done after all the modules separating the pairs have been integrated together.

Functional tests may have to be sensitized if the pairs are on the same chain of calls or to a least common ancestor if they are on different chains of command.

Note: keywords sensitized and least common ancestor can be referenced by the maintainer.

CHAPTER 7.0

EVALUATION OF THE APPROACH AND THE TOOL

In this chapter we present an evaluation of KBPUTA. The first section specifies the type of program for which this proof-of-concept system is designed. The second section shows the program that was chosen to illustrate KBPUTA. The third section describes how KBPUTA was implemented in ADS and assessees ADS as the application platform. Finally, the fourth section comments on the usefulness of KBPUTA.

7.1 SPECIFICATION OF TARGET PROGRAM

The system is targeted toward conventional software for which there is little or no formal specifications. The domain specific knowledge, goals and plans, will be captured manually through direct input from the maintainer or the developer. The program source code knowledge can be either entered manually or through the use of an automated static analyzer. As was noted in Section 5.4, knowledge on killing, transferring and defining modules cannot be automatically provided.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

We have made it easy for the knowledge base to be updated by the maintainer or end user. No help from the original builders of the knowledge base should be required. When a modification to the knowledge base is required, a window for each attribute of the object being modified object in the knowledge base (module, goal, plan, global variable etc) will be displayed. The user is prompted to enter each object attribute.

The system is targeted towards structured programs. The target program should either be developed using the structured techniques or be structured using one of the available "structuring" tools. The type of program that our system will handle will have the following characteristics:

* composed of modules
* can include global variables
* can include global constants
* includes local variables
* includes local constants
* includes module parameters
* can include input and output files
* can include user defined type definitions

The following restriction on the proof-of-concept system should be noted:

* a module cannot be defined within another module
* module calls cannot be recursive

There is no fundamental reason why the design cannot handle these restrictions. However, in order to reduce the complexity of the proof-of-concept system, they were not included.

The following is an illustration of the type of program our assistant is targeted towards.

```
Program Alpha
Definition of Global Variables: Glob1, Glob2 ..type real
  Module Beta(Parameter P1)
        Definition Local Variables: Local1, Local2 .. type integer
     Lines of Code
 end Module Beta
Lines of Code
  Call to Module Beta(Parameter Glob.)
 end Program Alpha
```

7.2 EXAMPLE PROGRAM USED IN THE KBPUTA PROTOTYPE

We have chosen our example program from a book by Kernighan and Plauger, "Software Tools in Pascal" (page 113). We have placed the program in an application environment in order to generate domain knowledge. We have removed some of the parameters and have used global variables in their place in order to illustrate the regression testing assistance provided by KBPUTA. The scenario we have created is that this program is used by university professors to sort keywords and their attached definitions. If the keywords are the same, the sorting program examines the characters that appear in the definitions in order to determine which keyword should be placed first. This will serve as the domain knowledge for the example program.

The program performs a shell sort of strings. The strings are input from a file into a string array (linebuf). An ENDSTR character is appended to each string. Another array (linepos) is initialized to contain a set of indices which point to the first character of each string in the string array. The strings are read until either a string is too big, or there is not enough room left in the string array. The strings are then compared. They are not physically moved. Instead the indices in linepos are moved. The sorted strings are then outputted using the indices in linepos to indicate their order of output.

The modified example program is as follows:

```
Program Keyword_Sort(infile,outfile);

Const
        MAXCHARS = 10000; {maximum # of characters }
        MAXLINES = 300;   {maximum # of lines }

type
        charbuf = array [1..MAXCHARS] of character;
        charpos = 1..MAXCHARS;
        posbuf = array [1..MAXLINES] of charpos;
        pos = 0..MAXLINES;

var
        linebuf : charbuf;
        linepos : posbuf;
        nlines  : pos;

Function gtext (infile :filedesc) :boolean;

var
        i, len, nextpos : integer;
        temp : string:
        done :boolean;

begin
        nlines : =0;
        nextpos :=1;
        repeat
                done := (getline(temp, infile, MAXSTR) = false);
                if (not done) then begin
                        nlines := nlines + 1;
                        linepos[nlines] := nextpos;
                        len := length(temp);
                        for i := 1 to len do
                          linebuf[nextpos+i-1] :=temp[i];
          linebuf[nextpos+len] :=ENDSTR;
                        nextpos := nextpos + len + 1;
                end
        until (done) or (nextpos >= MAXCHARS-MAXSTR) or
```

```
                  (nlines >= MAXLINES);
      gtext := done
   end;


      Procedure ptext (outfile : filedesc);

      var
             i,j : integer;

      begin
             for i := 1 to nlines do begin
                    j :=linepos[i];
                    while (linebuf[j] <> ENDSTR) do begin
                           putcf(linebuf[j], outfile);
                           j :=j+1;
                    end
             end
      end;

      Procedure Shell (var linebuf : charbuf);

      var
             gap, i, j, jg, integer;

      begin
             gap := nlines div 2;
             while (gap > 0) do begin
               for i := gap+1 to nlines do begin
                    j := i-gap;
                    while (j > 0) do begin
                           jg := j + gap;
                           if (cmp(linepos[j],linepos[jg],linebuf) <= 0) then
                                  j :=0;
                           else
                                  exchange(linepos[j], linepos[jg]);
                           j := j-gap;
                    end
               end;
               gap : =gap div 2
             end
      end;
```

```
Procedure exchange (var lp1, lp2, :charpos);

var
        temp : charpos;

begin
        temp : = lp1;
        lp1 :=lp2;
end;

Function cmp (i,j:charpos;) : integer;

begin
        while (linebuf[i] = linebuf[j])
         and (linebuf[i] < > ENDSTR) do begin
          i : = i + 1;
          j : = j + 1;
        end;
        if (linebuf[i] = linebuf[j]) then
                cmp : =0;
        else if (linebuf[i] = ENDSTR) then
                cmp : = -1;
        else if (linebuf[j] = ENDSTR) then
                cmp : =+1;
        else if (linebuf[i] < linebuf[j]) then
                cmp := -1
        else
                cmp : = +1
end;


BEGIN
        if (gtext(linepos, nlines, linebuf, STDIN)) then begin
                shell(linepos, nlines, linebuf);
                ptext(linepos, nlines, linebuf, STDOUT)
            end
        else
                error('sort: input too big to sort')
        end;
```

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

We have chosen the following to be the main goal of the above program:

Main Goal: Produce a list of sorted keywords and definitions for instructional purposes.

Description: To sort a list of subject keywords and definitions into descending order.

The main purpose of the keyword sort program, in terms of the end-user, is captured in the main goal. The description provides further information on the goal to the maintainer.

We have decomposed the main program goal into subgoals. Each subgoal corresponds to a module in the program. There are only subgoals for three of the modules due to the fact that two of the modules, Exchange and Compare, were originally defined internally to the Shell module. They have been defined as separate modules in order to be compatible with our design of KBPUTA. The subgoals are as follows:

Subgoal: Output sorted keywords and definitions

Description: To output into a file a list of the subject keywords and their associated definitions

Subgoal: Input keywords and definitions

Description: Input keywords and their definitions from a file

Subgoal: Sort keywords and their definitions into ascending order

Description: Sort the keywords and their associated definitions into ascending order

Subgoals for such as small program are obviously not very complex. For a larger program they would much more detailed. In fact the subgoals for a larger program could in fact be decomposed into further subgoals.

Plans were developed which reflect stereotypical programming actions. Specific variable names are not mentioned in the plans in order to keep them generic. These plans are probably more detailed than is required. As mentioned previously, the amount of detail of a plan depends on the requirements of the maintainers and their environment. The plans that we chose for the above program are as follows:

Plan: Input strings into Array on Condition Loop

Description: Input a file of strings until each string until either the last string is read, or there is not enough array storage remaining, of the string is too large to be read. Each string will have the ENDSTR character appended to it.

Related Goal: Input Keywords and Definitions

Plan: Set up array index to string array

Description: Set up an index array in which each element contains a number corresponding to the start of each string in the array.

Related Goal: Input Keywords and Definitions

Plan: Output strings using Index Array

Description: Output strings from the string array. The order of their output is determined from the indices in the index array which point to the sorted entries.

Related Goal: Output sorted keywords and definitions

Plan: String Shell Sort Plan

Description: Shell sort of strings. Similar too bubble sort except that in the early stages, far-apart elements are compared, instead of adjacent ones.

Algorithm: Bubble Sort

Related Goal: Sort keywords and their definitions into descending order.

Plan: Exchange String Indices

Description: Exchange elements in the string array

Related Goal: Sort keywords and their definitions into descending order.

Plan: Compare Value of Strings

Description: Compare keywords in each string. A value of 0 denotes equal strings. A value of +1 indicates either string1 is > string2 or 2nd string is shorter in length. A value of -1 indicates either string1 < string2 or 1st string is shorter in length.

We have manually added these plans and goals to KBPUTA's knowledge base. We have also input program, module and global variable specific information. We should note that if a global variable is defined and used in the same module, the definition takes precedence over the use. That means that if the global variable is defined, then it will not show up as used in the on the module display.

We have used all of the objects and attributes in the design outlined in the previous chapter except for the global constant object. It was decided that it would be sufficient for the proof-of-concept system to only contain a global variable object. For the same reason we do not include a parameter object. An example session using this program is provided in Appendix C. This session shows the goals and plans of this program and module knowledge. It also illustrates the global variable regression testing assistance provided. It also presents an example of a modification to the knowledge base.

In capturing the knowledge, we determined that it would be an advantage in the future to add objects to the knowledge base to represent user defined types.

7.3 IMPLEMENTATION OF KBPUTA IN ADS

7.3.1 INTRODUCTION TO ADS AND REASONS FOR ITS CHOICE

The use of a knowledge-based development tool has meant that less attention needed to be paid to the implementational aspects of the maintenance assistant. These have nothing to do with the program domain being modeled. Aion's Application Development System/WIN (ADS/WIN), version 6.2, was chosen as the application development tool for KBPUTA. The PC development version of ADS/WIN sells for approximately $10.000. It was released in the Spring of 1993.

The Gartner Group, in a paper on ADS version 6.0, remarked that ADS was "a well-tested ES offering that had been on the market in one form or another for several years now." (Gartner Group, 1991A). At the time of the report, there were 575 customers and 400 applications in production. The report also says that one of the main strengths of ADS is in its knowledge representation. Gartner concluded that "Aion has pursued a strong, effective implementation of the object oriented paradigm and has done so quite effectively" (Gartner Group, 1991A).

The main reasons ADS/WIN was chosen were; we had already gained some experience with the tool, the effective implementation of the object-oriented paradigm suited our design, and ADS is a Microsoft Windows development platform that allowed us to build a Graphical User Interface (GUI) for our proof-of-concept system. Aion Corporation recently merged with AI Corporation. KBMS is AI Corporation's flagship product. The new corporate entity has been

named Trinzic Corporation.

Aion also has development platforms for the IBM System 370, the DEC VAX, and Sun-4 and Sparc stations. The knowledge bases and rules can be easily transported across platforms. However, the Windows Displays cannot be.

The ability to use Windows allows the maintainer to easily use the menuing system. Since Windows application development standards were followed, the user interface will be familiar to all Windows users.

7.3.2 ARCHITECTURE OF KBPUTA AS IMPLEMENTED IN ADS

Knowledge is represented in different ways in ADS. The following is taken from "ADS Product Training Part 1 Manual" (Trinzic Corporation 1991, pg 2-25 to 2-31).

> *Classes, slot, instances, and class hierarchies* are used in ADS to model "real world" things in the application area - for example, machines in a factory, automatic teller machine transactions or investment types.
>
> ADS *class hierarchies represent* the relationship between different levels of abstraction in the application area - for example, the relationship between investments in general and

specific kinds of investments like mutual funds or stocks.

Rules are used in ADS to represent judgmental knowledge about your application area - for example, how to evaluate the customer's credit status.
Rules enable an application to infer new knowledge from existing knowledge. You can use ADS rules to write forward chaining applications, backward chaining applications, or applications that combine forward and backward chaining.

States, State Agendas and Functions are used in ADS to exercise control over applications. *States* provide a framework for logical sub-groupings of rules and other kinds of knowledge. You use State Agendas to specify high-level control statements which direct the inference engine.

Functions define the steps for procedural tasks that are performed multiple times during a consultation: for example, determining the average and sum of a list of numbers.

Figure 7.1 illustrates the architecture of KBPUTA. Rules always belong to individual states in ADS. A prompt is what is displayed to the user when the inference engine is looking for a user sourced value.

ARCHITECTURAL DIAGRAM OF KBPUTA

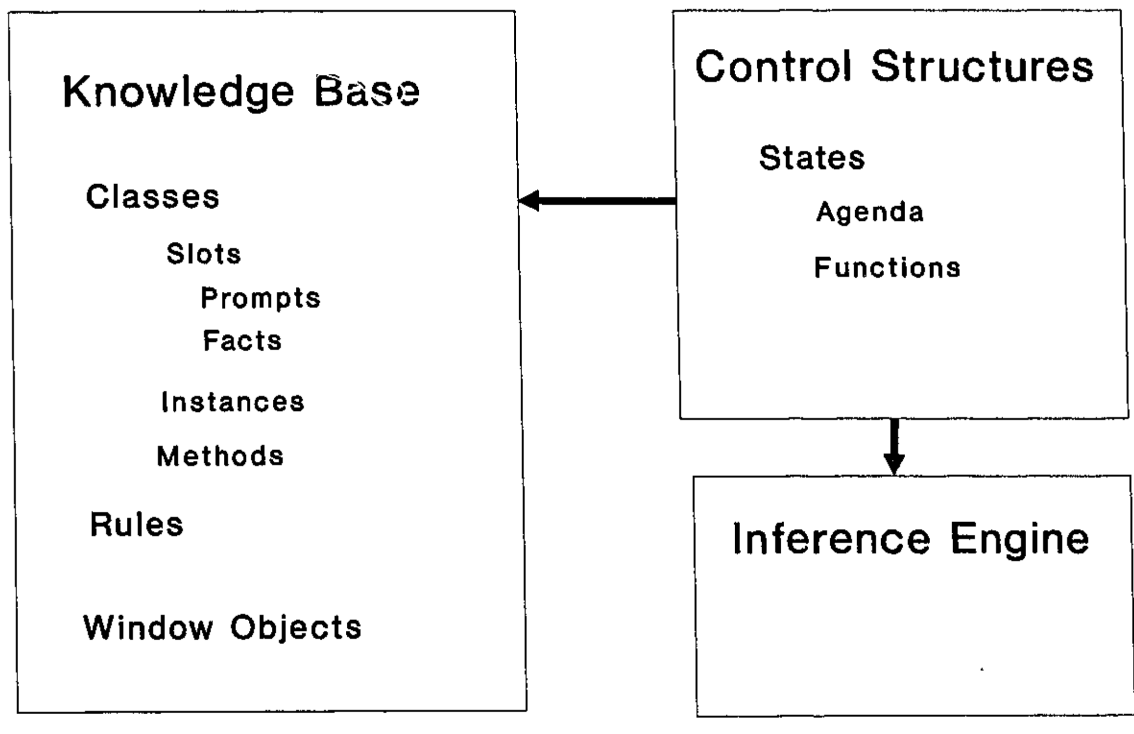


Figure 7.1: Architecture of KBPUTA

Figure 7.2 shows an example of a class in ADS. The module class is shown in this illustration. Clicking the appropriate buttons on the relevant Window Screens will cause the functions to be executed.

Figure 7.3 shows an example of a Rule in ADS. The name of the rule is the Unit_test_rule. It is part of the variable_anal state. This state is called when the user wants to obtain assistance on the *regression testing* of a global variable. When parameter Define is true it means that the module where the change is being made is a defining module of the global variable being analyzed. This rule appears in Section 6.5 under Unit Test Consultation. It is a system sourced rule. A class named modification is used to store all of the information related to a regression test of a global variable. For example, it stores the name of the global variable being analyzed and the name of the module where the modification is to be made. An instance of this modification is labelled mod1. The statement:

mods_to_be_tested = modification(mod1).glob_var_def_use

sets a parameter, mods_to_be_tested, equal to all the def-use module pairs for the global variable being analyzed. This information is then displayed to the user.

KBPUTA contains 5 rules in the above format. Another 5 rules are attached to displays so that the appropriate display is provided to the user depending upon what has been entered and the current content of the KB.

Figure 7.2: Example of an ADS Class

Figure 7.3: Example of an ADS Rule

7.3.3 ASSESSMENT OF ADS AS THE APPLICATION DEVELOPMENT PLATFORM

ADS/WIN is a complex system. Experience has shown that it takes approximately three months before a user feels comfortable with the system. Trinzic offers three four day courses in building knowledge-based systems and one course in using ADS/GUI. Knowledge of object-oriented concepts as well as experience in logic and writing rules is required.

However, the system is very robust. The ability to develop an application that is Windows based increases the usability of that application.

7.4 ASSESSMENT OF USEFULNESS AND APPLICABILITY OF KBPUTA

It is our position that KBPUTA is a very useful proof-of-concept system. The ability to provide three different views of a program will certainly increase *program understanding*. The easy way in which the knowledge is displayed will encourage users to use the system. The regression testing assistance provided will allow maintainer who are not familiar with the concepts behind *regression testing*, to easily follow a step by step plan.

The knowledge base has been designed to be easy to modify. It will grow as the maintainers' experience maintaining a piece of software grows. Its module oriented focus of having a subgoal

correspond to the function of a module will make it easy to be used by the people it is intended for, namely program maintainers.

The number of rules in the system for the *regression testing* of global variables will not change when the size of the program represented in the knowledge base increases.

However, we have not obtained any empirical results to back up our claim of the usefulness of the system. For example, as mentioned in Chapter 7.2, we determined that it would advantageous to add user-defined type objects. Further research is required to determine if the knowledge we are using is appropriate or if other types of knowledge are required.

The size of the run-time version of KBPUTA is 1.7 megs. For a larger program, it would be a good idea to store the knowledge in a database and then load it into the objects as required.

KBPUTA is applicable to all structured programming languages such as C, Fortran, PL1 and Pascal. It is designed to be used by maintainers who have no knowledge-based systems development experience.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Enhancements to KBPUTA would include;

* the addition of a help facility.
* less information being placed on each window.
* not having all the windows open at once.
* the ability to easily jump between related pieces of information. For example, when the user is viewing the information on a module, they should be able to click on the program plan implemented by this module and immediately jump to the screen containing information on this program plan.
* providing line numbers. For example, line numbers where a global variable is defined and used and where a module starts and ends.

In order to scale the assistant up to handle bigger programs, the windows would have to be changed in order to handle more information. For example, with a program with a large number of modules, if would be beneficial to have the user choose which way to modules are sorted on the screen. For example, they could be sorted alphabetically, by number of lines, or by the order in which the appear in the program.

CHAPTER 8.0

COMPARISON OF KBPUTA TO OTHER PROGRAM UNDERSTANDING SYSTEMS

In this section we present a comparison of KBPUTA with two other systems, PUDSY (Lukey 1980) and PROUST (Johnson & Soloway 1986, Johnson 1990). We also present different ways in which program knowledge has been represented.

8.1 PUDSY

PUDSY (Program Understanding and Debugging System) is a system that tries to perform program comprehension in order to identify bugs in a given program. PUDSY uses the code-driven approach.

PUDSY divides a program into chunks. A chunk is defined as a group of statements likely to form a useful unit of analysis. For example, PUDSY contains a heuristic that says that each loop should have its own chunk. It then determines what the input and output of each chunk are.

In order to determine what each chunk does, PUDSY tries to match chunks with chunks stored

in its knowledge base. If it cannot find a particular chunk in its knowledge base, then PUDSY symbolically executes the chunk.

PUDSY develops input and output assertions for each of the chunks by using the information stored in its knowledge base. These input and output assertions are then combined to derive an output assertion for the section of code under analysis. Then the input and output assertions can be compared to the original specification that can also be expressed in terms of output assertions.

PUDSY is an experimental system that has only been applied to small programs. PUDSY also does not contain domain specific knowledge and thus is of limited use as a human assistant.

8.2 PROUST

PROUST (Johnson & Soloway, 1985), is an example of a system that uses the problem-driven approach. This system recognizes plans in small pascal programs. The specification of goals and plans in PROUST was discussed in Section 4.3.

The input to the PROUST system consists of the program and a program specification that outlines the goals of the program and the objects that it must manipulate. The description is not an algorithm. It consists of a list of goals. The aim of the system is to find the most likely

mapping between the specifications and the code. This mapping is performed by searching for program plans.

PROUST selects one of the goals from the program specification. It then obtains from its knowledge base a list of plans that can implement this goal. Next, PROUST searches through the program to match the individual plans with the code. The process continues until all the goals have been mapped to sections of the source code.

Where there are no known plans for a goal, PROUST must be able to determine whether it is faced with an unknown implementation or whether it is faced with an incorrect or buggy section of code. Johnson and Soloway note that the more alternative plans there are, the harder it is to determine which plans the programmer was using.

In order to determine what situation PROUST is faced with it uses a knowledge base consisting of plan difference rules. The rules are triggered by differences between the plan being matched and the source code. The source code may be transformed or the differences may be explained as a bug.

The obvious problem with PROUST is that it can only identify plans that are stored in its knowledge base. At the time of writing of their article, the knowledge base contained about 35 goals, 55 plans, and 75 plan difference rules.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

This system, as is PUDSY, is an experimental system that has only been used on small programs.

A system similar to PROUST but which is trying to overcome the shortcomings of PROUST is under development at the Centre for Software Maintenance at the University of Durham (Calliss et al, 1988). Its knowledge base is to be based on the concept of program plans and is to contain information concerning how maintenance programmers perform their tasks. This knowledge will be used to form the system's heuristic knowledge that will dictate the weighing patterns on searches through this knowledge-based assistant. The knowledge base will also contain an internal representation of the program.

The program plans stored in the knowledge base will be divided into two categories: general program plans and program class knowledge. General program plans consists of a small number of plans that commonly occur in most programs. Program class plans are plans that are common to a particular type of program. For instance, a compiler written in C would have its own group of program class plans.

The system currently under development is interactive and will be able to provide the user with suggestions on where a modification should occur. It uses the concept of program transformations in order to prove that a program is equivalent to its specifications.

The system uses a small "kernal language" that consists of specification statements as the only

primitive statements. The kernal language also includes sequential composition, the if statement, a nondeterministic choice statement, a while loop, and simple recursive procedures. The language associates a function with each program through mathematical semantics. Two programs are said to be equivalent if their associated functions are identical.

Both of these plan based systems use a predefined knowledge base of program plans and goals. Unlike KBPUTA, no facility is in place to easily input plans and goals as well as other knowledge which we feel is important in any program understanding system.

8.3 LASSIE

LASSIE (Large Software System Information Environment) is a knowledge-based software information system (Devanbu et al, 1991). Its purpose is to help programmers in finding relevant information on a large software system. It was designed for the AT&T Definity 75/85. The system architecture consists of a large knowledge base, a semantic retrieval algorithm, and a user interface.

The knowledge base is an integration of architectural, conceptual, and code knowledge that they can be used by the programmer. It is a frame-based system. The basic premise of LASSIE is that it is important to provide programmers with multiple views of a programmer, from syntactic

knowledge to semantic knowledge. The syntactic information is at the level of a module call list. The programmer needs all these types of knowledge in order to fully understand a system. The knowledge base is produced through the process of reverse engineering. Reverse engineering involves the identification of the program requirements and design specifications behind a program. The information is manually loaded into the knowledge base. The authors feel that the cost of manually entering this information is more than offset by the benefits of building such a system.

LASSIE primarily processes queries concerning actions. Such as "What actions by the bus controller are caused by an action of an attendant?" The KB contains action descriptions written in the knowledge representation language, KANDOR. Currently, LASSIE does not contain any plan information. However, the intend to further develop LASSIE in order to provide this information. The authors note its importance in the understanding of programs.

This system is similar to ours in its focus on providing multiple views of a program. The difference lies in the focus of the work. The focus of LASSIE is on the knowledge base, while the focus of our work is on the user interface. We stress the involvement of the user. For example, LASSIE has no easy way of updating the knowledge base, while KBPUTA provides one. In the development of KBPUTA we took into consideration the needs of the user before building the knowledge base. We do not allow for specific Natural Language (NL) queries, we feel that the way the knowledge is presented eliminates the need for specific NL queries.

8.4 OTHER KNOWLEDGE REPRESENTATION SCHEMAS

When choosing how we were going to represent programming knowledge, we examined how this knowledge is stored in other software engineering knowledge-based systems. A few of the systems we examined are reviewed below.

A variety of systems are discussed in A Survey of Knowledge-Based Program Debugging Systems (Seviora 1987). The descriptions of the program vary in detail. Some systems construct a low-level representation of the program while others try to recreate the program's goal structure (PROUST). The more detailed the description, the easier the understanding process should be.

The first research project we examined was the CASE/MVS project (Symonds 1988). The basic object in the knowledge base for the CASE/MVS project is a framework consisting of a group of properties. A property is denoted by a symbolic identifier and a value that can either be natural language text or computer language text. Subproperties can also be created.

Various types of properties are used to represent the constructs required for the specification of the system. The project also allows property lists to be compiled into static models that are executable.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

The MARVEL system (Kaiser et al 1988) makes use of a different approach to programming knowledge. Under the MARVEL approach, facts are represented by objects and are stored in a database. These facts represent all the entities in the system as well as the relationship amongst the objects. Each object is an instance of a class that defines its type. The types of objects include module, procedure, type, design, description, user manual, and development.

Each class in the MARVEL system defines certain properties of objects and inherits other properties from its superclass or superclasses. One type of property called "attributes" defines the contents and status of objects.

Another system that we examined is Microscope (Ambras & O'Day 1988). Microscope is a system for helping programmers comprehend programs in order to perform modifications on them. Microscope's knowledge base is composed of frames and rules. Frames are used to represent data while rules are used to make inferences. The program source code is internally represented by frames. Cross-reference information is stored as a collection of attributes on variable, function , and module frames. One type of frame called annotations store documentation information.

CHAPTER 8.0

COMPARISON OF KBPUTA TO OTHER PROGRAM UNDERSTANDING SYSTEMS

This concluding chapter presents a summary of our results and a discussion of program understanding systems. It also contains a list of the unique characteristics of our research. The chapter concludes with a list of suggestions for future research.

9.1 SUMMARY OF RESULTS

We have successfully built a proof-of-concept knowledge-based maintenance and program understanding system. The methodology that was used can be used to develop a knowledge-based maintenance assistant using any object oriented knowledge-based system development tool. The knowledge base can easily be extended to handle other types of knowledge that are installation dependent. Knowledge is an expensive commodity, and minimizing the amount of knowledge required, the complexity of its representation, and its acquisition were priorities in the development of this system.

We have shown how regression testing assistance can easily be provided. This assistance will reduce the testing effort by providing relevant information to the maintainer before the modification is made and the tests are run.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

9.2 CURRENT VIEWS ON PROGRAM UNDERSTANDING SYSTEMS

At a recent panel on *program understanding* (Johnson 1992), the following points were presented:

Lewis Johnson (Developer of PROUST)

* systems are increasingly capable of recognizing high-level programming plans and domain concepts in programs

* program understanding tools must be designed to support the process of maintenance, ie program modification

* program understanding should focus on the following areas:
    - program understanding should be used to extract specific views of system
    - analysis should identify places where the view can be potentially affected by changes to other views

* programming plans must be represented so that program understanding can detect possible interactions between views.

Alex Quilici

* automated programming can contribute to solving the problem of aging software, but only as part of a complete environment designed to assist programmers in understanding software
* my research has shown that often the key to understanding programs is the domain dependent plans
* automated programmers will not be able to tackle realistic aging software systems in the near future. To do so would require vast number of domain-dependent plans.
* the process of program understanding should be a cooperative process between the system and the programmer
* a mechanism should be used so that programmers can record their understanding of the program

9.3 OUR VIEW OF PROGRAM UNDERSTANDING SYSTEMS

It is noteworthy that the direction of our research on *program understanding* was decided upon long before the above panel was held. Most of the points that came out of the panel have been addressed in our system.

The support for program modification has been addressed by the inclusion of regression testing assistance. The idea of providing different views of a program has been addressed by the use of a program goals view, a program plans view, and a module level view. The cooperative process between maintainer and the system has been addressed by building an assistant instead of a fully automated tool. The mechanism to record *program understanding* has been included in our assistant. The knowledge base in KBPUTA can easily be modified by the user.

We feel that the key to providing *program understanding* is the ability to view different types of knowledge about a program. The domain view, the plan view, and a module view are all extremely important in understanding a program. As well, the assistant should be very easy to use. Maintainers are often under a great amount of pressure and would not be interested in using a system that takes a long time to use. They want the knowledge in a timely way in order to understand the program in the least amount of time possible.

Although they might not need all of the knowledge we provide on the module windows, they

know that this is where they will find knowledge on a module, and not have to use a natural language query for a specific piece of knowledge.

9.4 UNIQUE CHARACTERISTICS OF OUR WORK

The following is a list of the unique characteristics of our work:

1) In KBPUTA, the program plans and their goals are input by the user. We did not build a system to automatically recognize program plans. In fact, researchers have stated that it is unlikely that plans can be identified automatically (Johnson, 1992). It will be necessary for the maintainer to provide some of the knowledge. The ability of the maintainers to input their own plans will allow more flexibility than in automatically capturing the plans in the code. To have the knowledge-base preloaded with plans and goals would create an enormous knowledge base. There would be a large number of domain-dependant goals and plans. This would be impractical. It will be up to the organization or the individual user of KBPUTA to decide exactly what they want to constitute a program plan. It will also allow the user to direct the building of their mental model. Not have it built for them.

2) We have made it easy of maintainer to use the assistant to record their understanding of a program. Plans and goals can be either modified or created.

3) By using the research done on the cognitive task of *program understanding*, we concentrated on the internal problem solving activity of the maintainer. Most current CASE tools concentrate on human computer issues of usability such as easy navigation and the direct manipulation of objects.

4) KBPUTA will present information on demand, thus allowing the user to pursue their own strategies for program comprehension (bottom up or top down). Different maintenance tasks require different strategies. It will also allow the knowledge contained in the system to be complemented by the knowledge of the maintainer.

5) Current systems are moving away from trying to replace experts to building systems for experts. Our system will act as an assistant to the maintainer, not as a replacement. A system designed to replace a maintainer would have to use such a large knowledge base that its implementation would be impractical.

6) Inclusion of regression testing assistance that will be used before the program is actually executed. This will reduce the testing effort by providing knowledge up front.

7) Inclusion of the *regression testing* work by Leung and White (1989, 1990, 1990A). We have built the first automated tool based around their regression testing model.

8) An increased concentration on domain knowledge in our definitions of program plans and goals.

9.5 SUGGESTIONS FOR FUTURE RESEARCH

In the future, we would like to obtain empirical results on the use of our maintenance assistant. Ramsey and Basili (1989) note that since research into software engineering is incomplete, the knowledge base of an expert system in this field will be exploratory and prototypical in nature.

As far as providing regression testing assistance is concerned, the system should also be extended to handle other types of testing besides that of global variable testing. Advice on parameter testing, for example, should be provided. We would also like to develop a formula for comparing the cost of tests. The assistant could then advise the maintainer which modification has the lowest testing cost.

With the increasing popularity of object-oriented programming, it would be worthwhile to modify the maintenance assistant to handle the paradigm. We predict that this modification will be fairly easy.

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

As mentioned in Section 5.4, a tool that will parses a program and automatically provide information on killing, transferring, and defining modules is not available. This tool is required in order to free the maintainer from recording this information manually.

KBPUTA was built as a proof-of-concept system. The user interface requires numerous enhancements in order to make it easier to use. These enhancements include; less data displayed on the windows, addition of a help facility, not having all the windows open at once, and the ability to jump between related pieces of information by simply clicking on the item with a mouse.

Other enhancements to KBPUTA to address the scalability issue would include, the addition of line numbers where each global variable or module is found, and the addition of a facility to sort the names of modules in various orders.

It would also be beneficial to build KBPUTA on a non-proprietary platform. This would make the assistant more portable.

KBPUTA was designed with the purpose of solving a problem which affects all data processing organizations. With additional work on KBPUTA the system will evolve from a proof-of-concept system to a production system.

APPENDIX A: BIBLIOGRAPHY

Adelson, B. (1981)
Problem Solving and the Development of Abstract Categories in Programming Languages
in Memory and Cognition, April 1981, pp 422-433

Ambros J., O'Day V. (1988)
Microscope: A Knowledge-Based Programming Environment
IEEE Software, May 1988, pp 40-49

Basili V.R., Mills H.D. (1982)
Understanding and Documenting Programs
IEEE Trans. on Soft. Eng. SE-8, pp 270-283

Basili V.R, Musa John D. (1991)
The Future Engineering of Software: A Management Perspective
IEEE Computer, Sept 1991, pp 90-96

Brooks R. (1983)
Towards a Theory of the Comprehension of Computer Programs
International Journal of Man-Machine Studies 18, pp 543-554

Caliss F.W., Khalil M., Munro M,. Ward M. (1988)
A Knowledge-Based System for Software Maintenance
in Proceedings of Conference on Software Maintenance-1988
Phoenix, Arizona, October 24-27, 1988
IEEE Computer Society Press, pp 319-323

Cleveland L. (1988)
An Environment For Understanding Programs
in Proceedings of the 21st Hawaii International Conference on
System Sciences, pp 500-509

Curtis B., Sheppard S.B. (1989)
Experimental Evaluation of Software Documentation Formats
Journal of Systems and Software 9, pp 167-207

Devanbu Premkumar, Brachman Ronald J. (1991)
LaSSIE: A Knowledge-Based Software Information System
Comm. of the ACM, Vol 34, No. 5, May 1991, pp 34-49

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Fairley R. (1985)
Software Engineering Concepts
McGraw Hill Book Company, New York

Feigenbaum, E (1983)
Knowledge Engineering: The Applied Side of AI
in Annals of the New York Academy of Sciences 426, pp 91-107

Fischer Gerhard, Girgensohn Andreas, Nakakoji Kumiyo, Redmiles David (1992)
Supporting Software Designers with Integrated Domain-Oriented Design Environments
IEEE Trans. Software Engineering, Vol 15, June 1992, pp 511-522

Fjeldstrad R.K., Hamlen W.T. (1983)
Application Program Maintenance Study - A Report To Our Respondents
in G. Parikh and N. Zvegintzov (Eds), Tutorial on Software Maintenance, IEEE

Freedman D.P., Weinberg G.M. (1980)
A Checklist For Potential Side Effects of a Maintenance Change
in Techniques of Program and System Maintenance, edited by Girish Parikh, Ethotech Inc, pp 61-68.

Gartner Group Inc. (1991)
Software Engineering Strategies: Selecting Regression Testing Technologies
Gartner Group Inc, Stamford CT, File Technology T-222-554, May 15, 1991

Gartner Group Inc. (1991A)
ADS Has Become a Formidable Development Tool
Gartner Group Inc, Stamford CT, File Products P-860-527, March 8, 1991

Gibson Virginia A, Senn James A. (1989)
System Structure and Software Maintenance Performance
in Communications of the ACM, Vol 32, No. 3. March 1989, pp 347-356.

Glass Robert L. (1992)
Editors Corner: We Have Lost Our Way
in Journal of Systems Software, 1992 Vol 18, pp 11-112

Harandi M.J., Ning J.Q. (1988)
PAT - A Knowledge-Based Program Analysis Tool
in Proceedings of Conference on Software Maintenance-1988
Phoenix, Arizona October 24-27, 1988
IEEE Computer Science Press, pp 312-318

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Harrold M.J, Sofa M.L. (1989)
Intraprocedural Data Flow Testing
in Proc. 3rd Symposium on Software Testing Analysis and Verification, pp 158-167

Hartman Jean, Robson David J. (1990)
Techniques for Software Revalidation
in IEEE Software, Jan 1990, pp 31-36

Horowitz S., Prins J., Reps T (1988)
Integrating Non-Interfering Versions of Programs
in Proc. of SIGPLAN 88, Jan. 1988

Johnson W.L., Soloway E. (1985)
PROUST: Knowledge-Based Program Understanding
in C. Rich and R. Waters (Eds), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufman Publishers Inc., Los Altos CA., pp 443-451

Johnson, W. Lewis (1990)
Understanding and Debugging Novice Programs
in William J. Clancey and Elliot Soloway (Eds), Artificial Intelligence and Learning Environments, MIT/Elsevier, MA., pp 51-97

Johnson W.L., Ning Jim Q, Quilici Alex, Devanbu Pram (1992)
Panel On: Program Understanding - Does It Offer Hope for Aging Software?
in Proc. of the 7th Knowledge-Based Software Engineering Conference, Sept 20-23, 1992 IEEE Computer Society Press

Kaiser G.E., Feiler P.H., Popousk S. (1988)
Intelligent Assistance for Software Development and Maintenance
IEEE Software, May 1988, pp 40-49

Kernighan Brian W., Plauger P.J. (1981)
Software Tools in Pascal
Addison-Wesley Pub. Company, Reading MA.

Kozacynski Wojtek, Letvosky Stanley, Ning Jim (1991)
A Knowledge-Based Approach to Software Understanding
in Proc. Of 6th Annual Knowledge-Based Software Engineering Conference, Sept 1991, sponsored by Rome Air Force Base, pp 203-214

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Letvosky, S. (1986)
Cognitive Processes in Program Comprehension
in E. Soloway and S. Iyenger (Eds), Empirical Studies of Programmers, Ablex Pub, Norwood, N.J., pp 58-79

Letvosky, S., Soloway E. (1986)
Delocalized Plans and Program Comprehension
IEEE Software, May 1986, pp 41-48

Leung Hareton K.N., White Lee (1989)
Insights into Regression Testing
in Proc. of 1989 Conference on Software Maintenance

Leung Hareton K.N., White Lee (1990)
A Study of Integration Testing and Software Regression at the Integration Level
1990 Conference on Software Maintenance, pp 290-301

Leung Hareton K.N., White Lee (1990A)
Insights into Testing and Regression Testing of Global Variables
in Software Maintenance: Research and Practice, Vol 2,pg 209-222

Lientz B.P. and Swanson (1980)
Software Maintenance Management
Addison-Wesley, New York

Lientz B.P., Swanson B.E. (1981)
Problems in Application Software Maintenance
Communications of the ACM, Nov. 1981, pp 763-769

Littman D.C., Pinto J., Letvosky S., Soloway E. (1986)
Mental Models and Software Maintenance
in E. Soloway and S. Iyenger (Eds), Empirical Studies of Programmers, Ablex Pub, Norwood, N.J., pp 80-98

Lucas Peter, Van der Gaag Linda (1991)
Principles of Expert Systems
Addison-Wesley Pub Company, 1991

Luckey, F.J. (1980)
Understanding and Debugging Programs
in International Journal of Man-Machine Studies, Feb 1980, pp 189-202

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

McGraw Karen L., Harbison-Briggs Karan (1989)
Knowledge Acquisition: Principles and Guidelines
Prentice Hall, Englewood Cliffs, New Jersey

Myers G.J. (1979)
The Art of Software Testing
Wiley, New York 1979, pg 122

National Bureau of Standards (1985)
NBS Special Publication 500-135
in Software Maintenance and Computers, edited by David H. Longstreet, IEEE Computer Press Tutorial, Los Almitos, California 1990, pp 2-13.

Osterweil Leon J., Taylor Richard N (1978)
A Facility For Verification, Testing and Documentation of Concurrent Process Software
1978 IEEE Order # Ch1338-3/78 pp 6-41

Pennington N (1987)
Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs
in Cognitive Psychology, Vol 19, 1987, pp 295-341

Parikh G., Zvegintzov N.
The World of Software Maintenance
from Tutorial on Software Maintenance, G. Parikh and N. Zvgintzov eds, CS Press, Los Alamitos Calif, 1988, pp 1-3

Ramesy Connie, Basili Victor (1989)
An Evaluation of Expert Systems for Software Engineering Management
in IEEE Trans. on Soft. Eng., Vol 15, No.6, May/June 1989, pp 14-15.

Rich C. (1981)
Inspection Methods in Programming
Technical Report AI-TR-604, MIT LAB 1981

Rich Charles, Waters, Richard C (1986) (editors)
Readings In Artificial Intelligence and Software Engineering
Morgan Kaufman Publishers Inc., 1986

Robson D.J., Bennett K.H., Cornelius B.J., Munro.M (1991)
Approaches to Program Comprehension
Journal of Systems Software 1991 14, pp 79-84

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Schank Roger C. (1982)
Reading and Understanding: Teaching from the Perspective of Artificial Intelligence
Lawrence Erlbaum Associates, Publishers Hillsdale, N.Y. 1982

Schank Roger C. (1984)
The Cognitive Computer
Addison-Wesley Pub. Co., Reading Mass.

Schneiderman N., Mayer R. (1979)
Syntactic/Semantic Interactions in Program Behaviour : A Model and Experimental Results
International Journal of Computer and Information Sciences, pp 219-238

Schneiderman N., Mayer R. (1979)
Syntactic/Semantic Interactions in Program Behaviour : A Model and Experimental Results
International Journal of Computer and Information Sciences, pp 219-238

Seviora R.E. (1988)
Knowledge-Based Program Debugging Systems
IEEE Software, May 1988, pp 20-31

Soloway E., Ehrlich K. (1984)
Empirical Studies of Programming Knowledge
IEEE Trans. on Software Engineering, Sept. 1984 pp 595-609.

Soloway E (1986)
Learning to Program = Learning to Construct Mechanisms and Explanations
in Comm. of the ACM, Sept. 1986, Vol 29, No.9

Sowa John F. (1984)
Conceptual Structures: Information Processing in Mind and Machine
Addison-Wesley, Reading Mass, 1984

Symonds, A.J. (1988)
Creating a Software-Engineering Knowledge Base
IEEE Software, March 1988, pp 50-57

Trinzic Corporation (1991)
ADS Product Training 1 (ADS 6.00)
Trinzic Corporation, 101 University Avenue, Palo Alto, California

KNOWLEDGE-BASED PROGRAM UNDERSTANDING AND TESTING ASSISTANT

Waters Richard C. (1982)
The Programmer's Apprentice: Knowledge Based Program Editing
in IEEE Trans. on Software Eng, Vol SE-8, Jan. 1982, pp 1-12

Waters Richard C. (1985)
The Programmer's Apprentice: A Session with KBEmacs
IEEE Trans. on Software Engineering, Vol SE-11, Nov. 1985, pp 1296 -1301

Weiser M. (1982)
Programmers Use Slices When Debugging
in Comm. of the ACM, July 1982

White Douglass (1991)
The Knowledge-Based Software Assistant: A Program Summary
in Proc. of the 6th Conf. on Knowledge-Based Software Engineering,
Sept. 1991, Rome Laboratory, pp vi-xiii

Wilde Norman, Thebaut, Stephen (1989)
The Maintenance Assistant: Work in Progress
in the Journal of Systems and Software 9, January 1989, pp 3-16.

APPENDIX B: KBPUTA SAMPLE SESSION

We present a sample session of KBPUTA in this Appendix. All of the options available will be shown.

We provide a diagram of the KBPUTA Menu organization. This is used to orient the reader as the screens are shown. The sample session begins with the Welcome Screen on page B-4. The Main Menu is then displayed on B-5. All of the options on the will be executed. The first option on the Main Menu is Information on Program. The Program Menu is shown on page B-6. All of the options on the Program Menu will be executed. Once the last screen of an option is reached, the screens are then backed out of by choosing the appropriate button. The Program Menu options consist of :

* Information on a Module (B-7)
The user can choose which module to view (B-8)
Domain Knowledge Display (B-9)
Static Information Display Page 1(B-10)
Static Information Display Page 2 (B-11)

* Information on Entire Program (B-12)
Domain Knowledge Display (B-13)
Static Information Display Page 2 (B-14)

* Information on a Global Variable (B-15)
The user can choose which Global Variable to view (B-16)
Knowledge Display (B-17)

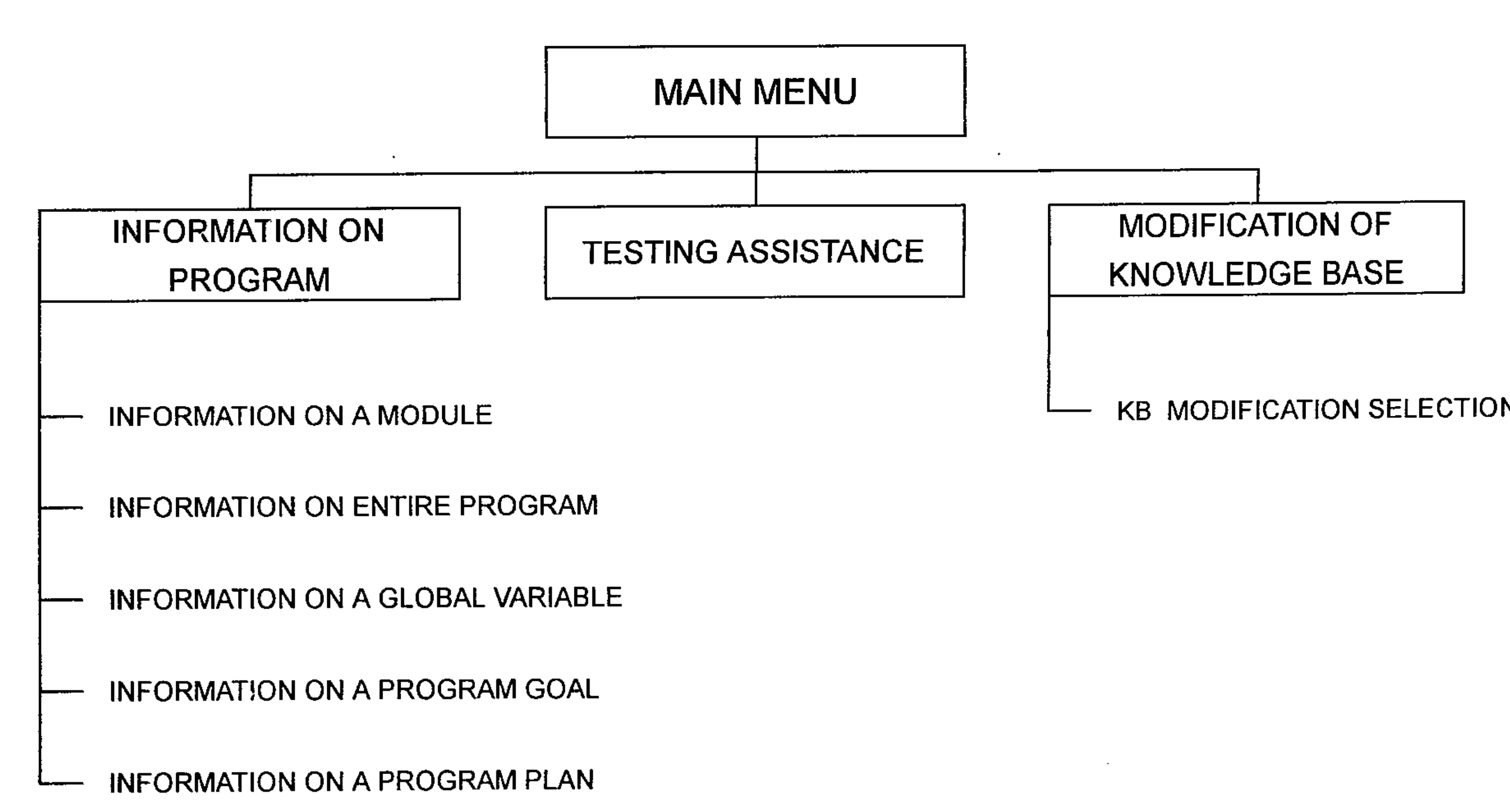
* Information on a Program Goal (B-18)
The user can choose which Goal to view (B-19)
Subgoals are denoted by a -
Goal Display (B-20)

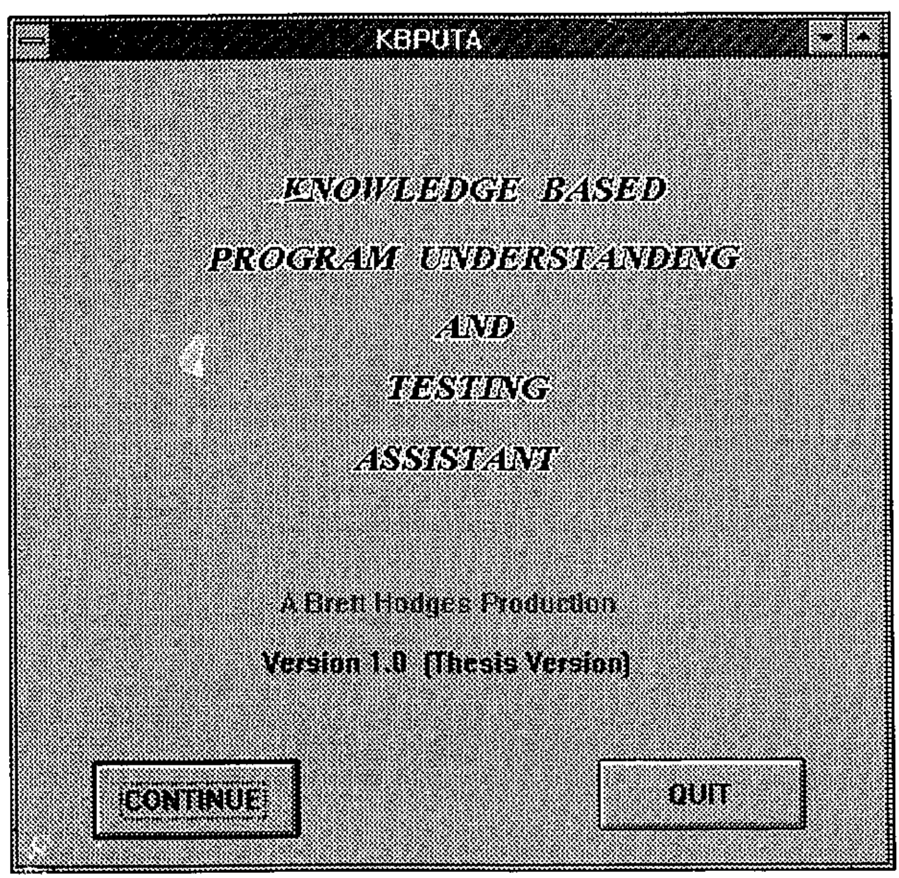
* Information on a Program Plan (B-21)
  The user can choose which Plan to view (B-22)
  Plan Display (B-23)

The Testing Assistance Option is then executed from the Program Menu (B-24). The user responds to the question "Does the modification involve a change to the program specification?". The Default answer is Yes (B-25). The system then informs the user of the type of regression testing required on page B-26. On page B-27, the user is asked for the name of the module in which the modification is to be made. Module Gtext is the default. On the next page, the user is asked to choose which global variable is to be analyzed. The user is then asked on page B-28 whether the modification is to be executed before or after the global variable being analyzed is referenced. The next five pages provide the user with information on how to perform the regression testing. The def-use module pairs (B-30) and the using modules (B-33) displayed are automatically pulled from the knowledge base.

The final option is then chosen on page B-34. The next page shows the screen for choosing which object to modify. The user has chosen to create a program goal. The system then automatically prompts the user for all of the attributes stored in the knowledge base for the goal object.

KBPUTA MENU ORGANIZATION




KBPUTA
KNOWLEDGE BASED
PROGRAM UNDERSTANDING
AND
TESTING
ASSISTANT
A Brett Hodges Production
Version 1.0 (Thesis Version)
CONTINUE
QUIT

MAIN MENU

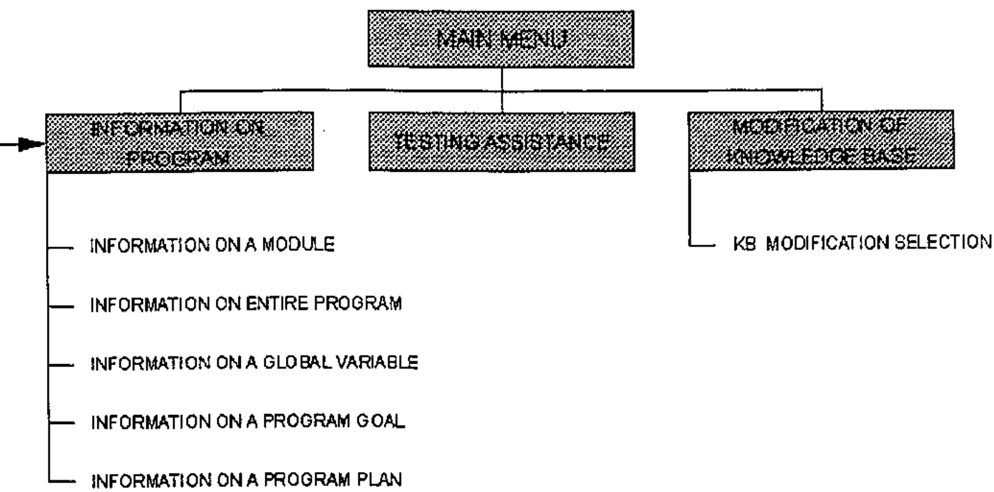
OPTIONS

- Information on Program
- Testing Assistance
- Modification Of Knowledge Base

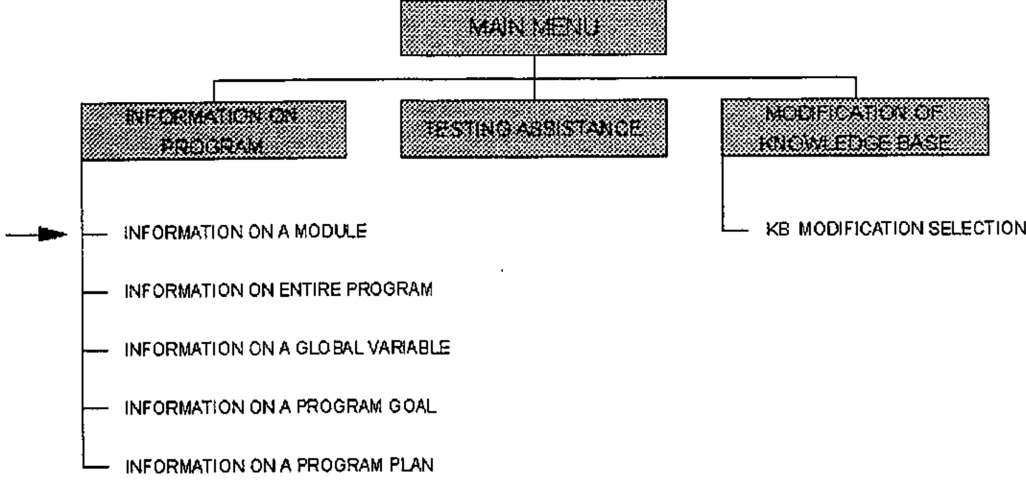
OK

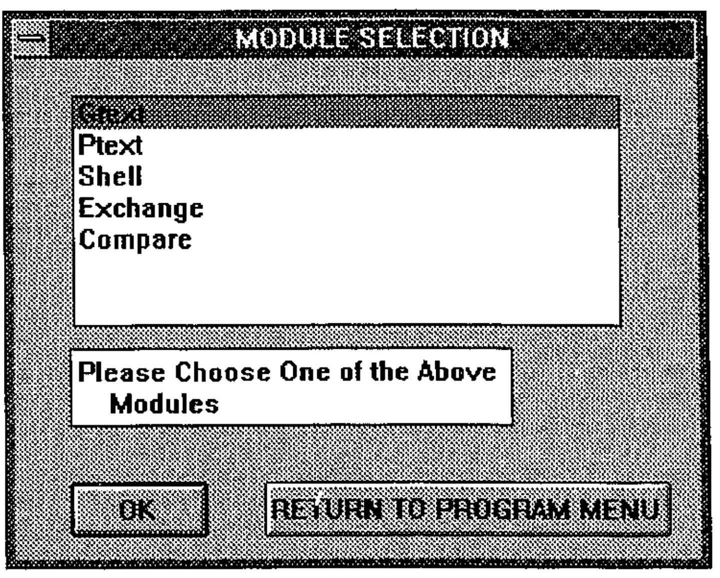
EXIT APPLICATION

KBPUTA MENU ORGANIZATION



KBPUTA MENU ORGANIZATION




MODULE SELECTION
Gtext
Ptext
Shell
Exchange
Compare
Please Choose One of the Above Modules
OK
RETURN TO PROGRAM MENU

MODULE DOMAIN KNOWLEDGE DISPLAY

Module Name: Gtext

Procedure or Function: function

Function Type (If Applicable): boolean

Related Goal: Input Keywords and Definitions

Plans:
Input Strings into Array on Condition Loop
Set up Array Index To Store Array

STATIC MODULE INFORMATION

RETURN TO MODULE CHOICES

MODULE STATIC INFORMATION DISPLAY Page 1

Modules Called

Modules Called By: Keyword_Sort

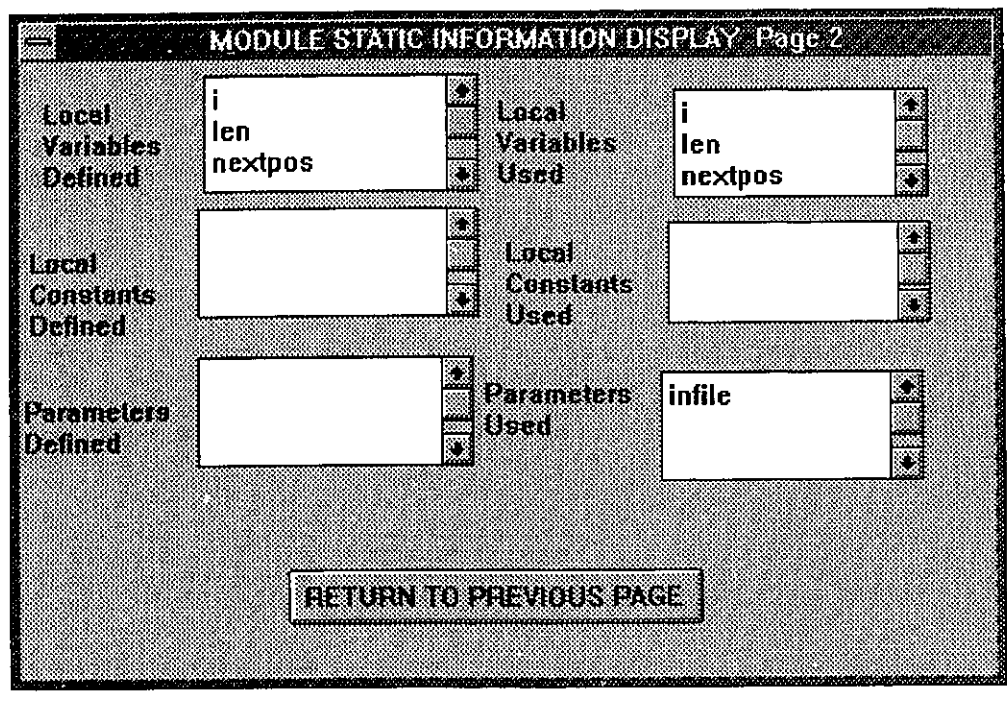
Global Variables Defined: nlines, linepos

Global Variables Used

Global Constants Used

NEXT PAGE

RETURN TO MODULE DOMAIN INFO


MODULE STATIC INFORMATION DISPLAY Page 2
Local Variables Defined
i
len
nextpos
Local Variables Used
i
len
nextpos
Local Constants Defined
Local Constants Used
Parameters Defined
Parameters Used
infile
RETURN TO PREVIOUS PAGE

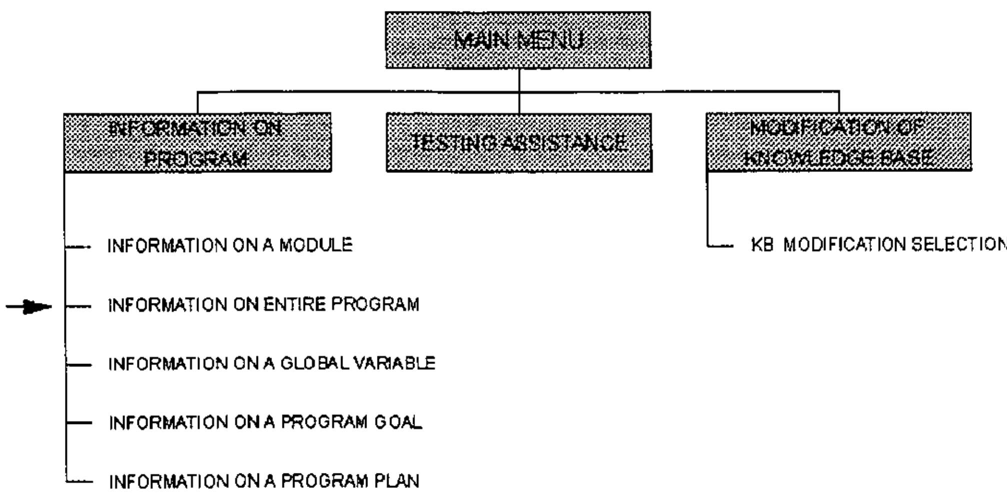
PROGRAM MENU

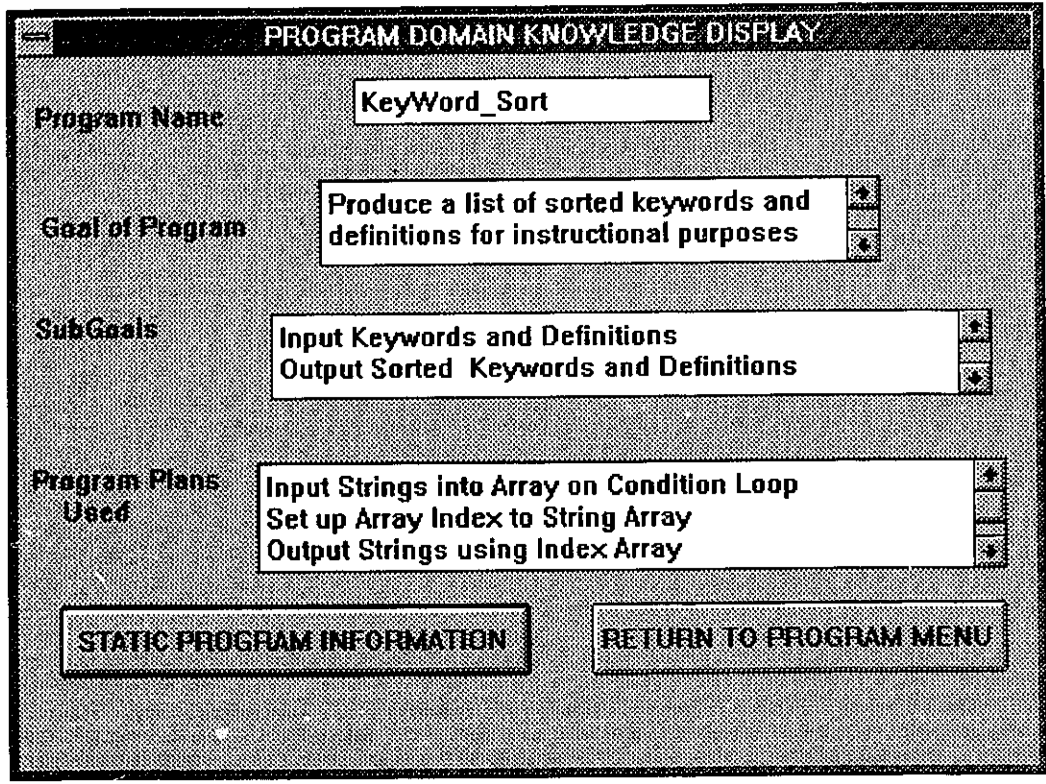
PROGRAM VIEWING OPTIONS

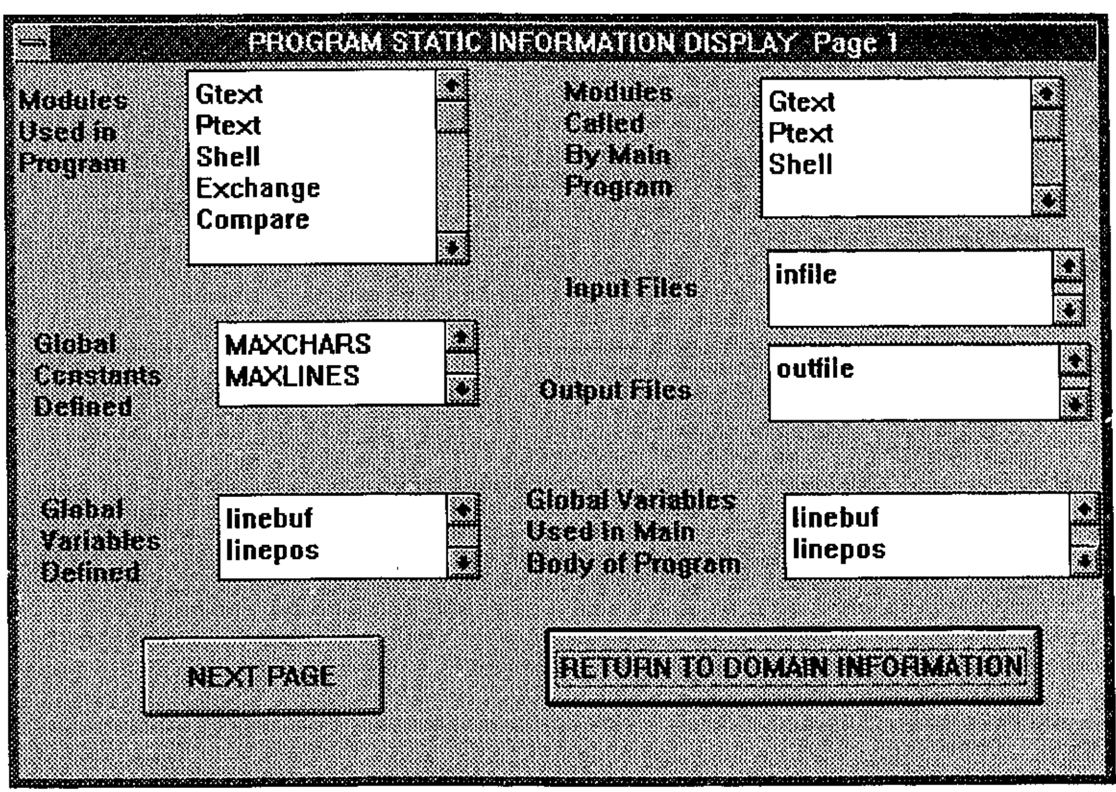
- ○ Information on a Module
- ● Information on Entire Program
- ○ Information on a Global Variable
- ○ Information On a Program Goal
- ○ Information On A Program Plan

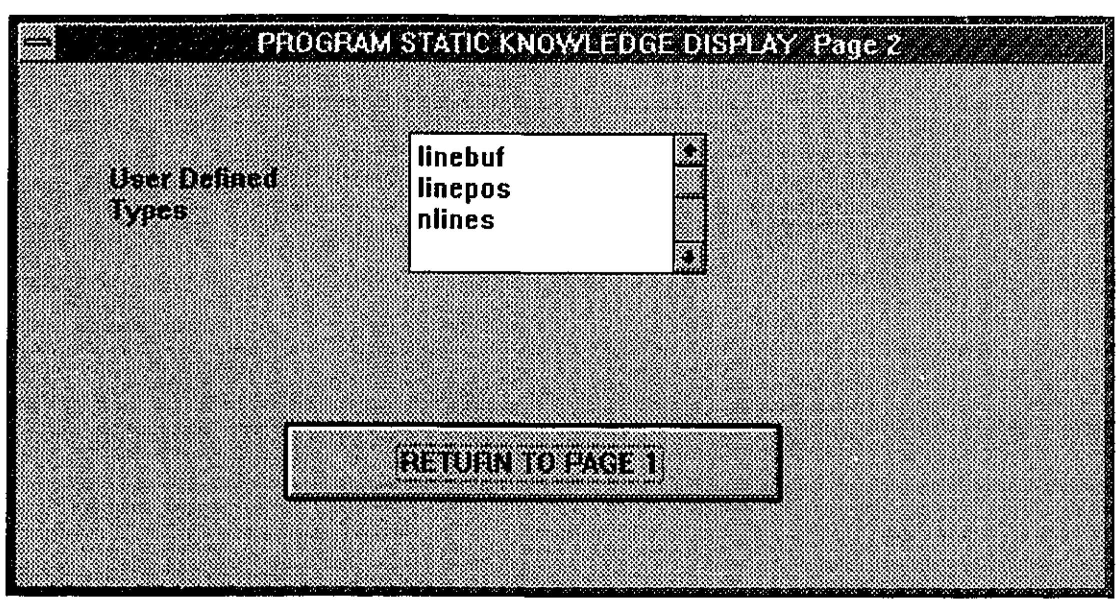
[OK] [RETURN TO MAIN MENU]

KBPUTA MENU ORGANIZATION




PROGRAM DOMAIN KNOWLEDGE DISPLAY
Program Name
KeyWord_Sort
Goal of Program
Produce a list of sorted keywords and
definitions for instructional purposes
SubGoals
Input Keywords and Definitions
Output Sorted Keywords and Definitions
Program Plans
Used
Input Strings into Array on Condition Loop
Set up Array Index to String Array
Output Strings using Index Array
STATIC PROGRAM INFORMATION
RETURN TO PROGRAM MENU


PROGRAM STATIC INFORMATION DISPLAY Page 1
Modules Used in Program
Gtext
Ptext
Shell
Exchange
Compare
Modules Called By Main Program
Gtext
Ptext
Shell
Input Files
infile
Global Constants Defined
MAXCHARS
MAXLINES
Output Files
outfile
Global Variables Defined
linebuf
linepos
Global Variables Used in Main Body of Program
linebuf
linepos
NEXT PAGE
RETURN TO DOMAIN INFORMATION


PROGRAM STATIC KNOWLEDGE DISPLAY Page 2
User Defined
Types
linebuf
linepos
nlines
RETURN TO PAGE 1

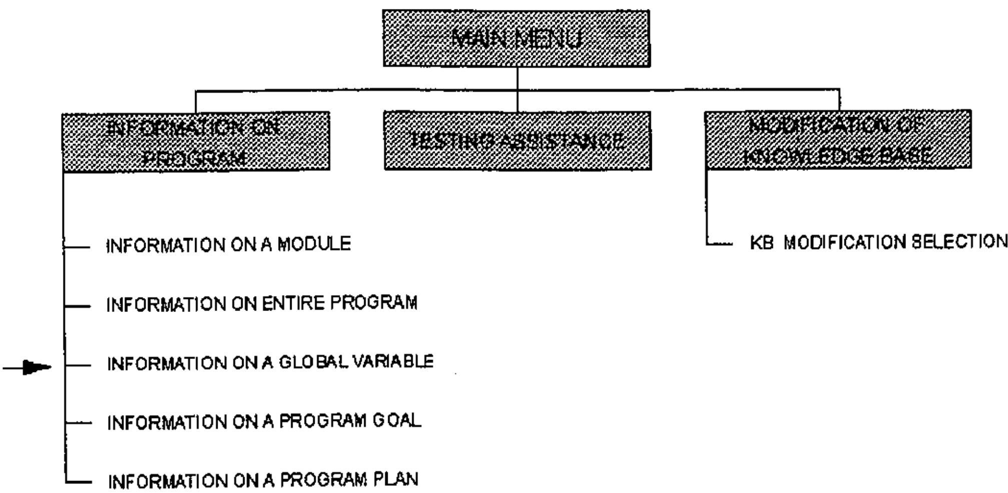
PROGRAM MENU

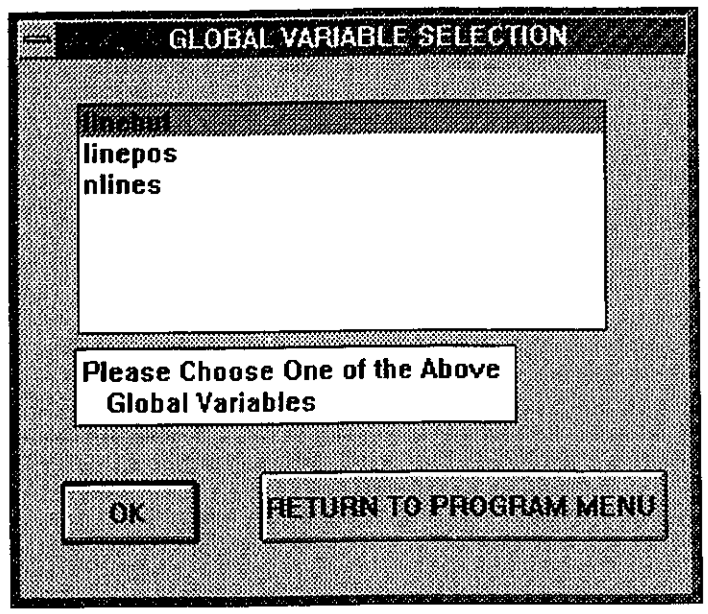
PROGRAM VIEWING OPTIONS

- ○ Information on a Module
- ○ Information on Entire Program
- ◉ Information on a Global Variable
- ○ Information On a Program Goal
- ○ Information On A Program Plan

OK | RETURN TO MAIN MENU

KBPUTA MENU ORGANIZATION




GLOBAL VARIABLE SELECTION
linepos
nlines
Please Choose One of the Above
Global Variables
OK
RETURN TO PROGRAM MENU

GLOBAL VARIABLE KNOWLEDGE DISPLAY

Name: linebuf

Type: charbuf

Define-Use Module Pairs: gtext ptext / gtext cmp

Using Modules: cmp

Defining Modules: gtext

Killing Modules:

Transferring Modules: shell

RETURN TO GLOBAL VARIABLE CHOICES

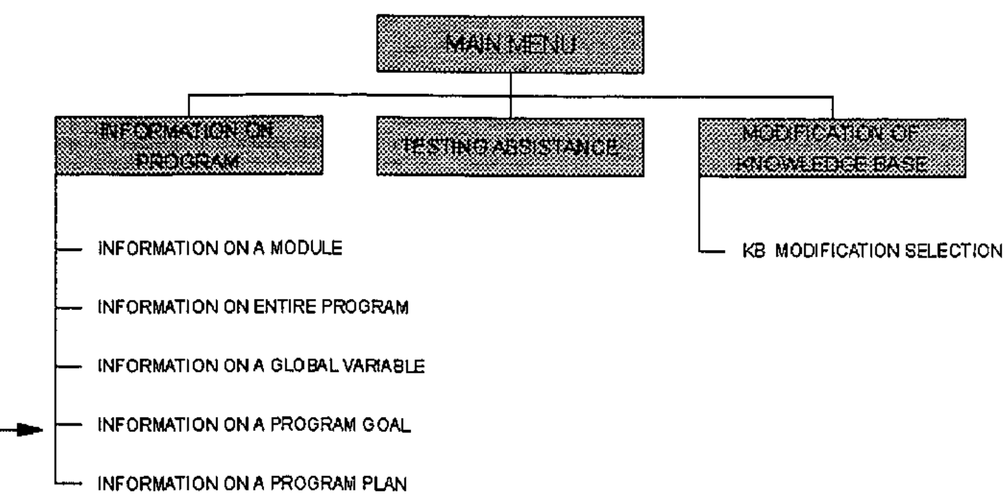
PROGRAM MENU

PROGRAM VIEWING OPTIONS

- ○ Information on a Module
- ○ Information on Entire Program
- ○ Information on a Global Variable
- ◉ Information On a Program Goal
- ○ Information On A Program Plan

OK | RETURN TO MAIN MENU

KBPUTA MENU ORGANIZATION



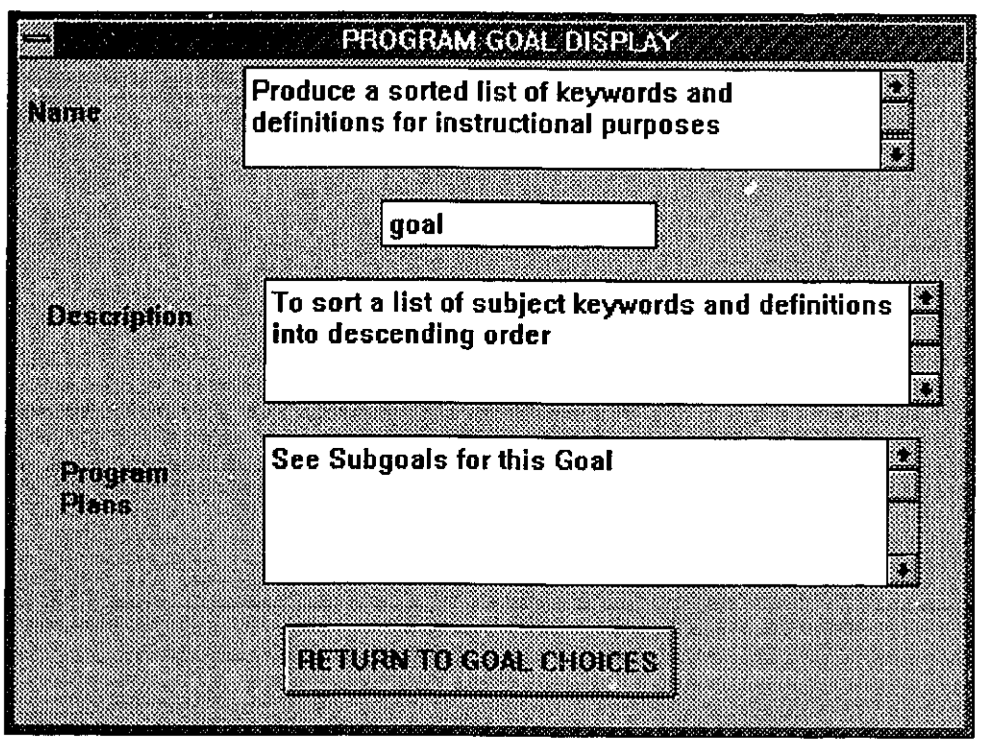
GOAL SELECTION

Produce a sorted list of keywords and definitions for instructional purposes
-input keywords and definitions
-output sorted keywords and definitions
-sort keywords and defintions into descending order

Please Choose One of the Above
Goals/Subgoals

OK

RETURN TO PROGRAM MENU


PROGRAM GOAL DISPLAY
Name
Produce a sorted list of keywords and definitions for instructional purposes
goal
Description
To sort a list of subject keywords and definitions into descending order
Program Plans
See Subgoals for this Goal
RETURN TO GOAL CHOICES

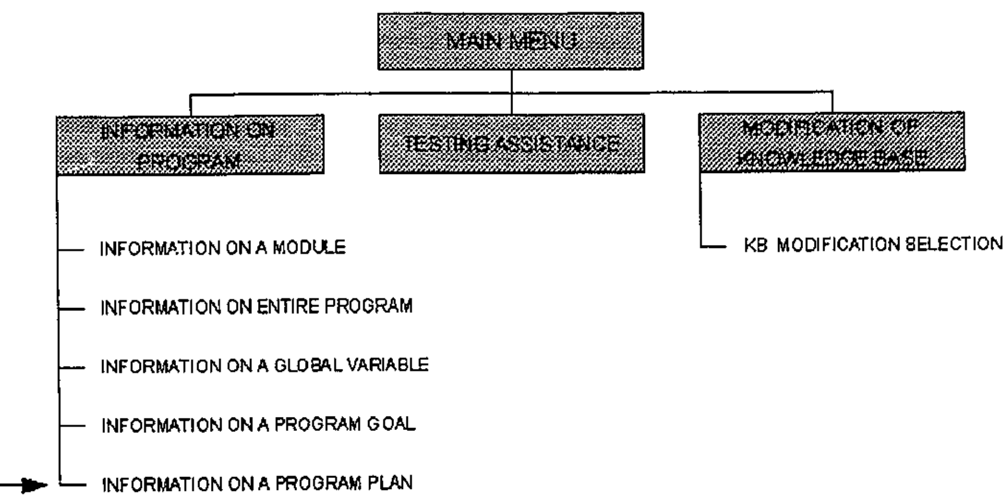
PROGRAM MENU

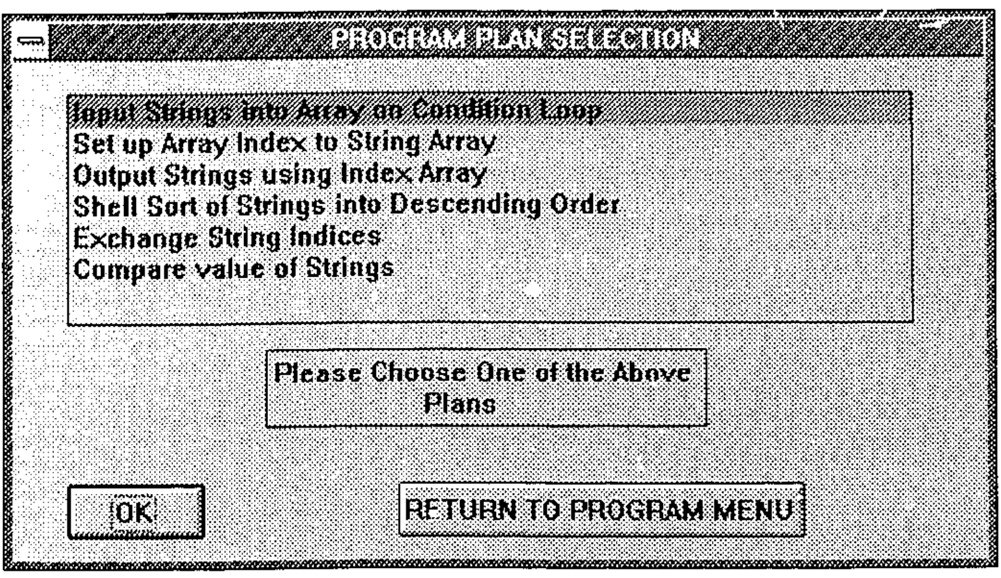
PROGRAM VIEWING OPTIONS

- ○ Information on a Module
- ○ Information on Entire Program
- ○ Information on a Global Variable
- ○ Information On a Program Goal
- ◉ Information On A Program Plan

OK | RETURN TO MAIN MENU

KBPUTA MENU ORGANIZATION




PROGRAM PLAN SELECTION
Input Strings into Array on Condition Loop
Set up Array Index to String Array
Output Strings using Index Array
Shell Sort of Strings into Descending Order
Exchange String Indices
Compare value of Strings
Please Choose One of the Above Plans
OK
RETURN TO PROGRAM MENU

PROGRAM PLAN DISPLAY

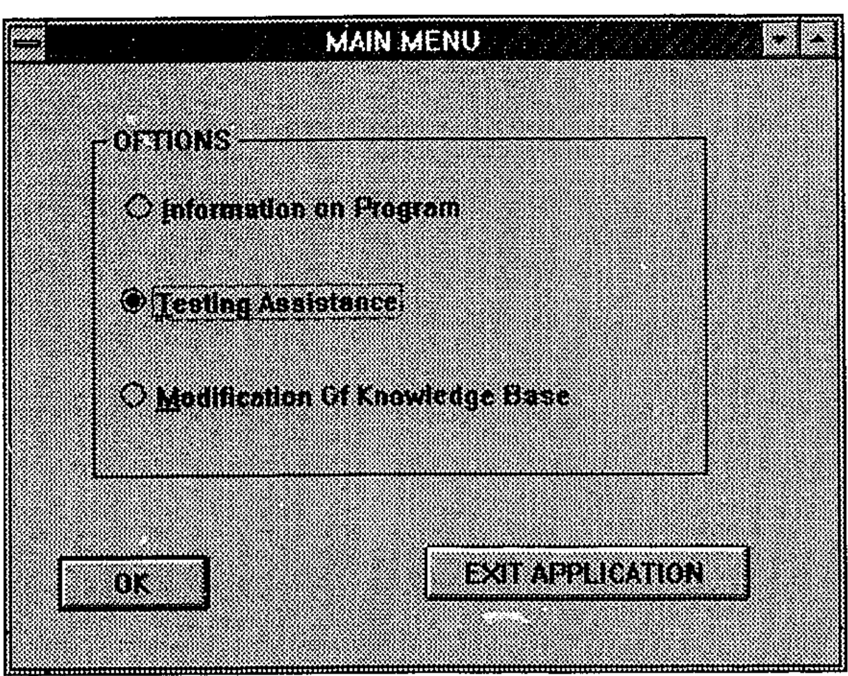
Name: Input Strings into Array on Condition Loop

Description: Input a group of strings until the last string isread or else there is not enough array storage remaning. Each

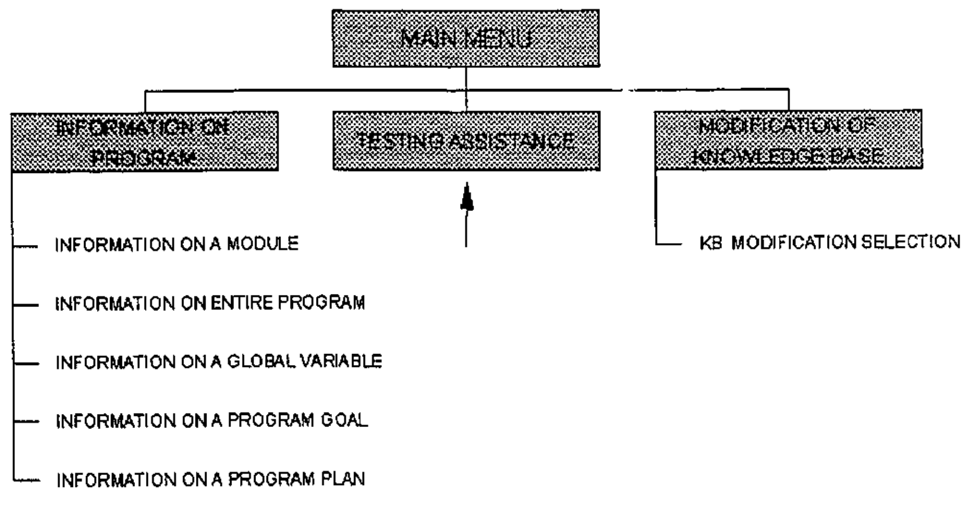
Algorithm:

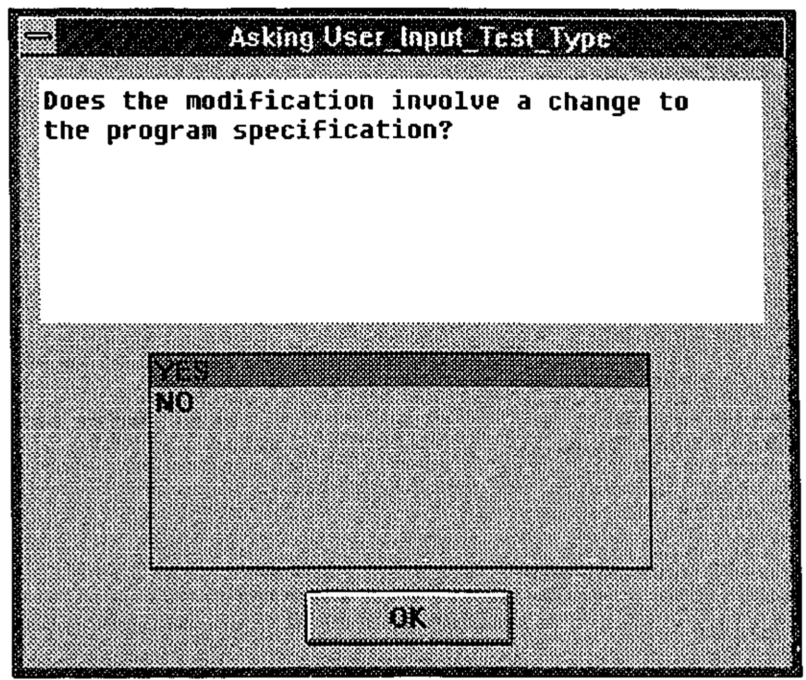
Attached Program Goal: Input Keywords and Definitions

RETURN TO PLAN CHOICES



KBPUTA MENU ORGANIZATION



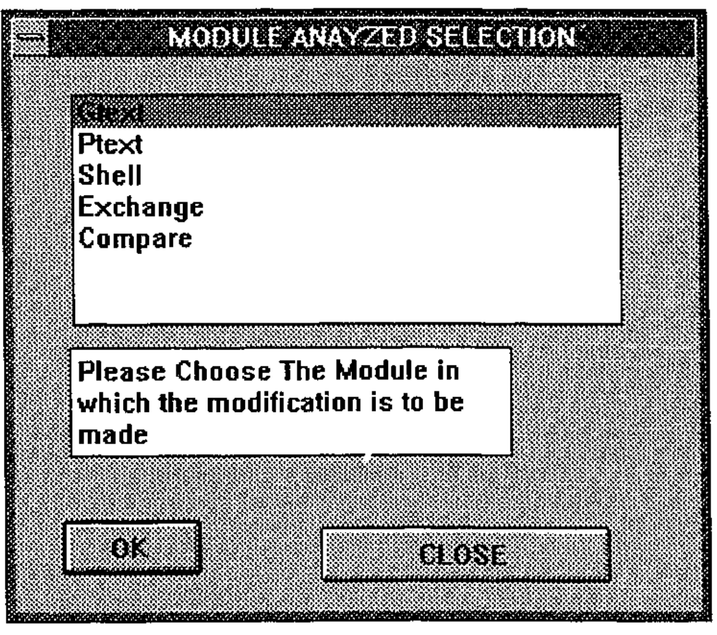

Asking User_Input_Test_Type
Does the modification involve a change to
the program specification?
YES
NO
OK

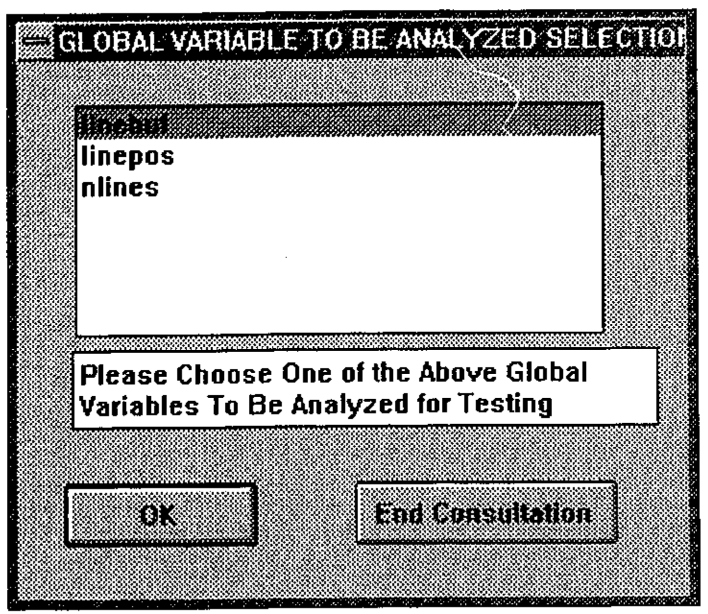
Showing Progressive_Reg_test

TYPE OF REGRESSION TESTING REQUIRED

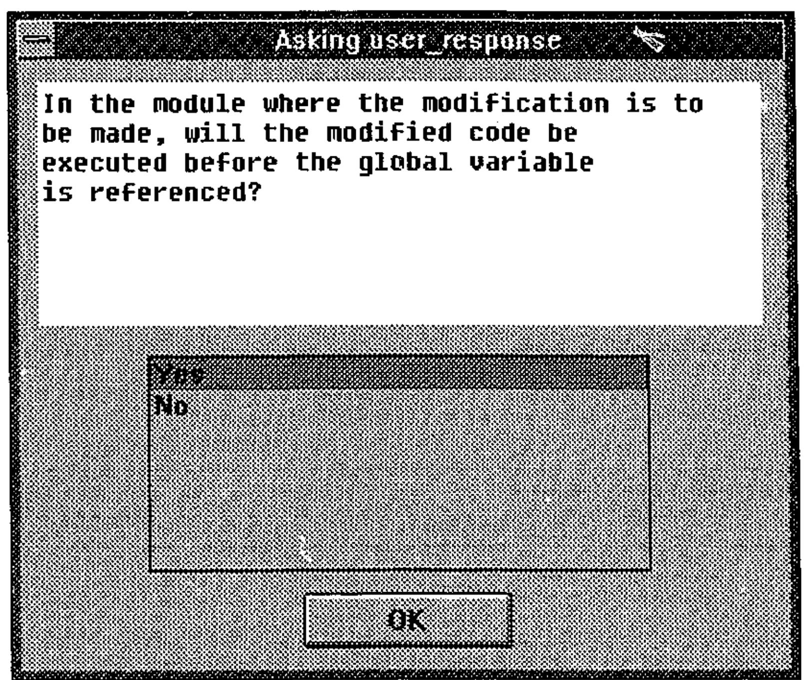
This Modification will require progressive
regression testing.
New Test Cases may be required

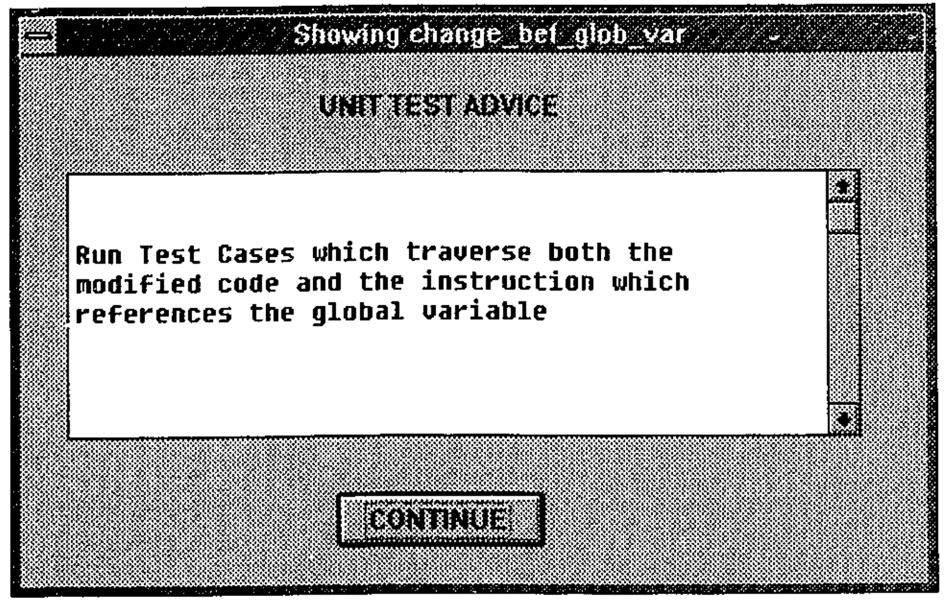
CONTINUE

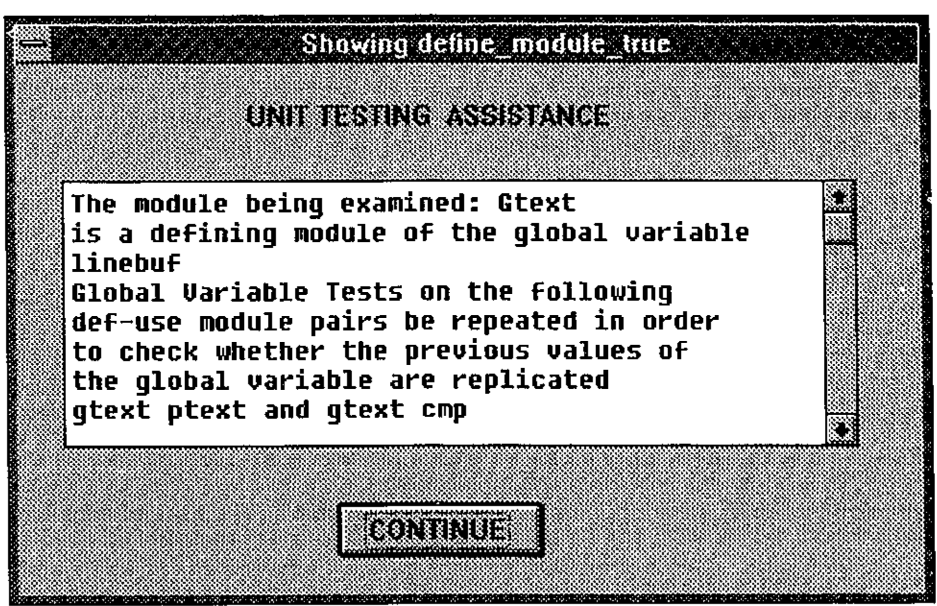

MODULE ANAYZED SELECTION
Gtext
Ptext
Shell
Exchange
Compare
Please Choose The Module in which the modification is to be made
OK
CLOSE

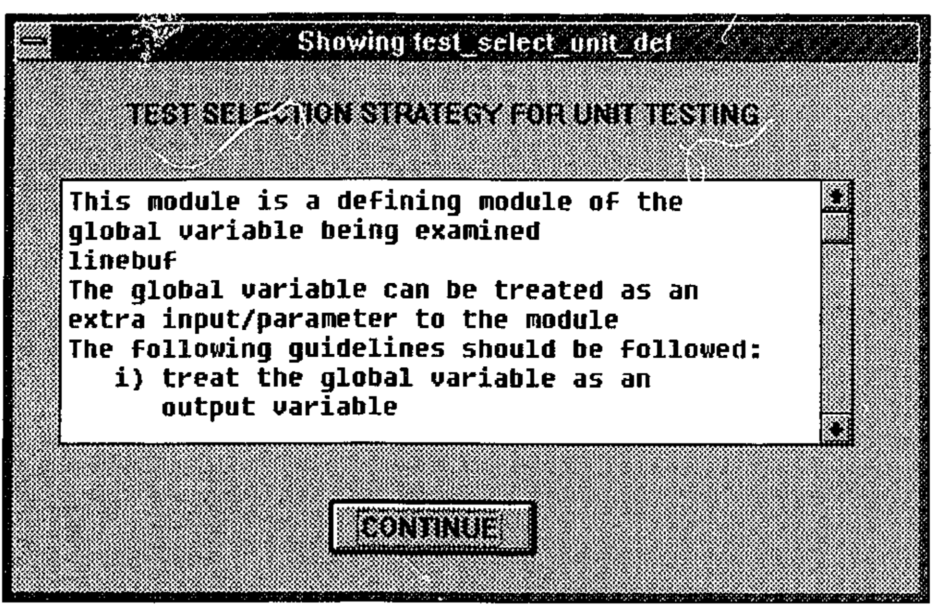


GLOBAL VARIABLE TO BE ANALYZED SELECTIO
linepos
nlines
Please Choose One of the Above Global
Variables To Be Analyzed for Testing
OK
End Consultation

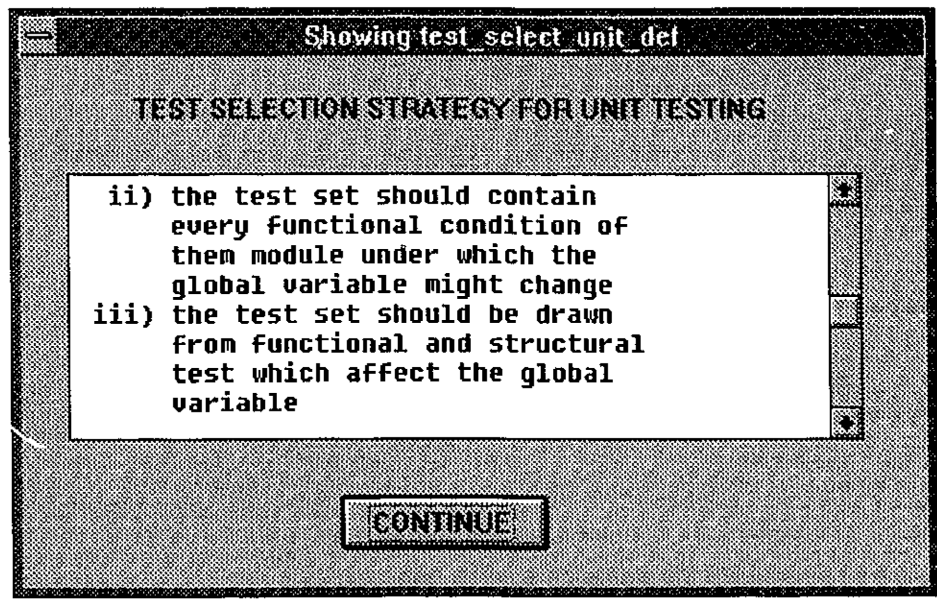

Asking user_response
In the module where the modification is to
be made, will the modified code be
executed before the global variable
is referenced?
Yes
No
OK


Showing change_bef_glob_var
UNIT TEST ADVICE
Run Test Cases which traverse both the
modified code and the instruction which
references the global variable
CONTINUE


Showing define_module_true
UNIT TESTING ASSISTANCE
The module being examined: Gtext
is a defining module of the global variable
linebuf
Global Variable Tests on the following
def-use module pairs be repeated in order
to check whether the previous values of
the global variable are replicated
gtext ptext and gtext cmp
CONTINUE



Showing test_select_unit_def
TEST SELECTION STRATEGY FOR UNIT TESTING
This module is a defining module of the
global variable being examined
linebuf
The global variable can be treated as an
extra input/parameter to the module
The following guidelines should be followed:
i) treat the global variable as an
output variable
CONTINUE

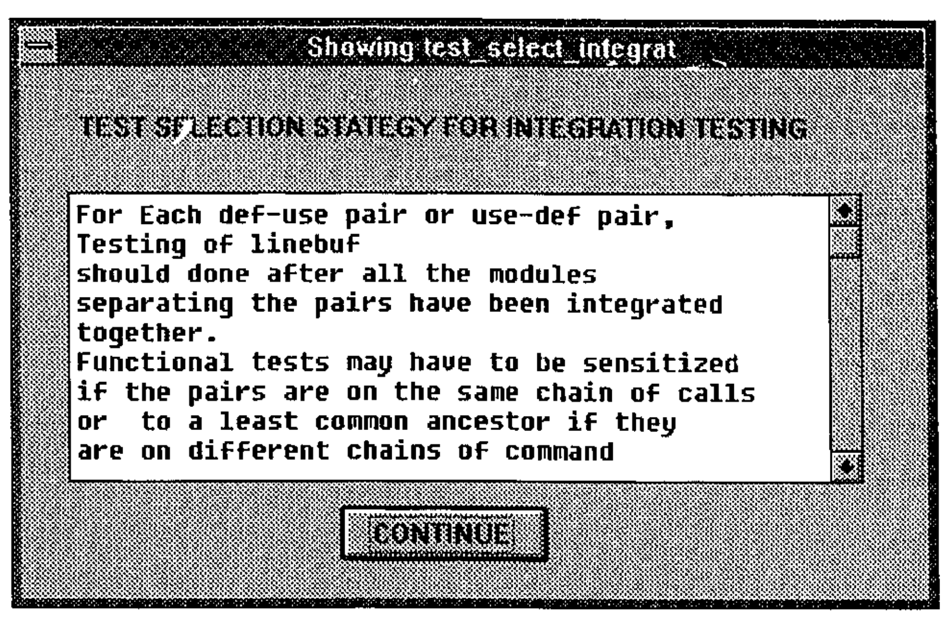

Showing test_select_unit_def
TEST SELECTION STRATEGY FOR UNIT TESTING
ii) the test set should contain
every functional condition of
them module under which the
global variable might change
iii) the test set should be drawn
from functional and structural
test which affect the global
variable
CONTINUE

Showing integrat_assist_def

INTEGRATION TESTING

Integration Tests should be performed on the
following using modules of
linebuf :
cmp and ptext

CONTINUE


Showing test_select_integrat
TEST SELECTION STATEGY FOR INTEGRATION TESTING
For Each def-use pair or use-def pair,
Testing of linebuf
should done after all the modules
separating the pairs have been integrated
together.
Functional tests may have to be sensitized
if the pairs are on the same chain of calls
or to a least common ancestor if they
are on different chains of command
CONTINUE

MAIN MENU

OPTIONS

Information on Program

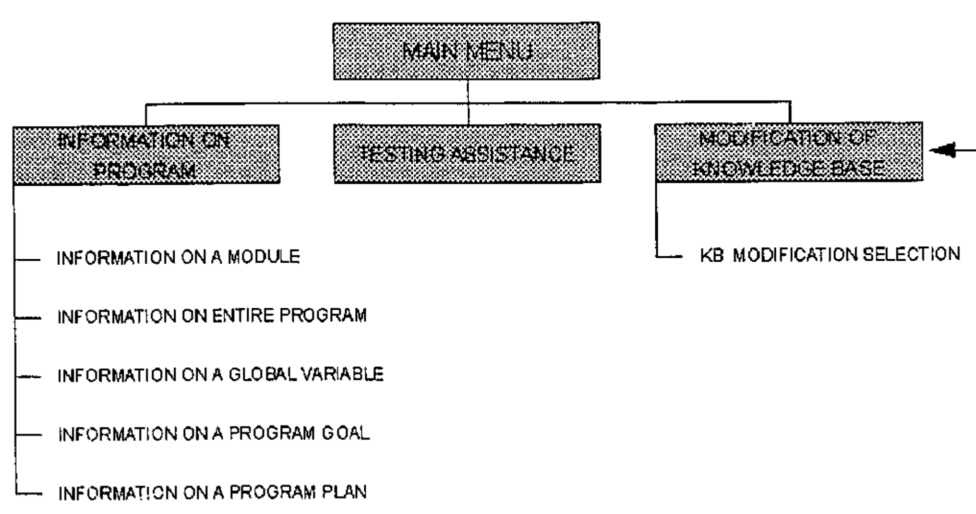
Testing Assistance

Modification Of Knowledge Base

OK

EXIT APPLICATION

KBPUTA MENU ORGANIZATION



KNOWLEDGE BASE MODIFICATION SELECTION

OBJECT

- ○ Module
- ◉ Goal
- ○ Global Variable
- ○ Plan

CREATE OR MODIFY

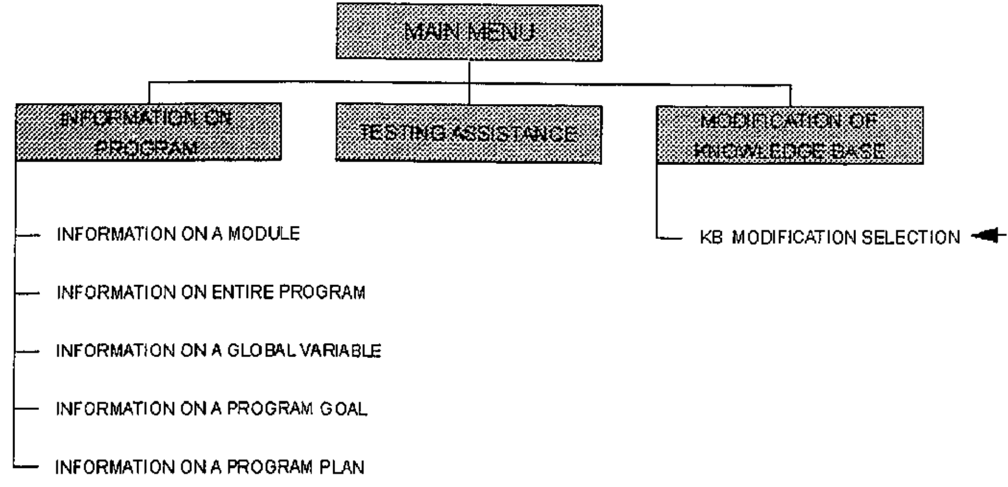
- ◉ Create A New Object
- ○ Modify an Existing Object

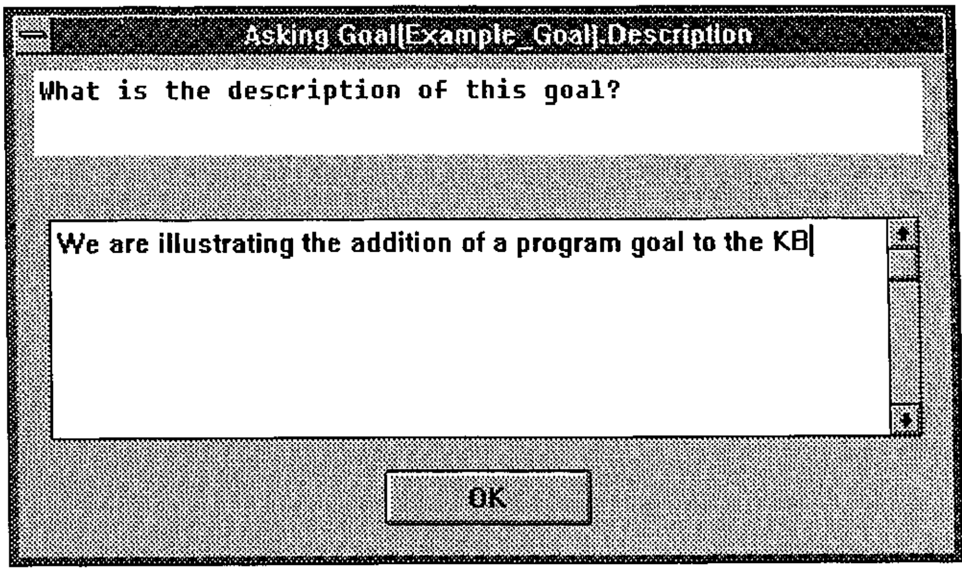
Name of Object: EXAMPLE_GOAL

Note: WORDS MUST BE SEPARATED BY DASHES
e.g. Module_Name

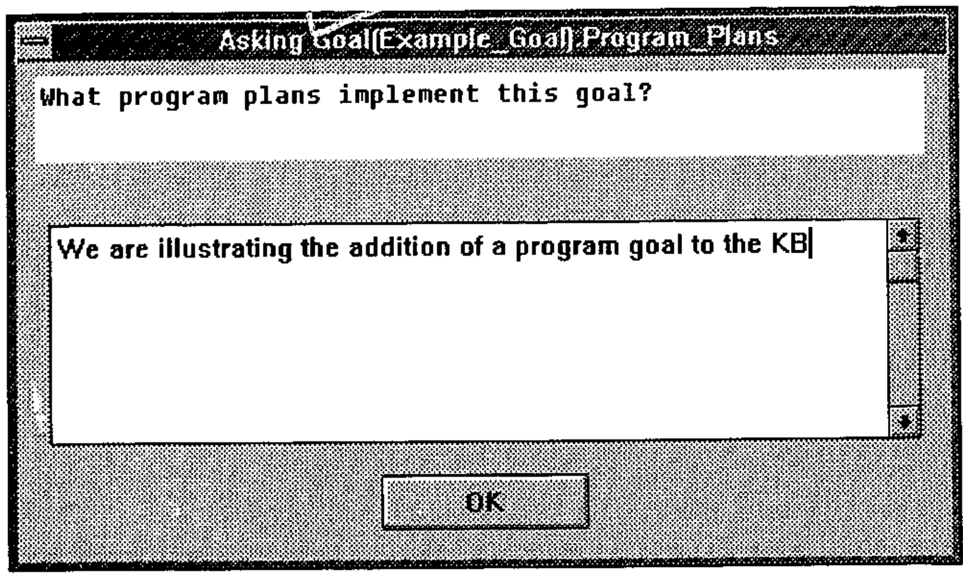
OK | RETURN TO MAIN MENU

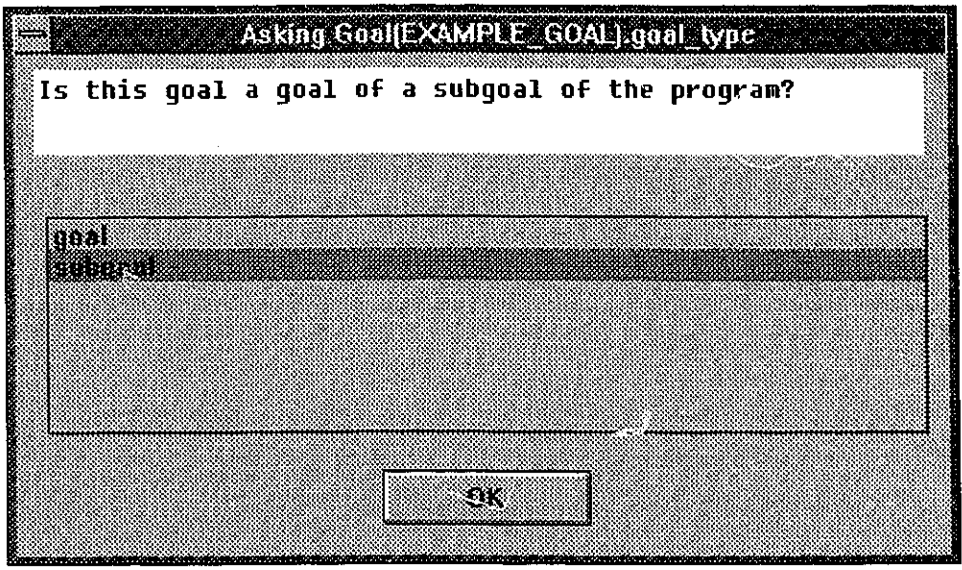
KBPUTA MENU ORGANIZATION





Asking Goal[Example_Goal].Description
What is the description of this goal?
We are illustrating the addition of a program goal to the KB
OK


Asking Goal(Example_Goal).Program_Plans
What program plans implement this goal?
We are illustrating the addition of a program goal to the KB
OK


Asking Goal[EXAMPLE_GOAL].goal_type
Is this goal a goal of a subgoal of the program?
goal
subgoal
OK