An Interactive, Graphical, Program Design and Development Environment

Kaizad Bomi Heerjee

This thesis is submitted to the Council for National Academic Awards in partial fulfilment of the requirements for the award of the degree of Doctor of Philosophy

Department of Mathematics and Computer Studies
Dundee College of Technology
Bell Street, Dundee
Scotland

September 1988

Table of Contents

1. Introduction 1

2. Software Design 7

2.1. The Jackson Design Methodology Reviewed 17

3. An Animated Programming Environment 23

3.1. The Design and Programming Mode 25

3.1.1. Software Development Using APE 29

3.1.1.1. System Design 29

3.1.1.2. Programming 30

3.1.1.3. Testing and Debugging 31

3.1.1.4. Coding and Documentation 33

3.1.1.5. Modification and Maintenance 33

3.2. Text Processing Mode 33

4. The APE Environment 37

4.1. General Manager 37

4.2. Screen Manager 38

4.3. History Mechanism 39

4.4. Syntax Directed Editor/Interpreter 39

4.5. Code Generator 39

4.6. Program Animator 40

4.7. Document Processor 41

4.8. On-line Help 42

4.9. Learn Package 42

5. Interface Design of the APE System 44

5.1. Design Considerations 45

5.1.1. Screen Design 47

5.1.2. Dialogue Design 50

5.1.2.1. Menu Design 56

6. Implementation 60

6.1. Extensions to Jackson Structure Diagrams 60

6.2. Internal Representation 66

6.2.1. Linking Nodes Together 68

6.2.2. Displaying the Coding Tree 74

6.2.3. Displaying the Skeleton Tree 77

7. Validating and Verifying the Implementation 80

7.1. The Uses of Formal Methods in Existing Systems 81

7.2. The Notation 84

7.3. Specification of a Module in Standard ML 88

7.3.1. The Environment 92

7.3.2. The Paste Operation On Trees 95

7.4. Questions and Problems Identified by Interrogating the Specification 98

8. Empirical Evaluation of the APE System 107

8.1. Questionnaire Evaluation 109

8.1.1. Apparatus 109

8.1.2. Subjects 110

8.1.3. Method 110

8.1.4. Results 112

8.2. Comparative Evaluation 114

8.2.1. Apparatus 114

8.2.2. Subjects 114

8.2.3. Method 115

8.2.4. Results and Discussion 115

8.2.4.1. Quality of the Solution 116

8.2.4.2. Speed of Performance 119

8.2.4.3. Correlation Between Quality and Speed 121

9. Conclusions 125

9.1. Experiences in Applying Formal Methods 126

9.2. Empirical Evaluation of the APE System 128

9.3. Future Work 133

Appendix A: The Curses Library Package 135

Appendix B: Correctness Proofs of Standard ML Functions from its Axioms 139

Appendix C: The Questionnaire 148

Appendix D: Questionnaire Analysis 158

Appendix E: Decision Table Implementation 162

References 166

Publications:

1. The Design and Evaluation of an Animated Programming Environment, *People and Computers – Proc. of HCI '88*, Manchester (UK), September 1988 ..... A-1

2. Experiences in Applying Formal Methods to Existing Software, *Proc. of a Formal Methods Workshop*, Middlesbrough (UK), July 1988 ..... B-1

3. The Rapid Implementation of SQL - A Case Study Using YACC and LEX, *Information and Software Technology*, vol 30, no. 4, pp. 228-236, Butterworth & Co. (Publishers) Ltd., Guildford (UK), May 1988 ..... C-1

Table of Figures

2.1. Process-Oriented Design Techniques 10

2.2. A Jackson Structured Diagram 18

2.3. A Sequence Representation 18

2.4. An Example of the Sequence Construct 19

2.5. An Iteration Construct 19

2.6. An Example of an Iteration Construct 19

2.7. A Selection Construct 20

2.8. A Single Selection Construct 21

2.9. The Program Structure for Lunch 22

3.1. A Hierarchical Structure 25

3.2. An Overview Tree 28

3.3. Evolution of Debuggers 32

4.1. UNIX Document Processing Tools 42

5.1. Diagrammatic View of the Screen Layout 48

5.2. APE Command Language Commands 56

6.1. Construct of a WHILE Loop in APE 62

6.2. Implementation of a WHILE loop in APE 62

6.3. A Selection Construct in APE 63

6.4. A Sub-problem Representation 64

6.5. Implementation of a Selection 64

6.6. Initial Implementation of a Selection in APE 64

6.7. A Selection Without the Else Clause 65

6.8. Selection Construct Implementation in APE 66

6.9. A Selection with an Else Clause 66

6.10. Implementation of a Selection in APE 66

6.11. Eldest Son-Young Brother 69

6.12. Internal Representation of a Node 69

6.13. The C Data Structure for a _TREE 69

6.14. The _LOCS C Data Structure 72

6.15. The _DDEFS C Data Structure 73

6.16. The _SPEC C Data Structure 73

6.17. Node Linkages 74

6.18. The C Data Structure for a _LINENODE 75

6.19. Data Structures for the Skeleton Tree 78

7.1. The Specification for the Paste Operation 98

8.1. Marks Assigned to Nineteen Subjects 117

8.2. Task Completion Times (in minutes) 119

8.3. Quality vs. Time Taken for the APE Group 121

8.4. Quality vs. Time Taken for the Non-APE Group 121

D.1. User Responses to the Questionnaire 160

D.2. Median Ratings 161

ACKNOWLEDGEMENTS

No work such as this could have been completed without the help of numerous people, and I have been particularly fortunate in the co-operation that I have received from others during my stay in this country.

In the first place I thank my parents, Bomi and Shireen, and my two sisters, Trista and Havovi, for their unfailing love, support and patience that has been a constant source of encouragement to me.

As for all academic writers, I owe an intellectual debt to the many people with whom I have discussed my research. Dr Colin Miller, Dr William Samson and Dr Michael Swanston were respectively my director of studies and associate supervisors, and to each of them I would express my sincere thanks for having willingly proffered their advise, throughout the duration of this project and on every aspect of the work described herein. Useful discussions were had with my industrial advisor, Mr K G Murray (Ferranti Defence Systems plc), and my thanks go to him for giving up his time. I would also like to extend my thanks to Miss Pat Dugard for her help with the statistical analysis.

Finally, but by no means last of all, I would express my sincerest thanks and gratitude to three friends; Anita Reinboth, Hosung Randelia and Pritam Chita, for their continual help, advice, support and encouragement.

An Interactive, Graphical, Program Design and Development Environment

Kaizad Bomi Heerjee

ABSTRACT

New software development methodologies are being produced with increasing frequency. The latest techniques claim to produce software of unprecedented reliability and productivity, yet are seldom substantiated by empirical evidence. Researchers in the field of human-computer interaction have long held the view that well designed interactive systems increase performance levels over conventional techniques. Intuitively this seems logical, but very little work has been done to substantiate this claim empirically.

This thesis aims to show that well designed structured programming environments provide productivity gains and increase performance levels over conventional techniques.

An Animated Programming Environment (APE), has been developed which is an interactive, graphical, program design and development system that embodies structured programming and top-down design. The system supports the development of programs for a variety of block structured languages whilst working conceptually at the level of Jackson diagrams.

Formal methods were applied to validate and verify the APE system. The immediate benefits are an increased understanding of the system and the detection of some errors in the implementation. By interrogating the implementation and documentation, axiomatic specifications were written and a prototype of the APE system developed in Standard ML.

The principal benefit of constructing a formal model is the development of a framework to aid communication between personnel involved with system maintenance. The model can also be used to investigate future changes, and since this framework provides relevant abstraction of user and system behaviour it should facilitate improved documentation and user learning.

Evaluation of the system was carried out during the design and implementation stages of the development life-cycle. The evaluation was based on responses to a questionnaire and a comparison with conventional means of generating code. The questionnaire evaluation elicited users' general impressions about the system and its interface, and their detailed views on more specific aspects of the system. The comparative evaluation showed no difference in the median quality of the solution to a programming problem, but a significantly reduced time and variance in quality compared to conventional methods.

CHAPTER 1

Introduction

Software development is widely acknowledged to be expensive, time-consuming and error-prone. As software production, maintenance and testing costs escalate in proportion to the total system cost, the computing profession can ill afford to continue to produce substandard software. Yet, this software crisis persists due mainly to poor problem definition and program design.

New software development methodologies are being produced with increasing frequency with the latest techniques claiming to better the previous ones in their ability to address the software crisis. The proponents of these structured methods often claim results of unprecedented software reliability, productivity and performance. However, these claims are seldom substantiated by empirical evidence.

Researchers in the area of human-computer interaction have long held the view that well designed interactive systems increase performance levels over conventional techniques, such as editors and compilers. Although this seems logical very little work has been done to substantiate this claim empirically. Greater efforts are being devoted to the development of these interactive systems, but their positive impact is seldom evaluated.

It is the purpose of this thesis to demonstrate empirically that well designed structured programming environments lead to increased productivity and performance. To this end, an Animated Programming Environment, APE [40,42,43], has been developed which is an interactive, graphical, program design and development system, that embodies structured programming and top-down design. The system supports the development of programs for a variety of block structured languages whilst working conceptually at the level of Jackson structured diagrams.

The system generates code in a high-level language, incorporates automatic syntax checking and generates data definitions automatically where possible. A run-time animation component has also been incorporated that enables program monitoring and debugging.

Using the same interface, the system has been extended to provide automated support for UNIX† dependent text processing facilities. It includes facilities for document structuring and the automatic invocation of these text processing facilities. The system is also compatible with the Revision Control System [105,106] for storing, retrieving and logging revisions of text.

Although the system is interactive and incorporates an optional use of menu facilities, it is not hardware specific and can be used with any terminal that supports direct cursor

† UNIX is a trademark of Bell Laboratories.

positioning.

Human factors techniques were used throughout the design and implementation stages of the development life-cycle. The characteristics of the user interface were based, as far as possible, on available human factors knowledge and techniques.

Formal methods have been applied to validate and verify the implementation of the APE system. As an immediate benefit this process resulted in an increased understanding of the system and also led to the detection of some errors in the implementation. The main aim in formally specifying the APE system was to build a model of the system that was consistent with the implementation. The model would be used to describe precisely the behaviour of the system, check for correctness, and test possible changes, modifications or reimplementations of the system.

During the model building process, questions can be formulated about the desired behaviour of the system that may indicate problems either in the system design or documentation. In the longer term the formal model should provide a more precise description of the system for educating new personnel on the basis of improved documentation and form a good starting point for specifying proposed changes to the system itself.

The evaluation was based on responses to a questionnaire and a comparison with conventional means of generating code. The questionnaire evaluation was designed to elicit users' general impres-

sions about the system and its interface, and their detailed views on more specific aspects of the system. The comparative evaluation was designed to provide an objective test of the value of the system in software development.

1.1. Structure of the Thesis

The thesis continues in Chapter 2 with a review of relevant literature. Software design techniques, including architectural and detailed design stages, are surveyed. Recent advances in distributed software systems design methods, visual programming and programming visualisation systems are also reviewed. In the absence of a complete methodology for designing software, a set of interim measures are outlined. Following that, an overview of Jackson's method of program design, the methodology on which the APE system is conceptually based, is presented.

The APE system is introduced in Chapter 3. Interactive graphical and text based systems are reviewed and the drawbacks of each are noted with reference to the APE system. The process of software development in the APE system is then discussed and an overview of the text processing mode of operation is presented.

The APE system consists of nine independent modules. These components that determine the complete APE environment are outlined in Chapter 4.

The design of the human-computer interface of the APE system

is discussed in Chapter 5. In particular, the design process and the characteristics of the user interface are described.

Some implementation details are presented in Chapter 6. These include a discussion on:

- the extensions made to the Jackson's methodology in order to enable program generation
- the underlying data structures
- a method used in the system for displaying and producing a line-printer form of the design diagram.

The application of formal specifications to validate and verify the implementation is presented in Chapter 7. The benefits of applying formal methods, particularly to existing software, are described as is the specification process itself. Axiomatic specifications are written and a prototype of a module in the APE system is developed in Standard ML [38]. The formal specification was developed by interrogating the implementation and documentation. Several problems exposed as a consequence are presented.

The evaluation of the APE system is described in Chapter 8. The evaluation was based on responses to a questionnaire and a comparison with conventional means of generating code. The first experiment was designed to evaluate the user interface of the system and was based on responses to a questionnaire. The second evaluation was designed to provide an objective test of the value

of the system in software development.

Conclusions drawn from this thesis are presented in Chapter 9. Ideas for further work and a discussion on the directions for future research are also presented.

Additional topics of a general nature are described in the appendices and include:

- an overview of the curses library routines; the package used by the APE system for driving its user interface
- examples of program verification; correctness proofs of Standard ML functions from their axioms
- the questionnaire used in the evaluation
- the analysis of user responses to the questionnaire
- the task presented to users in the comparative evaluation.

CHAPTER 2

Software Design

During the late 1960s and early 1970s, software engineering was largely a research activity and had little impact on the software development practices of the time. Special emphasis was placed on performance and secondary emphasis on efficiency. Today, economic pressures have led to four more factors being recognised as requiring special emphasis – reliability, extensibility, re-usability, and maintainability.

Maintaining and modifying software accounts for eighty per-cent of the total software expenditure [17]. As the volume of existing software grows, so does the expense of maintenance and modification. This escalating trend can be counteracted by design-ing and implementing software in a way that minimises errors and maximises the ease with which such errors can be corrected and modified. The software crisis has led to the emergence of some of the most important concepts in modern software development tech-niques, including

- top-down design
- stepwise refinement [115]
- modularity [80,63]

- structured programming [22,71].

The recognition of the relationship between programming languages, problem solving, and high quality software has also led to the design of many structured programming languages, notably Pascal [114], Ada, and Modula-2.

Structured programming was the first of these concepts to have a widespread impact on the way that software was written, although in a somewhat different sense than was originally intended by Dahl *et al* [22]. They considered code generation and program comprehensibility rather than the entire software life cycle.

A more significant impetus came from the 'life cycle' concept that provides a framework for bringing together software production and management techniques. Wasserman [110] presents a good analogy between this concept and with the steps involved in developing a custom-built house. This analogy applies to developing software, since the problems of specification, design, and implementation arise in both domains.

Research in software engineering has since resulted in the proposal of several new methodologies and tools for the production of reliable and maintainable software systems. Each different technique has its own notation and set of rules defining how design should be expressed. Many of these design techniques use a diagrammatic representation of software objects and their inter-

relationships. By representing programs as semantically suggestive graphical images, Raeder [84] believes that one can narrow the gap between mind and medium. Two-dimensional displays for programs, such as flowcharts and even the indenting of block structured programs, have been known to be helpful aids in program understanding [95]. More recently several program visualisation systems, such as those by Myers [74], Baecker [6], and Brown [15], have demonstrated the usefulness of pictorial displays. Clarisse [20] claims that graphical programming uses information in a format that is closer to the user's mental representation of a problem, thus allowing data to be processed in a format closer to the way objects are manipulated in the real world.

Various approaches have been developed for designing and producing reliable and maintainable software using diagrams. These approaches to software design are a process that translates software requirements into a detailed design representation. Most design methodologies adopt a life cycle approach to describing software development. They identify certain activities that signal the beginning of a project, the tasks to be performed, and the products that will be delivered at the end. However, this division of task that is required to get from the beginning to the end, varies widely from one methodology to another.

There are two major design stages involved with this kind of design process, namely *architectural* and *detailed* design [29,32,83]. The architectural design stage deals with the

specification of the overall system requirements. The emphasis is on defining the overall structure of the system, decomposing the system into modules and precisely specifying the interface between modules. The detailed design stage transforms the architectural design into an implementable software system. This is possible as the details for each module are well defined. The emphasis here is on the selection and evaluation of algorithms with the aim of producing a representation of the system in a form that can be easily built. Most of the current popular diagramming techniques can be classified into these two design stages.

Architectural design techniques can be further sub-divided into two groups; *process-oriented* and *data-oriented* approaches. The process-oriented approach emphasizes the process of decomposition and structure in creating software architecture. Processes can be characterised as modules with an interface and an internal structure that may be called on to do a task. These methodologies aim to help the designer characterise the interface, internal structure, and control mechanism of these processes and include techniques as shown in figure-2.1.

The data-oriented approach emphasizes the data design component of software systems and techniques for driving the data design. The design methodologies in this category include the Object-Oriented design approach [13] and the Conceptual Database design method [23]. Object-oriented design is based on the concepts of information hiding and data abstraction. Some process-

Functional Decomposition [80,115]
Data Flow Design Methods
    Structured Design [124]
    Structured Analysis Design Technique (SADT) [87]
HIPO (Hierarchy, plus input, Process, Output) [51]
Data Structure Design Techniques
    Warnier-Orr Design Method [108]
    Michael Jackson Design Method [52]
Module Interconnection Languages
    MIL [24]
    Abstract Design and Program Translator [4]
    Graph Description Language [122]

Figure 2.1: *Process-Oriented Design Techniques*

oriented design methodologies, such as functional decomposition and modular programming, attempt to control the complexity inherent in large systems by trying to hide the details of how one part communicates with the others. However, these concepts are not rigorously enforced. Here, object-oriented methodologies succeed where process-oriented techniques fail.

In a process-oriented design, the basic decomposition criterion is that each step in the process represents a module, where as in an object oriented method an object in the system represents a module. In the object oriented design approach all components of a system are viewed as objects. Each object consists of some memory and represents an individual addressable entity with a set of operations that may be performed on its contents.

Object-oriented design techniques require a change in individual problem-solving and thinking strategies. This method is characterised by the central idea of using messages to communicate with objects and is more natural than process-oriented design

methodologies as it allows for the creation of different kinds of abstract data-types. These abstract data-types can then be directly mapped into the designer's view of the problem.

In designing software systems, *object-oriented* design techniques have several advantages over conventional *process-oriented* design methods, namely

- better productivity – which is addressed by abstraction
- maintainability – that is achieved by information hiding
- data integrity
- security – which along with data integrity is defined by protection domains.

Although this design approach is more suitable for control-dominant or real-time systems, features that are not well addressed in conventional design methods, its theoretical foundations need more study and applications need to be developed [123].

The architectural design techniques outlined above can only be used to show the module-level structure of a system. Design representation techniques are available that can be used to transform the architectural design into an implementable software system. These design techniques show the detailed logic of a program and can be classified into two categories; graphical representation techniques and language representation techniques.

Historically, flowcharts have served as the primary tool for graphically showing the detailed logic of program modules. Although flowcharts are easy to use they have many disadvantages [110], as a result of which they are no longer favoured as a representation method. Structured programming enthusiasts suggest that they do more harm than good, and flowcharts have now largely been superseded by alternative methods that support structured programming. These methods include Nassi-Shneiderman Charts (N-S Charts) [76], Hierarchical Graphs [120], and Chapin Charts.

Of the language representation techniques, the Program Design Language (PDL) [16], is the most widely used. Alternatively known as 'Structured English', 'pseudocode' or 'meta-code', depending on the amount of formalism involved, a PDL can be viewed as a high-level programming language that uses English-like statements to express the detailed logic of a program module.

Of the design techniques currently available, many of them, such as functional decomposition, data structure design methods and data flow design methods, have been extensively used with a large degree of success. Yet, there is still room for innovation in this area. A survey of the popular techniques is given in Freeman [32], Heerjee [44], Yau and Tsai [123], and Bergland [10].

The design of distributed software systems is more complicated, owing to design constraints and interactions of software components of the system. Most of the methods mentioned above are primarily concerned with sequential software systems and do not

directly address the design problems associated with parallel and distributed systems. Several design methods have been proposed, including Petri Nets [81], Communicating Sequential Processes [49], and Attributed Grammars [64], and some techniques have been developed that verify the design of distributed systems [121].

Recent technological advances have made high-resolution, bit-mapped graphics available on inexpensive microcomputers. This has resulted in an upsurge of interest in program visualisation and visual programming [35,57,75,27]. A survey of this subject area is presented by Raeder [84].

Work in the area of visual programming and program visualisation has led to better understanding of the role of diagrams, particularly dynamic diagrams, in the software development life cycle. Considerable work, however, remains to be done in areas such as providing support for data abstraction and exploring ways of ensuring that dynamics enhance the clarity of visualisation. In particular, Brown *et al* [14] suggest that a more complete dynamic vocabulary appropriate to programming must be developed.

Software development environments and integrated programming environments in particular, have recently received considerable attention as they offer the prospect of improving the quality of the software development process. These environments aim to provide designers and developers with an integrated computer-aided system that supports specification, design, coding, testing and

maintenance and is consistent with the level of abstraction of the software development process and the application domain of the intended products.

In summary, there are no complete methodologies, yet, for partitioning large problems, or for producing better documentation that overcome the shortcomings and attributes of existing methodologies. There are no guidelines on design methods to solve a particular problem, and no generally accepted metric for quantifying program complexity. There is limited help for maintenance programmers, and more published examples of real-time applications that use these design methods are required. While there are currently no methodologies that attempt to cover all these areas, one attempt in this direction is the Multiview approach.

The Multiview approach [117] is a step towards a combination of methodologies. If a common development database could be defined it is possible that many distinct approaches could contribute to software development using the database as the interface. The developer would then have the flexibility to select approaches that met specific needs, while different people could use the methods they preferred. However, judging by the developments in the area of hardware this open interface approach is unlikely to progress for some considerable time.

Until a complete method for designing software is found, Bergland [10] proposes a set of interim procedures that designers should follow for designing the structure of a large program. They

include:

(1) applying a bottom-up approach, using the principle of abstraction to hide the peculiarities of the hardware and create a desirable virtual machine environment

(2) mapping the system flow diagram into this virtual machine environment

(3) constructing a data flow model of the problem, or using the inversion process as stated by Jackson [52]

(4) constructing data structure diagrams that correspond to each data flow path

(5) using data structure design techniques for combining simple programs

(6) implementing each cluster as a concurrent, asynchronous process under a suitable programming environment - for example, UNIX.

Although these interim measures do not all fit together to form a design method, they do present a reasonable approach until a complete method of designing software is found. In the absence of a complete methodology it is appropriate to consider one of the most popular and productive program design methodologies currently in use.

2.1. The Jackson Design Methodology Reviewed

The Jackson Structured Design technique, which is similar to another established diagramming technique by the name of Warnier-Orr [108] diagrams, has the advantage that it represents both the data structure and program structure. In addition, the program structure is derived from the data structure, the input and output data of a program being used to create the program structure.

Jackson describes his own methodology as having the following characteristics:

(1) It is non-inspirational; it depends little, or not at all, on invention or insight by the designer.

(2) It is rational; the design procedure is based on reasoned principles, and each step may be validated in the light of these principles.

(3) It is teachable; people can be taught to practise the method and two or more programmers using the method to solve the problem will arrive at substantially the same solution.

This leads to consistent program design, as the program is derived from the data that must be designed first, that is a helpful and desirable approach.

(4) It is practical; the method itself is simple and easy to understand, and the design produced can be implemented without difficulty in any ordinary programming environment.

Jackson uses a hierarchical structure (Figure-2.2) to represent program structures. This tree structure is similar in appearance to a structure chart, comprising boxes arranged in levels and connected by lines.

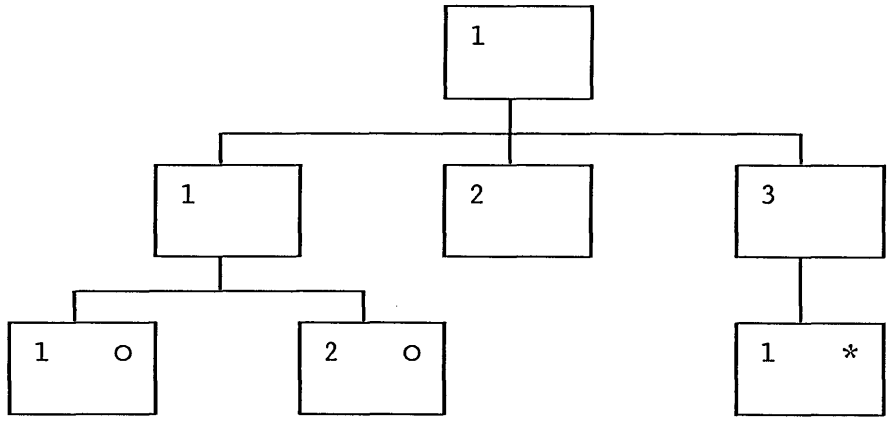


Figure 2.2: A Jackson Structured Diagram

There are three basic component structures to define the solution of a problem using this hierarchical approach, namely

1. sequence
2. iteration (or loop)
3. selection.

(1) **Sequence:** A sequence has two or more components, each being executed once, in order (Figure-2.3).

The figure suggests that the solution of a given problem is the sequence with components A then B then C. The notation is extensible to denote a sequence of several component parts. An example of this construct is demonstrated by the event of having *Lunch* which can be considered as a meal consisting of three courses; the *First Course* followed by a *Main Course* and

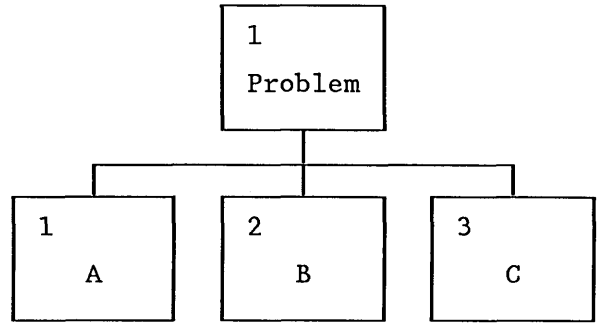


Figure 2.3: A Sequence Representation

finally *Coffee* (Figure-2.4).

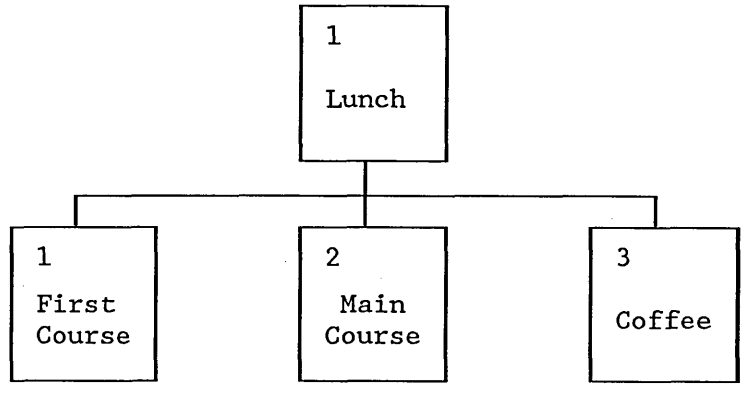


Figure 2.4: An Example of the Sequence Construct

(2) **Iteration:** An iteration, or repetition, has one component occurring zero or more times. Figure-2.5 shows the diagrammatic representation for an iteration.

The solution of the problem is zero or more occurrences of component A. The asterisk shows the repeated component. An example of this construct would be to have *Cups* of coffee (Figure-2.6).

(3) **Selection:** A selection has one or more components of which one, and only one, may occur (Figure-2.7).

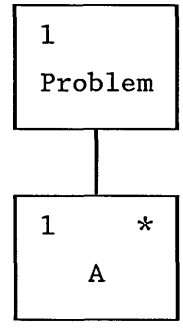


Figure 2.5: An Iteration Construct

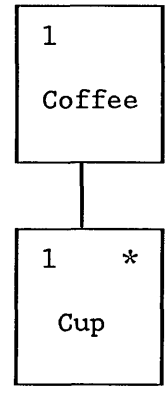


Figure 2.6: An Example of an Iteration Construct

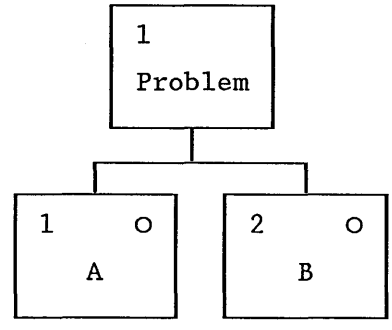


Figure 2.7: A Selection Construct

The solution of the problem is either component A or component B. If component B is a null operation, then the solution is either component A or nothing. Figure-2.8 represents the usual shorthand for this case.

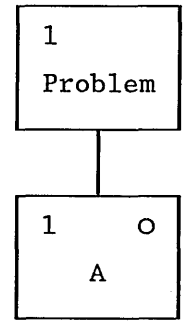


Figure 2.8: A Single Selection Construct

As for a sequence, the notation can be extended for several component parts (or choices).

These three constructs can be combined together to generate the complete program structure. For example, the complete event of having *Lunch* can be specified as having a *First Course* which consists of either *Soup* or *Fruit Juice*, followed by the *Main Course* and finally zero or more *Cup*'s of *Coffee* (Figure-2.9).

An Animated Programming Environment, APE, has been developed that is a powerful and flexible software development system, conceptually based on Jackson structured diagrams. The design, validation and empirical evaluation of this system is discussed in the remainder of this thesis.

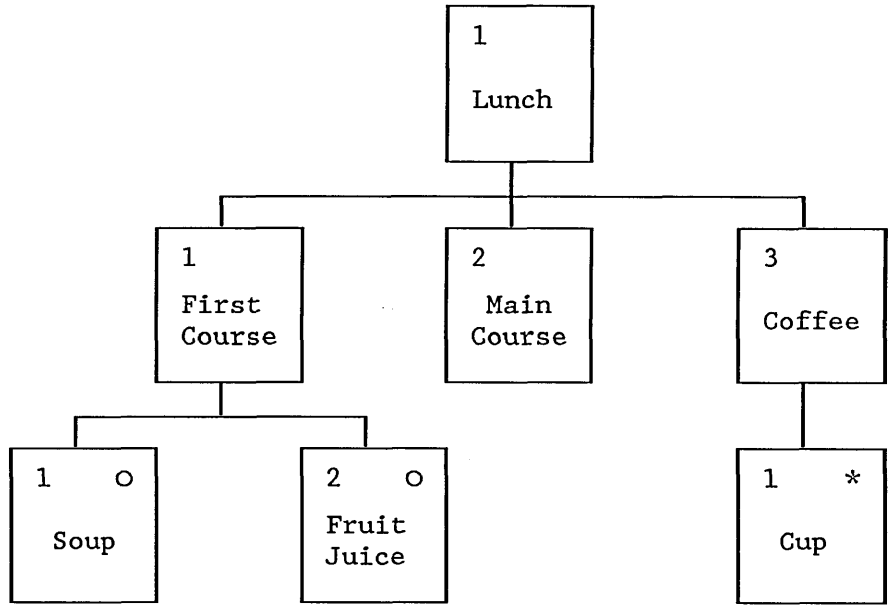


Figure 2.9: The Program Structure for Lunch

CHAPTER 3

An Animated Programming Environment

APE, an acronym for an Animated Programming Environment, is an interactive, graphical, program and text design and development system. The system enables users to develop programs and generate documents whilst working conceptually at the level of Jackson diagrams [52]. Programs can be rapidly created and manipulated using diagraming tools and can then be transformed automatically into source code.

The Jackson structured programming methodology was selected for two main reasons; this methodology has been widely accepted as an established program design technique and, it offers adequate formalism for many of its steps to be automated. Jackson's method of program design makes extensive use of diagrams and the APE system exploits this graphical technique to automatically generate high-level language code in Pascal that can then be compiled.

There have been other graphical system which use diagrams, such as SADT [87], SSA [33], Tse [107], IORL (Input/Output Requirements Language) [28], and PSL/PSA (Program Statement Language and Program Statement Analyser) [101]; but these system do not enable automatic code generation directly from the design diagram. For instance, PSL with its associated PSA divides system functions into sub-functions, precisely specifying the inputs and

outputs of each, and the analyser performs consistency checking between functions. However, PSL/PSA is not completely rigorous or capable of automatic code generation.

On the other hand, there are projects that automate the software development process, such as SAMM [99], and USE [110]; but these are often developed independently of the popular diagraming tools although some attempts of integration have been made [119]. More recently methods such as HOS [68], Kindra [103], and HIPO [98], aim to deliver verifiable code automatically from the design diagram, but there is little evidence that such system have surmounted the problems of presenting a user friendly interface. Other system based on formal methods, such as Z [1] and VDM [53], also suffer from the above problems. The APE system provides a compromise by offering a reasonable amount of formality while presenting the specification to users in a diagrammatic form.

Text based systems [7,102,12], have also been used to generate code automatically from the specification, but such systems have potential problems in that they impose a lexical and syntactic code on the designer which must be learned. Diagrams are generally considered to provide a less error prone method of representing information, although they may hide ambiguities in certain cases. Jackson structure diagrams, however, are just as explicit as text based specifications and therefore make an excellent user interface. For this reason the APE system was specifi-

cally designed around an extended set of Jackson structure diagrams‡.

At present, the APE system can be operated in three different modes, namely, design, programming and text-processing. These different modes of operation are discussed below.

3.1. The Design and Programming Mode

In the APE system, an algorithm is the basic entity for designing software and is represented by one or more tree diagrams (Figure-3.1). In creating these, the designer need not consider the final code at the statement level as the system handles all the control flow aspects of the program. The system incorporates syntax checking and generates data definitions automatically where possible, as a result of which potential sources of errors by the designer are removed.

Algorithms are generated by splitting or refining a problem into simpler components. At the higher levels, problems can be partitioned into logically disjoint sub-problems that can then be tackled independently. Each level of refinement simplifies the sub-problem further until it can be solved directly. This method of hierarchical program development embodies the modern approach of structured programming and top-down design.

---

‡ The extensions made to Jackson structured diagrams are discussed in Chapter 6.

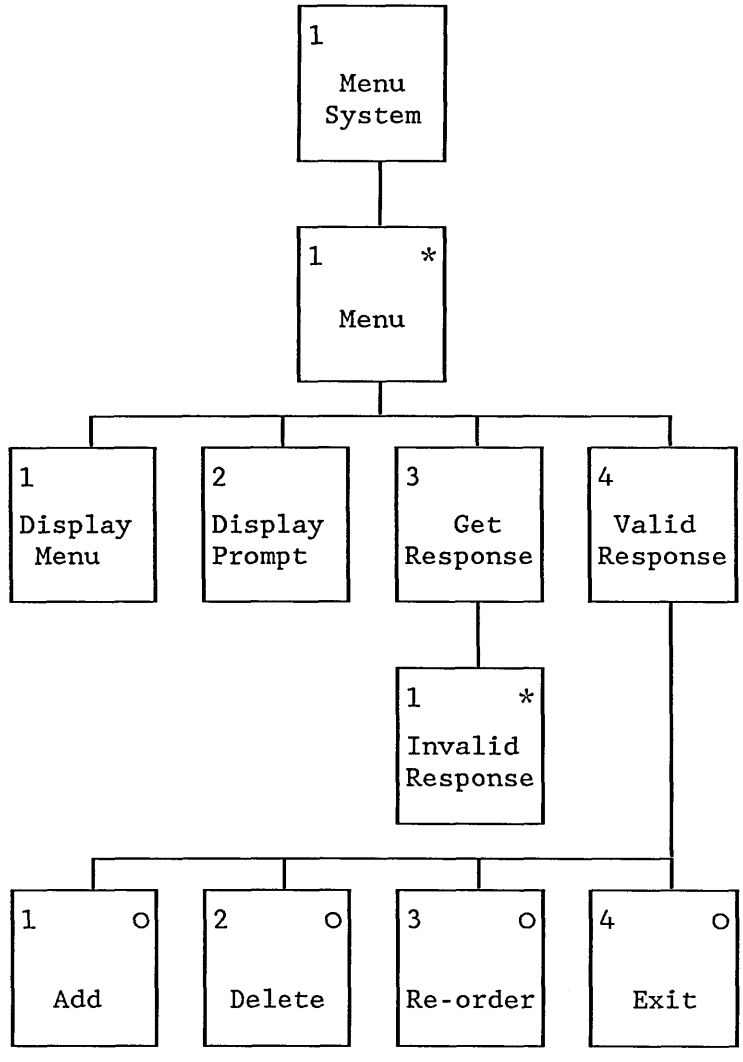


Figure 3.1: A Hierarchical Structure

APE encourages the designer to split the program into components of manageable size. Each subroutine in the program consists of an individual tree. A complete program will normally consist of a single top-level diagram that contains definitions of other subroutine diagrams that are used to create the complete unit. The program generator will automatically traverse this tree of diagrams thus enabling the creation of large programs from simple commands.

APE is designed to be a diagram driven system with a user interface that models the development process and, which will relieve the programmer of many of the tedious and repetitious tasks in designing, writing, documenting and maintaining programs. First the design specification of a program is entered in a *system* tree using comments and underlying control structures. Each leaf of this tree, representing a module (or sub-routine), is then refined to produce code. Thus the complete source code is a refinement of the system tree and contains design information such as the functional specification and the designer's original comments.

Two further tree diagrams are provided during this refinement process. The *coding* tree, Figure-3.1 above, is the default. The shape of the nodes on this tree is determined when the system is initially invoked, and is usually the first node that appears on the screen. All editing operations for manipulating the design diagram are performed on this tree. The second diagram, known as the *skeleton* tree, is a condensed version of the *coding* tree and provides a wider perspective view of the overall structure of the design diagram. This tree can be traversed and is also used for generating a hard copy listing of the design diagram.

Once a module has been defined its interface is available to the other modules that invoke it. This facilitates interface checking. Additionally, a graphical representation of the inter-relations between modules, based on these interface definitions,

is displayed in the form of an *overview* tree. This tree is similar to the *skeleton* tree except for the information and operation that can be performed on it. The *overview* tree displays the current state of the user program. The root node displays the name of the program whilst the siblings display the names of modules. The flags on the nodes of this tree represent the current state of these modules. For example, according to Figure-3.2, the program named 'Menu' is made up of three modules; 'Add', 'Delete', and 'Re-order', of which module 'Delete' has been declared but not defined (since it has a lower-case flag), where as modules 'Add', and 'Re-order' have been coded (indicated by an upper-case M).

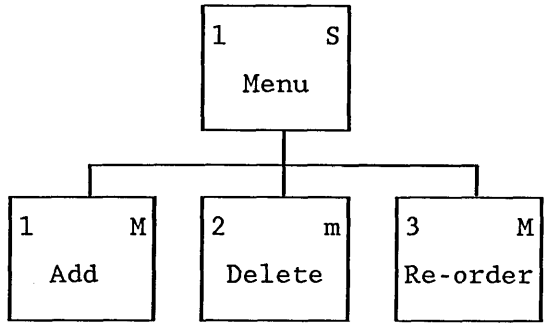


Figure 3.2: An Overview Tree

The *overview* tree is also useful for moving between different modules. As each node in this tree represents the name of a module in the program hierarchy, traversing to a particular node on the *overview* tree and quitting it will place the user on the coding tree that contains the module as its root node. If the module has not been coded, as in module 'Delete' above, a new *code* tree is generated and the user placed in the systems syntax-based editor together with a prompt of the module declaration that is expected.

3.1.1. Software Development Using APE

The software development process in the APE system consists of six main stages:

- system design
- programming
- testing and debugging
- coding
- documentation
- maintenance and modification.

These phases are described in the sub-sections below.

3.1.1.1. System Design

In the *system design* phase the APE system imposes a tree structure on the problem under consideration. This tree is intended to represent the hierarchical relationships between a system and its sub-systems. The system tree is then refined, using the strategy of "divide and conquer", until each leaf finally encompasses a manageable part of the whole problem. The text produced in the nodes of the tree during this phase will consist of comments. However, the text in the leaves of this tree will consist of the formal declaration for a sub-system, or module.

Each module consists of a single tree. The inter-

relationships between modules, based on their interface definitions, is displayed graphically in the form of an *overview* tree (Figure-3.2 above).

3.1.1.2. Programming

The *programming* phase in essence consists of refining the *overview tree* produced at the design stage into a complete program. Each node can be refined in one of three ways; into a sequence of tasks, one of several selections, or a task that has to be iterated zero or more times. A guideline for the rough decomposition of a node in the *overview* tree is that each leaf encompass a simple programming statement or a call to another module, assuming that adequate support is provided by other modules in the system.

Programming statements and data definitions can be inserted by using the system's syntax directed editor. If the input is syntactically incorrect, a one line error message is displayed. The system prompts the user for the type of any undeclared identifiers, thus enabling a user to concentrate on the structure of a program. All declarations are stored in the symbol table and are output when the program is generated. These declarations are not visible in the design diagram.

The generated program is formatted, and comments are automatically inserted. Stubs are generated for each undefined module, thus enabling the program to be compiled and executed.

The code always matches the design diagram perfectly and the comments form a simple means of cross referencing. Experience shows that the code is seldom, if ever, of importance since all work is done at the level of diagrams. There is a strong analogy here with high level languages that are compiled to produce assembler code.

3.1.1.3. Testing and Debugging

The testing of a system may consume half the time and money of the whole development effort [34], and correcting bugs, typically accounts for twenty percent of maintenance [60]. The APE system identifies the specification of testing as a task within program specification. This is done using the knowledge of the structure of the design diagrams.

The APE system provides two methods for *testing* and *debugging* programs. A substantial amount of testing can be done by independently exercising each module in both these methods. In the first method, the source code for the completed modules and the stubs are compiled and linked with the appropriate libraries to form an executable program. Successful compilation leads to execution. The user has the option of using a debugger to trace program execution.

The second option enables a user to view the run-time animation of the design diagram. As a program is a dynamic object, the only way a user can study the details of a program is by seeing it

in operation. Although computer software involves dynamic behaviour, traditionally most of the documentation describing the operation of the software is static. When some software is executed, it usually gives little clue to the internal workings of the software itself. Software animation gives a diagrammatic representation of the changing states of a software system as it is executed. These states can be presented at varying levels of detail, including:

- a graphical representation of the software component
- a code view that shows the execution of code statements and the effect it has on the data.

Animation also provides a powerful debugging tool. Debuggers have undergone significant changes and have kept pace with language development (Figure-3.3). This phase of software development often highlights errors in a program. To help in the task of error correction and software re-use, the APE system provides the programmer with advanced editing tools, such as cut, paste and history, for modifying the design diagram.

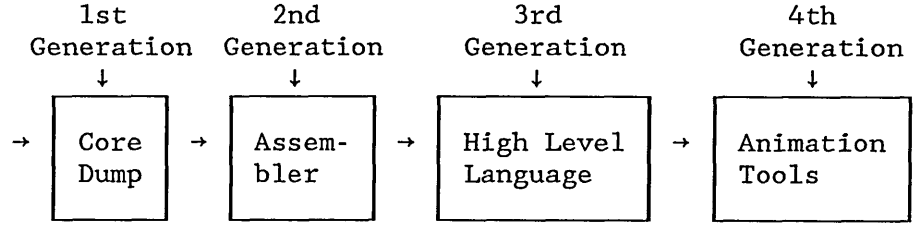


Figure 3.3: Evolution of Debuggers

3.1.1.4. Coding and Documentation

The *system documentation* consists of a complete program listing and a detailed system diagram for each module. APE generates the complete program listing by traversing the trees that correspond to individual nodes on the *overview tree*. Data definitions and comments are automatically inserted in the program and stubs generated for undefined modules. The listing can then be formatted in one of several ways (e.g. highlighted keywords, etc). The detailed design diagram that comprises the design specification and the modules is in the form of a *skeleton* tree.

3.1.1.5. Modification and Maintenance

Since each connection and dependency between modules is maintained and documented by the APE system, no link between modules can be left out of the documentation. *Modification* to the system is achieved by editing the tree diagram. For this the APE system provides advanced editing operations such as cut and paste. Additionally, the modified design diagram is automatically checked for consistency.

3.2. Text Processing Mode

The APE system was originally envisioned as a tool for software program designers and programmers but, considering the hierarchical structured view that APE offered, the system was extended for document processing as well. The extensions made to the system were limited since most of the features were derived

from the programming mode. An overview of the text-processing mode is presented in this section.

In any text processing application two principle tasks are involved in preparing documents, namely:

(1) defining the contents and the structure of a document

(2) generating the document from the specification of its appearance.

The former task generally deals with editing, while the latter is known as formatting and is concerned with the layout of document objects on hard copy media, such as paper, and various soft-copy devices, such as a visual display terminal.

Processing text has long been an important application of computers. Initially editors had been used for this purpose but, a combination of technological and economic conditions have led to more attention being given to formatting systems as well. Because of the increasing costs of manually produced documents, decreasing costs of computers and storage, and the availability of high-quality computer-controlled printers, typesetters, and display devices, the use of computer based formatting systems for a wide variety of technical, business, and literary documents has now become feasible.

The APE system uses an object model of documents [90], in which a document is an object that comprises a hierarchy of more

primitive objects. Each object is an instance of a class that defines the possible constituents and representation of the instance. This model is analogous to the one developed in an object-oriented programming environment. An example of such document classes would include papers for a journal or conference, theses, and programs in a given language; while the lower level classes would include document components such as headers, footers, paragraphs, tables, equations, and fonts. Within this object model framework, the major operations of document processing can be thought of as a mapping from one object to another.

To manipulate this hierarchical object model framework, the APE system provides an extensive set of user commands. All the general utilities of the programming mode, such as tree traversal and history, are available. In addition, there are some tools provided by the system that are specific to this mode of operation.

(1) All nodes in this mode of operation are equivalent, unlike the programming analogue, except that the associated text files may optionally be encrypted for reasons of security.

(2) Each node can accept a file, and the system will remember the file name for subsequent operations. All file manipulations such as rename, editing, and delete are provided by the system.

(3) Files can be stored using the Revision Control System (RCS) [105,106]. This facility is similar to the UNIX Source

Code Control System (SCCS) [3,86]. The APE system provides the users with an interface to all the commands in RCS, thus providing a powerful tool for storing, retrieving and logging revisions of text.

(4) Formatting is done automatically by the system. This is achieved by using a software tool called *doctype* [45]. The user can either invoke the formatter on the file associated with a node, or on the files associated with a complete sub-tree.

The formatter keeps track of the document processing tools available on the UNIX / ULTRIX‡ system and automatically invokes the appropriate pre- and post-processors which are needed to print any particular document.

‡ ULTRIX and VAX are a trademark of Digital Equipment Corporation.

CHAPTER 4

The APE Environment

The APE system provides an extensive set of user commands for manipulating the design diagram [46]. It includes features such as traversal and editing facilities for the tree diagram, and a history mechanism that permits the storage and retrieval of several tree structures. Although the system is interactive and incorporates an optional on-screen menu facility, it is not hardware specific and has been designed to be used with any cursor control terminal, such as a VT100.

The APE environment consists of nine components, each of which is described below. A more detailed discussion on each of these modules is presented in Heerjee [47].

4.1. General Manager

The General Manager controls all system activities. It initialises system processes, menu routines and error checking procedures. The General Manager also receives user input from an input device, such as a keyboard, which is analysed and then transmitted to the appropriate process. The decoding of commands and menu selection messages also takes place here. In addition, the General Manager accesses graphics procedures from a terminal capability database. This enables the system to interface with the terminal primitives.

The General Manager also divides the screen into three windows; the menu, text and node windows. The screen design is discussed in the Chapter 5.

4.2. Screen Manager

The Screen Manager handles the user interface of the system. It accepts input from an input device and displays the textual and graphical response from the system. The Screen Manager is responsible for drawing the various tree diagrams available in the system. The enlarged diagram is the default. Another diagram in the system provides a perspective view of the design diagram by displaying more levels/nodes on the screen. In the programming mode of operation, the Screen Manager also provides an option for viewing the overall structure of a program in the form of an overview tree.

The Screen Manager provides facilities for traversing the various tree diagrams. The simplest way is by moving from one node to another, either by using the four cursor keys, the four *vi*† keys, or using the systems pull down menu facilities. Options are also provided for faster traversal of the tree diagrams. The *Page-Up* and *Page-Down* options will move the user up or down a screen-full, while the *goto* option is useful for moving to a specific node on the current tree.

† Vi is the UNIX operating systems screen oriented text editor.

4.3. *History Mechanism*

The History Mechanism is a database built in the APE system and is used for the storage and retrieval of tree structures. All new trees are generated here. Similarly, a tree can be deleted from the APE system only through this tool. The History Mechanism is also useful for moving between various trees already stored in this database.

In the text processing mode of the APE system, the History Mechanism is interfaced with the Revision Control System [105], for storing, retrieving and logging revisions of text.

4.4. *Syntax Directed Editor/Interpreter*

The APE systems syntax directed editor/interpreter is an interactive tool for the Pascal source language. The editor interprets statements, a line at a time, and produces an error message when a syntax error occurs. The system prompts the user for any undeclared identifiers and generates a symbol table which is output when a program is generated.

The editor is built using two UNIX tools, YACC and LEX [48]. Presently the editor handles Pascal source alone, however, this can easily be extended to handle other block structured languages such as C and Ada.

4.5. *Code Generator*

The Code Generator generates code in the Pascal language by

accessing the information held in trees of the History Mechanism. To do this, the Code Generator builds up an internal representation of the overview tree. It then traverses the various tree diagrams held in the History Mechanism, that are represented by nodes on the overview tree, and outputs the final code. Comments are automatically inserted and stubs generated for any incomplete module.

4.6. Program Animator

The Program Animator is used for generating the information necessary for animating a program.

The simplest approach to developing animated graphics for a program representation is by including a 'graphics library' in the program and to insert calls to image drawing routines whenever the program changes in an interesting way. This however has a major drawback in that it makes the programmer write two programs in parallel, and there is no guarantee that the two will complement each other.

Another approach to developing animated graphics for a program is by inserting 'hooks' into the low-level run-time software. In this way, procedures that display modifications of the program state as graphical 'side-effects' are then attached so that the progress of a program can then be monitored without any modifications to the program itself.

A third approach is to use a separate, independent monitoring process that extracts the desired information form the program when it is in operation.

The animation component of the APE system, which is part of the programming mode, is based on a combination of the second and third approaches. To animate the design diagram, the user does not have to change the original program. Instead the Program Animator makes the necessary changes to the code, produced by the Code Generator, by inserting the appropriate 'hooks' and extracting the desired information from the APE systems run-time library. The 'hooks' consist of the necessary information that is required by the Screen Manager to re-build the design diagram, such as regenerating the APE tree. Features such as diagram traversal and moving between windows are provided by linking in the appropriate APE system routines.

In animating the design diagram, the user traces the execution of the program. Breakpoints can be set and values of variables traced as in a conventional debugger. All input and output from the program occurs in a separate window at the top of the screen. During animation, the cursor traverses the tree diagram following the node that is currently being executed.

4.7. *Document Processor*

The Document Processor [45] is used in the text processing mode of the APE system. The text processing environment in the APE

system provides support for UNIX dependent text processing facilities (Figure-4.1). It includes facilities for document structuring and the automatic invocation of these text processing facilities [45].

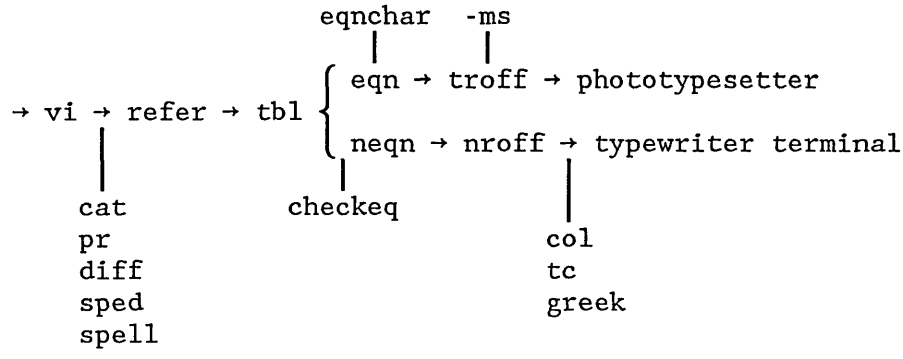


Figure 4.1: UNIX Document Processing Tools

4.8. On-line Help

The on-line help option provides either a single screen-full of information or a complete on-line manual. The *aide-memoir* contains a guide to some of the useful commands available to the user for a particular mode of operation. The manual follows the style of the UNIX programmer's manual [104], in which natural language text is used to give a precise yet easily understandable description of the system.

4.9. Learn Package

The Learn Package gives Computer Aided Instruction courses and practice to help the novice user operate the system. The Learn Package is interactive and is built as a separate module in the APE system. This module is interfaced with the Screen Manager

which does the screen drawing routines.

CHAPTER 5

Interface Design of the APE System

A major goal of software engineering is enhanced programmer productivity. This is partially achieved by using tools whose data access is integrated and coordinated by the centralised use of a database. However, a major consideration in the design of an effective software engineering environment must be the suitability and power of its human-computer interface; it is estimated that for over 95 percent of the human/machine interactions, people costs are greater than machine costs [8].

Increasing effort is now being devoted to the development of interactive systems, but their positive impact is seldom validated. Both Nickerson [77] and more recently Shneiderman [93], have raised concerns about the paucity of research in this area. Shneiderman argues that intuition and limited experience that may have been adequate for designing earlier applications might not be appropriate for future interactive systems that will have many diverse users. To overcome this, several authors, such as Hansen [37], Wasserman [109], Foley [31], Norman [78], Benest [9], Edwards [26], Lund [65], and Poulson [82], have put forward lists of goals to assist designers in the design of interactive systems. However these lists can often be criticised for being contradictory or too imprecisely stated to be defined and measured. Furthermore, it simply is not known whether these various design

approaches will work for different user types, problem types, organisational environments, etc.

As interactive tools become more prevalent, their effects on productivity need to be assessed. There are several questions in this area that can be posed, including:

(1) Do interactive tools improve productivity in a quantifiable way?

(2) Does the complexity of the interface affect programmer productivity?

(3) Does the quality of the interface influence the quality of the system that is finally produced?

The APE system was evaluated in order to investigate the claim that interactive tools can increase productivity and reduce frustration over traditional development techniques. Human factor techniques were used throughout the design and implementation stages of the development life-cycle. The design of the human-computer interface of the APE system, based on human factor techniques, is discussed in the remainder of this chapter. The method and results, based on responses to a questionnaire and a comparative evaluation, are presented in Chapter 8.

5.1. Design Considerations

Clearly an interactive system should not be considered good solely by virtue of being interactive. The true test of the value

of the system lies in whether the users are satisfied with it. In most interactive systems, the screen displays are a critical component of successful designs. Well-designed and carefully prepared screens can maximise user productivity, eliminate or reduce input errors, and promote user satisfaction.

In the development of the APE system, considerable effort was spent on designing the interface that would be as 'user friendly' as possible. During the early stages, preliminary design decisions were based on available human factors information regarding the interface. Information from users of similar systems was gathered and a baseline proposal generated. These guidelines included:

- the number of nodes to be displayed on the screen
- size of the default node
- use of multiple windows
- use of inverse video and the possible use of colour
- character sets
- use of a cursor control terminals
- use of audible sounds and other devices such as a mouse
- the format of data entry for items
- the command syntax and sequence

- on-line help
- learn and reference packages.

In the early design process, and subsequent implementation, user involvement was limited. However, pilot tests were carried out on all modules of the system during the development and iterative testing process. This form of testing is rapid and inexpensive, and often generates productive information. Additionally reports, and any new suggestions or ideas, were submitted each month during the implementation stage of the system.

The benefit of using these human factor techniques throughout the design process was demonstrated by the generally favourable responses given by the users in the Questionnaire evaluation (discussed in detail in Chapter 8). The characteristics of the APE system's human-computer interface are now described below.

5.1.1. Screen Design

For most interactive systems, the layout of information on a screen is important, since densely packed or cluttered screens can often overwhelm and distract users. Smith and Mosier [96], highlight the complexity of this issue by suggesting 162 guidelines for displaying data on screens. Screen designs will inevitably involve some individuality and innovation. The hopes of an expert system that will do screen layouts seem remote as the demands of each task and user groups are varied and difficult to measure.

An early design decision was that the APE system should be hardware independent, and capable of running on any terminal that supports direct cursor positioning. To account for the limited screen size several options were available, including organising the display in pages, and the use of windows. It was decided to make sequences of screens similar throughout the system for similar tasks. Within a sequence, the information on the screen gives an indication to the user of their current position. The option of going backward in a sequence is also provided as this helps in trying alternatives. System messages provide guidance and employs a user-centered phrasing [118]. Furthermore, all system messages are displayed in a consistent format at a fixed location on the screen. The opportunity of using multiple overlapped windows was limited because of the size of the screen and the time it would take to update it.

The APE system creates three windows when it is invoked;

- menu window
- text window
- node window.

A diagrammatic view of the overall terminal screen is shown in figure-5.1.

The menu window is a single line window at the top of the screen and highlights the menu bar options, in inverse video.

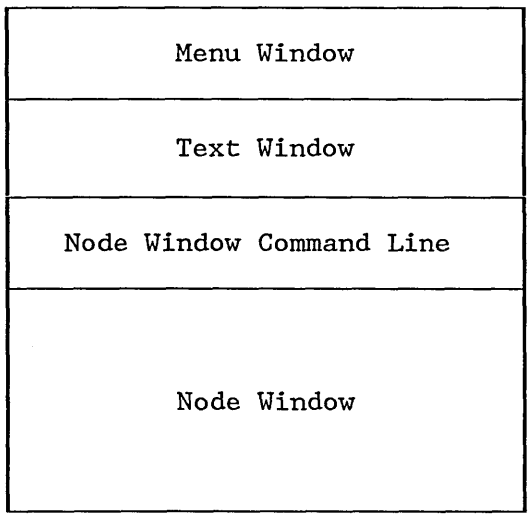


Figure 5.1: Diagrammatic View of the Screen Layout

The coding tree is displayed in the **node** window. The maximum size of this window is terminal dependent. The size of the nodes on this tree, together with that of the window in which it is displayed, can all be varied to a certain degree by the user on invoking the system. The first row of this window is known as the **command** line, as all interaction with the system, except with the menu window, occurs here. All system prompts and messages are generated on the command line as well.

The **text** window is made up of the remaining portion of the screen that consists of the **menu** window at the top, and the node window below. The text window is the only window in the system where scrolling occurs. All input into a node takes place here, as the system does not provide any direct access to the internals of a node except through this window.

In addition to these three windows, the system also uses

other windows when specific tasks are to be performed.

5.1.2. Dialogue Design

In designing the command set several issues were considered. Macaulay and Norman [67] offer eight high-level objectives for designing the dialogue in an interactive system. These include:

(1) *Functional Simplicity*: The complexity of the interface language is an important element in dialogue design [25,21]. High complexity refers to a large number of options available to the user at any one level in the computer system, and low complexity is represented by few response options available to the user at any one level but several levels are present. The total number of response options may be the same, it is only their representation to the user that differs. Therefore, for both casual users and inexperienced programmers 'functional simplicity' is recommended, where the alternatives presented to the user are either limited or built up from a small number of primitive elements.

(2) *Command Complexity*: Functional Simplicity implies that commands have a restrictive function. Conversely, commands may have different functions depending on the types of flags, options or default values that it may take. However, Benbasat *et al* [8] take the view that complexity is increased by adding commands rather than by increasing the range of functions within the commands.

(3) *Response Flexibility*: The flexibility of response offered to users by interactive systems, with either a fixed form of response – such as the constrained choice in system controlled dialogues – or a free response – such as the ones offered by natural language systems – is considered to be an important issue in dialogue design [70,30].

(4) *Language Equivalence*: This is derived from the theoretical separation of language and dialogue [67]. The dialogue in most interactive systems contains a syntax that is unique to that system. This issue of equivalence and non-equivalence of syntax for systems and for users requires further investigation [67].

(5) *Dialogue Control*: Shneiderman [92], Miller [70] and Benbasat *et al* [8] argue the importance of dialogue control by either a user or system in designing the dialogue structure.

(6) *Dialogue Format*: There are certain physical differences in screen layout for different dialogue representations such as menu selection and instruction response [67].

(7) *Task and Control Dialogues*: Even though increasing attention is being paid to improving the user interface, in many cases a human-computer dialogue is not directed towards the interactive task itself. There is a need for supplementary materials because of the confusion and errors that will always be generated on the part of a user. To this end, many

dialogue designers are enticed by the notion of on-line help facilities and tutorials that use the same interactive system.

(8) *Timing*: The time it takes for a system to display information in response to a user selection have produced many conflicting findings. This is mainly due to the costs, task complexity, speed of task performance, error rate and error handling procedures of different systems. These issues are further complicated by the impact of personality differences, time of day, fatigue, familiarity with computers, experience with the task, and motivation [18,91].

Further to these issues, Shneiderman [92] lists eight design principles for interactive systems. These include:

(1) Strive for consistency.

(2) Enable frequent users to use shortcuts.

(3) Offer informative feedback.

(4) Design dialogues to yield closure.

(5) Offer simple error handling.

(6) Permit easy reversal of actions.

(7) Support internal locus control.

(8) Reduce short-term memory load.

These principles were used in designing the command set for the APE system.

In the APE system, the primary input device for manipulating the interface is the alphanumeric keyboard; occasionally the cursor control keys are used. The commands are usually a single keystroke. For example, the 'S' key is used for generating sequence nodes, while the cursor keys are used for traversing the tree diagram.

The main interactive styles used by designers of an interactive system include,

(1) menu selection

(2) natural language

(3) command language

(4) direct manipulation

(5) form fill-in.

Each of the above interactive styles have some advantages and disadvantages. For example, the advantages of having a menu driven system includes:

- shortening the learning time
- reducing the number of keystrokes

- permitting the use of a dialogue management tool.

The shortcomings of such a system include:

- the danger of having many menus; which may slow down experienced users
- consume valuable screen space
- require a rapid display rate.

Most interactive systems incorporate at least two of the above five interaction styles in their interface. A combination of two, or more, styles in an interface would be more effective, when the required task and users are diverse, as the disadvantages of one would be compensated by the advantages of the other. Command language and direct manipulation were the two interaction styles that were initially used in implementing the APE system. Both these styles of interaction have distinct merits and shortcomings and are discussed below.

(1) *Direct Manipulation*: This style of interaction is appealing to novices, easy to remember for intermittent uses, and with some careful design, it can be rapid for experienced users. The central ideas in this style of interaction is that actions are rapid, incremental, and often performed with physical actions - such as cursor motion devices - instead of complex syntactic forms. However, this style of interaction does not guarantee the success of a system, because a poor

design, slow implementation, or inadequate functionality can undermine acceptance.

The testing of this style of interaction is important at an early stage of the design process because this approach can easily lead to problems for the designers and users. Examples of systems that use this style of interaction would include visual display editors, video games, and the Apple Macintosh MacPaint program.

(2) *Command Languages*: Command languages originated with operating system commands. The distinguishing features of this style of interaction is in the immediacy and the impact that they have on a device or information. The commands are brief and their effect transitory.

On learning the syntax, command languages offer greater flexibility, are convenient for creating user defined macros and support user initiative. This form of interaction is appealing to "power" users [92]. A disadvantage of this style is the amount of memorisation required, because users must recall the notation before any action can be initiated. Furthermore, the error rates are high, and substantial training and retention is required. Users of this style of interaction "are often called on to accomplish remarkable feats of memorisation and typing" [92]. For example to obtain the names of users on the UNIX system, the following command may need to be typed:

```
awk 'BEGIN {FS=":"} {print $5}' /etc/passwd > users
```

Each command in the APE command language carries out a single task. The sum total of the commands equals the number of tasks. Initially, the APE system performed a limited number of tasks and this approach proved adequate. However, as the system grew larger the command set became more extensive and complex. This resulted in a more complex strategy employing single letters, shifted single letters, and 'CTRL' key plus single letters. Additionally, some commands stood alone, whereas others had to be combined in different patterns. An example of the many commands offered by the system is shown in Figure-5.2†. This profusion of commands enabled the expert users to achieve a complex task with very few keystrokes. However, the large command repertoire made the task of the novice and intermittent user difficult. Therefore, it was decided to incorporate a menu interface into the system.

5.1.2.1. Menu Design

The primary goal of menu designers is to create a sensible, understandable, and distinctive semantic organisation of the selections available to the user. A problem in designing menu systems, for character based terminals in particular, is caused by limited screen size. This shortcoming can be overcome by organising the items within a menu. Since the APE system has an extensive set of commands, the meaningful organisation of menu items is

† The complete list is presented in the APE programmer's manual [46].

APE Command Language Commands

Traversing Commands

k	Move Up
j	Move Down
h	Move Left
l	Move Right
CTRL-F	Page Forward
CTRL-B	Page Back

Manipulating the Design Diagram

s	Create Sequence Nodes
c	Create Selection Nodes
w	Create a WHILE Loop
f	Create a FOR Loop
r	Create a REPEAT Loop

Match a string

/pat	Go to node with pattern forward

Invoking the Menu Options

ESC-A	Expose the APE menu option
ESC-H	Expose the History menu
ESC-M	Expose the Help (or manual) menu
ESC-T	Expose the Traverse keys menu
ESC-E	Expose the Edit menu

Figure 5.2: *APE Command Language Commands*

essential. The usefulness of organising menu items has been demonstrated by Leibelt *et al* [61].

There are several types of menu systems. Some of the well known ones are listed below:

(1) Single Menus: These include:

- binary menus
- multiple item menus
- extended menus
- pull down menus
- permanent menus
- multiple selection menus.

(2) Linear Sequence Menus

(3) Tree Structured Menus

(4) Acyclic and Cyclic Menus

The APE system makes use of the **Single**, and **Tree Structured** menu systems. In the former case, a combination of pull down, or pop-up, and extended menus is used. The style is similar to that of an Apple Macintosh. Selections can be made either by moving the pointing device over the menu items, or by typing in the required escape sequence at the keyboard. The contents of the pull down menu are then displayed. Since the pull down menu covers a portion of the screen, it is necessary to limit the text to a minimum. This is achieved by creating a hierarchy of menus. Common choices are displayed at the top level with additional items that lead to the next screen in the extended menu sequence. At the top level, some collections are classified into mutually exclusive groups with distinctive identifiers. For example, the commands for traversing the tree diagram are placed under the option

'Traverse', whilst the manual pages can be obtained by invoking the 'Manual' option. However, indexing and classification are complex tasks and often there is no general solution that is acceptable to everyone. Despite these problems, tree structured menus enable a large collection of data to be made available to a novice or intermittent user.

Several studies [69,56], have commented on the tradeoffs between the depth and breadth of menus, suggesting four to eight items per menu and no more than three to four levels. However, with large menu applications, such as the APE system, one or more of these guidelines must be compromised.

Tree structured menus are used within the APE system with the on-line help facility and the learn tutorial. In designing the menu interface under this category, it was decided to group logically similar items together, and ensure that items in a menu would not overlap with items in another. It was also decided to display an index on the screen as this would enable the user to maintain a sense of position. Finally, informal tests were carried out on sample users before a terminology was chosen thus ensuring that items in the menus were distinct from one another.

CHAPTER 6

Implementation

The APE system is written in C [55] and has been implemented on a VAX-11/750 under the UNIX/ULTRIX operating system. Versions of APE have also been successfully ported to run on Micro-VAX work-stations, a SUN work-station, IBM compatible personal computers, and under the VMS operating system.

The APE system uses the functions in the Curses [5] screen package for driving its human-computer interface. The curses package is a collection of terminal independent, low-level input/output functions. These subroutines enable a program to do most of the common type of terminal dependent functions, those of motion optimisation and optimal screen updating, by decoding the information in the termcap† terminal database. An overview of the Curses library package is presented in Appendix A. In the remainder of this chapter, the extensions made to Jackson structured diagrams are described, following that some implementation details of the system are presented.

6.1. Extensions to Jackson Structure Diagrams

An overview of Jackson's method of program design was presented in Chapter 2. In the sub-sections below, the extensions made to this methodology which enable program generation are

† *Termcap* is the UNIX terminal capabilities data base.

discussed.

6.1.1. Loop Types

In Jackson structure diagrams the looping structure only serves to show the requirement of a simple iteration in the algorithm. No indication of the type of the loop or the range is given. For code production, however, some method of deciding the type and range is required.

There are three basic loop types that cater for most problem requirements. These loops are language independent in that they can be implemented easily if the language does not contain them explicitly. The three loop types contained in most modern high level programming languages, including Pascal, are:

(1) **While**: The While structure caters for loops that must be iterated zero or more times until some logical expression becomes false. The logical expression must be shown to enable code generation.

(2) **Repeat-Until**: This structure caters for loops that must be iterated one or more times until some logical expression becomes true. Again this logical expression must be shown.

(3) **For**: The For structure caters for loops that are to be iterated a predetermined number of times. The loop counter and the limits of its range must be shown.

These looping constructs have been implemented in the APE system by minor alteration to Jackson structure diagrams. When a looping construct is selected, two nodes are produced. The top node represents the head of the loop whilst the bottom one represents the repetitive body.

The head node contains a flag, one of 'W', 'R' or 'F', to show the loop type. The terminating case, or the iteration range, of the loop is inserted in this node. The body of the loop is shown by an asterisk, '*', as in normal Jackson diagrams. During code production the contents of the head node are inserted into the control structure determined by the loop's type flag. For example, the *while* loop, as specified by the structure in Figure-6.1, would produced the Pascal code as shown in Figure-6.2.

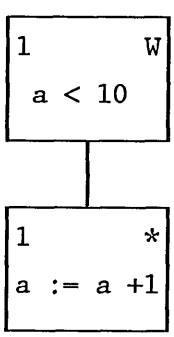


Figure 6.1: Construct of a WHILE Loop in APE

```
while (a < 10) do
begin
    a := a +1
end;
```

Figure 6.2: Implementation of a WHILE loop in APE

6.1.2. Selection Types

The Jackson diagram for a selection construct only suggests the possible paths taken at the selection. It does not show how the decision about which path to take is made. For code generation a method is required for indicating the predicates to make the selection.

In APE, the conventional Jackson diagram for a selection construct is extended to show the predicates. The selection nodes are shown with the normal flag, 'O', but two further nodes are introduced below this node.

(1) The eldest sibling, flagged with a 'P', contains the predicate to select that option.

(2) The second sibling contains the statements that will be executed if that option was chosen.

The graphical representation for a selection construct in the APE system is shown in Figure-6.3.

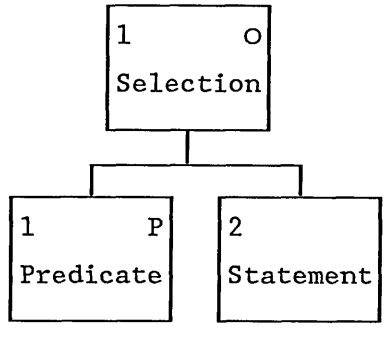


Figure 6.3: A Selection Construct in APE

The select sub-problem represented by Figure-6.4, would ideally be implemented by a SELECT statement with the format as shown in Figure-6.5. The statement corresponding to the unique true predicate would be executed, then control would pass to the statement following the END SELECT. The implementation represents exactly the structure described by the diagram. However, this representation is not provided by any programming language and had to be implemented using available features. The implementation that was initially chosen is shown in Figure-6.6. The 'else' clause was generated by the system.

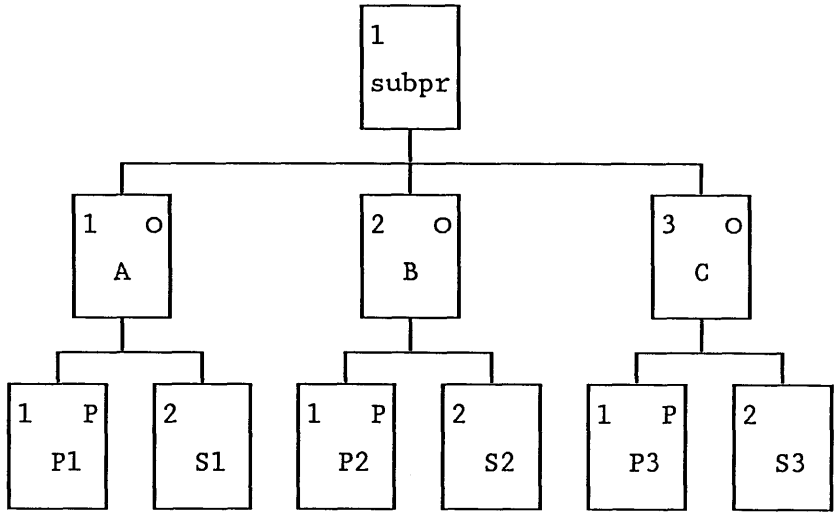


Figure 6.4: A Sub-problem Representation

```
SELECT
    p1 ⇒ S1;
    p2 ⇒ S2;
    p3 ⇒ S3;
END SELECT;
```

Figure 6.5: Implementation of a Selection

The 'if-then-else' construct was preferred over the 'case'

```
{ SELECT }
if p1 then
    begin
        S1;
    end
else if p2 then
    begin
        S2;
    end
else if p3 then
    begin
        S3
    end
else
    ERROR;
{ END SELECT }
```

Figure 6.6: Initial Implementation of a Selection in APE

statement mainly because the latter had two major shortcomings. These were:

(1) strings cannot be easily tested

(2) they are implementation dependent, as different compilers handle the default case differently.

The implementation, as shown in Figure-6.6, had one disadvantage. Consider the example as shown in Figure-6.7. If the above implementation were applied to this figure, the 'ERROR' branch would be executed whenever the tests for 'a > b' and 'c ≤ d' failed. To overcome this, a further refinement of the above implementation was necessary.

In the extended implementation an 'else' node, signified by an 'E' flag, was provided. The 'else' node can only be constructed as a sibling of the last selection ('O') node. Furthermore, this

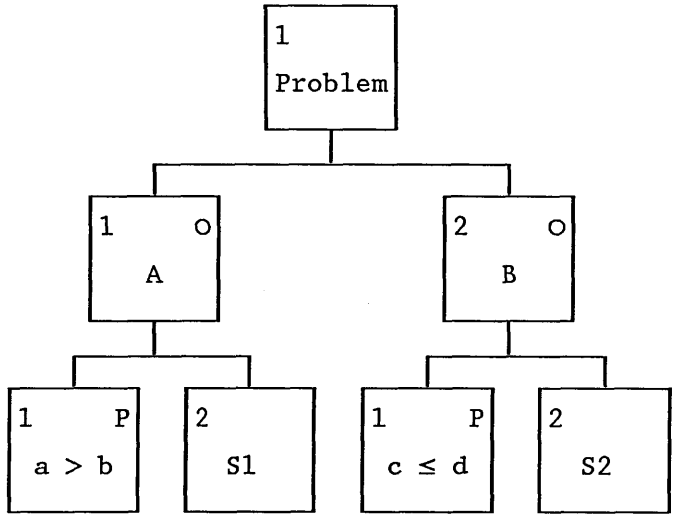


Figure 6.7: A Selection Without the Else Clause

node can occur only if there is more than one selection node. Figure-6.8 is the new implementation for Figure-6.7. Figure-6.9 is a diagrammatic representation of the selection construct with an 'else' clause, and Figure-6.10 is its implementation in the APE system.

```
{ SELECT }
if (a > b) then
begin
    S1;
end
else if (c ≤ d) then
begin
    S2;
end;
{ END SELECT }
```

Figure 6.8: Selection Construct Implementation in APE

6.2. Internal Representation

A tree in the APE system is composed of nodes connected by branches that occur singly, or in sets, and whose members may be

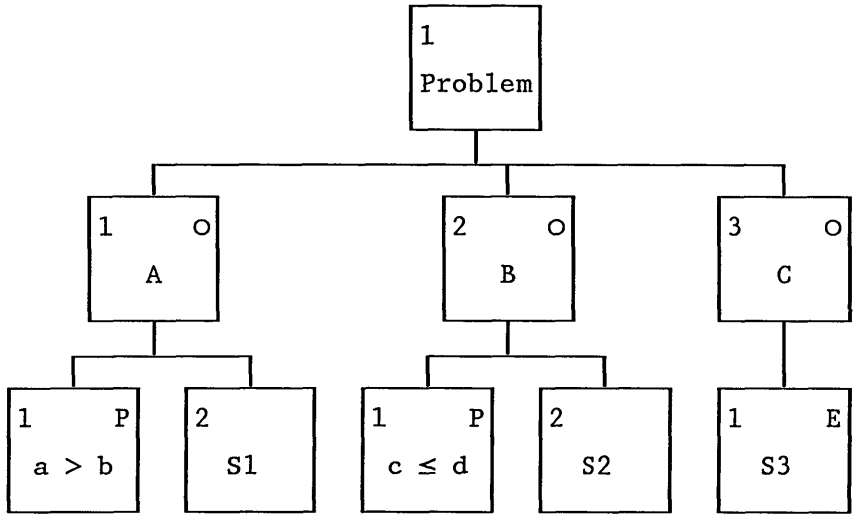


Figure 6.9: A Selection with an Else Clause

```
{ SELECT }
if (a > b) then
begin
    S1;
end
else if (c ≤ d) then
begin
    S2;
end
else
begin
    S3;
end;
{ END SELECT }
```

Figure 6.10: Implementation of a Selection in APE

sequential or selected, but not both. This contributes to the simplicity of the tree structure because it ensures mutual exclusion in selections. Nodes may be iterated, and following the syntax of Jackson's method of program design, iterative nodes may belong to sequences alone.

Programs in the APE system are based on the approach of decomposition in a hierarchical way, such as functional

decomposition or step-wise refinement. Therefore, the only information that needs to be associated with branches of a tree that connects pair of (father-son) nodes is the identity of the pair of nodes being linked. This information is carried in the data structure about nodes, thus removing the need of a separate record type for branches.

Tree drawing is an important function within the APE system. Nodes are placed on a tree for printing by calculating the co-ordinates of the top left hand corner of nodes as points in x-y space. Hence, in the input, each node specifies its parent. The system then derives the vertical position of this node from that of its parent. The tree drawing routine within the APE system can process unlimited sizes and shapes of trees, the only restriction being imposed by the hardware.

6.2.1. Linking Nodes Together

The natural construction for a node of a tree would have had pointers to all its siblings. This could have been implemented in one of two possible ways; either as an array, or a linked list of pointers. The former representation would impose constraints on the number of siblings possible, while the latter would result in a very complex system. A suitable compromise is the *Eldest-Son-Younger-Brother* style of tree. In such a system, a node of the tree has references to its next younger brother and its eldest son, as shown in Figure-6.11. To access a sibling of a node, reference is first made to the eldest son. The chain of younger

brothers is then traversed to reach the required node.

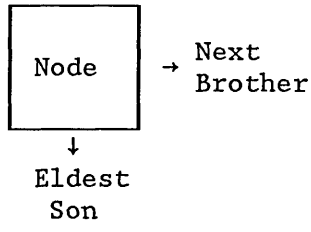


Figure 6.11: Eldest Son-Young Brother

For ease of traversal, two further pointers were introduced. Each node was given a pointer to its parent and another pointer to its older brother (Figure-6.12).

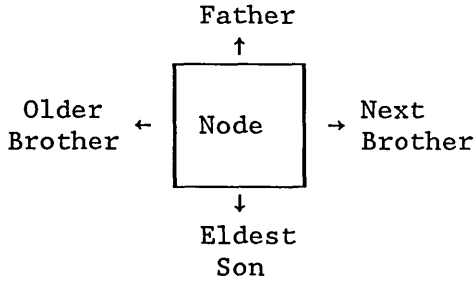


Figure 6.12: Internal Representation of a Node

The underlying data structure that stores information about each node on the tree diagram is called a *_TREE* and is defined in Figure-6.13.

The variables *head* and *_root* return pointers to the *_TREE* structure. The former is a pointer to the active node in the tree, while the latter references the root node of the tree.

Each node of the tree is assigned a unique generation number (field *gen_number*) with respect to its brothers. The number of

```
# define  _HIDE          1  /* mask sub-tree                  */
# define  _ONSCREEN      2  /* display node on screen         */
# define  _HISTORY       4  /* tree saved in history          */
# define  _TRACE         8  /* trace contents of node         */
# define  _BREAKPOINT   16  /* set breakpoint                 */

typedef struct _TREE        /* _TREE data structure           */
{
    int             options;       /* options                 */
    char            flag;          /* node type               */
    char           *text;          /* text in node            */
    char           *file;          /* file name               */
    short           gen_number;    /* generation number       */
    short           no_of_sons;    /* no. of siblings         */
    union {
      _LOCS         _coord;        /* x, y co-ordinates       */
      struct _TREE *_treeptr;      /* pointer to root node    */
    } u;
    _SPEC           box;           /* node specification      */
    _DDEFS         *_Defs;         /* declarations            */
    struct _TREE   *father;        /* the parent              */
    struct _TREE   *son;           /* the first sibling       */
    struct _TREE   *youngbrother;  /* node to the right       */
    struct _TREE   *oldbrother;    /* node to the left        */
} _TREE ;

_TREE *_root;  /* pointer to the root of a tree        */
_TREE *head;   /* pointer to the active node in a tree */
```

Figure 6.13: The C Data Structure for a _TREE

siblings under a node is held in the entry *no_of_sons* and the textual contents is referenced by the character pointer **text*.

The *flag* field is a single character that specifies the type of a node. In the programming mode, nodes can take a variety of flags, each of which is based on the extended set of Jackson structure diagrams. In the text-processing mode all nodes are equivalent, except that the associated text files may optionally be encrypted in which case the system generates a special flag.

The *options* entry is an integer expression formed by combining one or more of the '# define' constants. Options are set by using the bitwise operators. The meaning of these manifest constants is as follows:

_HIDE: Used by the *skeleton* tree generator. If set, the sub-tree below the node is not displayed on the screen.

_ONSCREEN: This constant is set for nodes of the *coding* tree that are to appear on the *node* window.

_HISTORY: Used by the system to check whether a tree has been saved in the *history* database. This constant is set in the root node of the tree.

_TRACE: Used with the animation option in the *programming* mode (Further details on using this option are discussed in the animate(1) entry of the APE Programmer's manual). If set, the user specified values are output on a separate window whilst animating the design diagram.

_BREAKPOINT: Used with the animation option in the *programming* mode. If this constant is set, the debugger is invoked whilst animating the design diagram.

The chief purpose of using manifest constants is storage conservation. Using constants (as above) and bitwise operators, five (or

more) variables can be stored in a single integer, as opposed to using five (or more) integers.

If the _ONSCREEN constant is set for a node, i.e. the expression 'head→options & _ONSCREEN' is true, the entry *_coord* is used for assigning values that determine the location of the node on the screen. The field *_coord* is of type *_LOCS* and is defined in Figure-6.14.

```
/* The data structure that stores the x, y
 * co-ordinates of a node on the screen.
 */
typedef struct _LOCS
{
    short  _x;  /* the x co-ordinate */
    short  _y;  /* the y co-ordinate */
} _LOCS;
```

Figure 6.14: The _LOCS C Data Structure

Each node of the *overview* tree has a field, **_treeptr*, which references the root node of the corresponding *coding* tree that this node represents.

In the programming mode of operation, the character pointer, **file*, contains debugging information that is used by the system if the animation option is invoked. In the text-processing mode, this field contains the name of the file associated with the node.

The entries, *_Defs* and *box* are specific to the programming mode of operation. The data definitions local to a tree are stored at the root node in a structure called *_DDEFS* (shown in Figure-6.15).

```
typedef struct _DDEFS /* _DDEFS data structure   */
{
    char *_Dname ;    /* module name              */
    char *_Dtitle;    /* the type of a module     */
    char *_Dlabel;    /* label declarations       */
    char *_Dconst;    /* constants declarations   */
    char *_Dtype ;    /* type declarations        */
    char *_Dvar  ;    /* variable declarations    */
    char *_Dproc ;    /* subroutine declarations  */
} _DDEFS;
```

Figure 6.15: The _DDEFS C Data Structure

The field *box* (defined in Figure-6.16) is of type *_SPEC* and contains the specification associated with a node. Any information held in the *_SPEC* data structure is unused at present and is reserved for a future implementation.

```
typedef struct _SPEC      /* specification of a node    */
{
    char *input;          /* inputs to a node           */
    char *output;         /* outputs from a node        */
    char *comment;        /* comments in a node         */
    char *operations;     /* operations to be performed */
    char *specification; /* constraints for a node      */
} _SPEC;
```

Figure 6.16: The _SPEC C Data Structure

Finally, each node is linked to its parent (by **father*). A parent is linked to its eldest son (by **son*); siblings are linked to the brother nodes on the right and left (by **youngbrother* and **oldbrother* respectively), except for the rightmost and leftmost nodes that have NULL pointer fields. Figure-6.17 shows the references to individual structures that are always through C pointers. These linkages result in a multiway linked list.

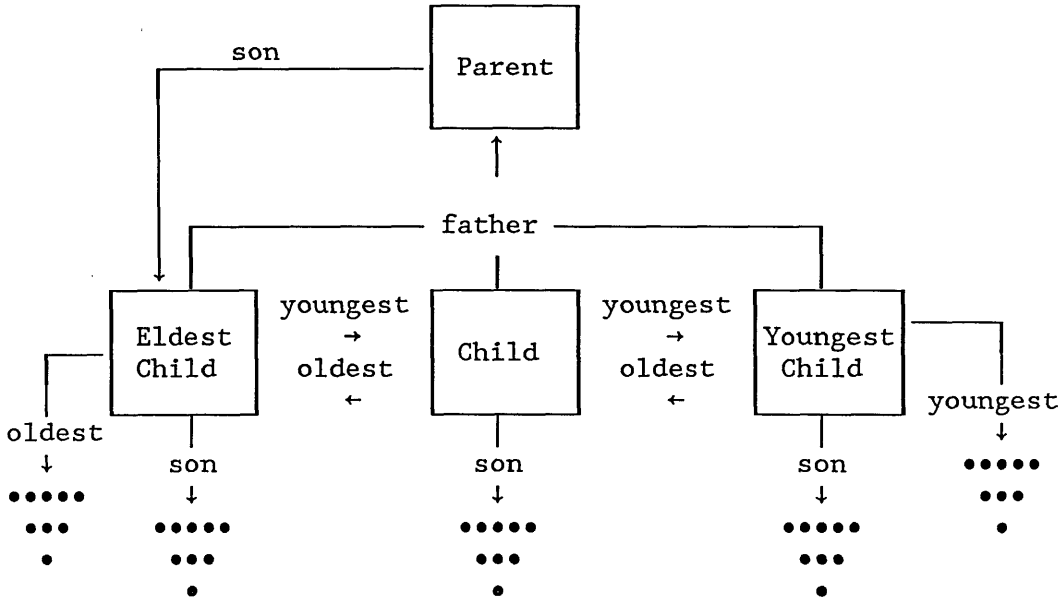


Figure 6.17: Node Linkages

6.2.2. Displaying the Coding Tree

The design of the coding tree was a compromise between conflicting requirements. While working at a particular node it is important to have some knowledge of its context. This suggests including as much of the surrounding tree as possible on the screen. Additionally, each node must display information to identify it from other nodes on the tree, suggesting a large box to represent each node.

The original implementation was to have a separate data structure to represent the screen. When the screen was required to be drawn, the information from the active node would be inserted in the structure at the central box of the middle line. Information about the older and younger brothers being inserted in the

appropriate boxes around the centre. The parent of the active node would have its information inserted in the top line and the siblings' information would be copied into the bottom line's boxes. The Screen Manager would then use the information in the structure to produce a representation of the screen.

The above implementation led to overheads of both time and space owing to the duplication of information. Although there was no realistic limit on the available space, the time factor proved unacceptable.

In the present implementation the screen data structure was overlaid onto the *_TREE* data structure, thus removing the need for extra copies of the data to be stored. The screen layout that provided the best presentation of the design diagram is selected by using the terminal capabilities stored in the *termcap* terminal database. These capabilities include the width and the number of lines on a terminal screen. Based on this information, the size and shape of nodes for the *coding* tree are determined.

The graphical output requires knowledge of the subtree that is to be displayed on the screen, together with information about the position of each node on the screen. The position for each of these nodes is stored in a data structure called _LINENODE (Figure-6.18).

The *node* window on which the *coding* tree is drawn is divided into three levels; _TOP_LINE, _MIDDLE_LINE and _BOTTOM_LINE.

```
# define  _TOP_LINE        32  /* top, middle   */
# define  _MIDDLE_LINE     64  /* and, bottom   */
# define  _BOTTOM_LINE    128  /* levels of the
                                * node window   */

typedef struct _LINENODE
{
    _TREE *left;      /* left of current node   */
    _TREE *right;     /* right of current node  */
    _TREE *first;     /* first node on a level  */
    _TREE *centre;    /* centre node of a level */
    _TREE *current;   /* reference to 'head'    */
} _LINENODE ;

_LINENODE _top;       /* top level    */
_LINENODE _middle;    /* middle level */
_LINENODE _bottom;    /* bottom level */
```

Figure 6.18: The C Data Structure for a _LINENODE

Three instances of the _LINENODE data structure are created, one for each level on the *node* window. These are *_top*, *_middle* and *_bottom*. The node of current interest (entry **current*), referenced by the global variable *head*, is placed in the centre (referenced by **centre*) of the screen, its parent above it and any siblings below it. The default is to place a brother on either side (entries **left* and **right*) of the current node (if they exist) and a maximum of four siblings below it. The system also indicates the presence of nodes outside the bounds of the screen.

To draw the screen, the line pointers, *_top*, *_middle* and *_bottom*, are set to point to the appropriate nodes on the tree. The bitwise operators are used, together with the manifest constants defined in Figure-6.18, to set the *options* field in these nodes to indicate the level on which the node is to appear. The layout procedures then assigns the coordinates to these nodes

using the _LOCS data structure. The graphics driver uses this information stored in the *_LINENODE* data structure with Curses screen management functions, to draw the terminal screen.

6.2.3. Displaying the Skeleton Tree

Tree data structures are commonly used in computer science and algorithms to build and use them are well known [59]. However, it is difficult to generate an aesthetically pleasing drawing of non-binary trees automatically [112,85,100], and published algorithms such as given by Wirth [116] do not provide a satisfactory solution to the problems involved. There are three main problems involved with drawing trees:

(1) computing the X and Y co-ordinates of a printed node

(2) printing nodes sequentially owing to the physical characteristics of standard printers

(3) the limited page width, as the area required to print a large tree commonly exceeds that of a page.

The printing algorithm for the *skeleton* tree in the APE system partitions the output into vertical stripes corresponding to the page width. These stripes are printed sequentially so that the desired output may be obtained by assembling these stripes side by side.

The algorithm works in two phases. The first phase computes the horizontal position of each node in the tree and creates an

auxiliary list structure corresponding to the graphical structure of the output. In the second phase, the algorithm scans the auxiliary list and prints the output in vertical stripes, depending on the page width, which can be manually assembled later.

The auxiliary list is defined by the C data structures called *reflink* and *refhead* and is shown in Figure-6.19.

```
typedef struct reflink
{
    int              pos;       /* position                  */
    int              next_pos;  /* next available slot       */
    _TREE           *tnode;     /* reference to node         */
    struct reflink *l_next;     /* link to right node        */
    struct reflink *l_prev;     /* link to left node         */
} reflink;

typedef struct refhead
{
    reflink        *node;     /* references a reflink node */
    struct refhead *h_next;   /* link to right node        */
    struct refhead *h_prev;   /* link to left node         */
} refhead;
```

Figure 6.19: Data Structures for the Skeleton Tree

The key to the algorithm is in computing X and Y coordinates. Although the Y position is directly related to the level on which the node is to appear, the X position is a function of the positions of its neighbours. This global information is not easily diffused by the standard traversal algorithms.

Initially, the position (*pos*) of a node is determined with respect to its parent. As a second step, *pos* may be altered depending on the existence and position of a neighbour node on the same level, in which case *pos* is decreased to maintain one empty

slot between itself and the neighbour. The final value of *pos* depends on the existence and position of its siblings and is computed to be halfway between its leftmost and rightmost siblings. The algorithm attempts to place the siblings of a node one slot to the right and left of its own position; if a single node exists, that node is placed directly below the parent; if no siblings exist, the computed position is maintained.

As positions are being computed, the algorithm builds an auxiliary data structure. This structure uses the *refhead* and *reflink* elements. For each node, a *reflink* is created containing a pointer (*tnode*) to the node and the node's computed position. The *reflink*'s for the nodes on one level are chained by the *l_next* and *l_prev* pointers to form a doubly linked lists headed by *refhead* elements that are themselves linked together in the same way.

The layout procedure then uses the data structure built by the above algorithm to print the tree using multiple stripes if necessary. The procedure first builds the auxiliary data structure, then proceeds with printing the stripes, lines and slots respectively.

The algorithm used is general in that it requires no extra fields to be defined in the tree nodes. Additionally, the algorithm uses multi-stripe output to print a tree of any width.

CHAPTER 7

Validating and Verifying the Implementation

Formal methods play an important part in rapid prototyping, validating, verifying and documenting computer systems. The use of natural language in defining such systems is prone to ambiguity and vagueness, whereas mathematical notations, although more precise, are cryptic. Formal specification techniques are therefore becoming increasingly important for defining systems moreover, they enable the use of formal methods in subsequent phases of the development cycle. The potential benefits of applying formal methods to the design and production of software is a topic of much discussion [53,54,50].

The value of a specification, such as that presented in this chapter, is that it defines the system in question and so enables its properties to be determined by reasoning rather than by performing experiments. The latter approach can be difficult if the system is complex, and costly if it has not yet been built. Specifications can be constructed that answer the questions raised and that adopt a certain level of abstraction. If these specifications are presented in a mathematical notation, the question of their meaning and consistency can then be answered using mathematical means.

Formal methods were applied to validate and verify the imple-

mentation, and as an immediate benefit have resulted in an increased understanding of the system and also led to the detection of some errors in the implementation. In this chapter, the use of formal methods with particular reference to the description of an existing piece of software is outlined. The specification process itself is described and a sample specification presented. A module of the APE system is then specified in Standard ML [89] following that the questions that were raised during the specification process are discussed.

7.1. The Uses of Formal Methods in Existing Systems

Formal methods can be used throughout the software development life-cycle;

- to describe precisely the behaviour of a system
- as an unambiguous starting point for producing system documentation
- to check for correctness
- for testing possible changes, modifications or reimplementations of the system.

Using formal methods involves developing a precise description of the system so that its structure and detail can be understood and communicated within a design team, and between designers and clients. This information is best obtained by building a model of a system using formal methods.

The task of constructing a model is an iterative process. An initial model is put forward and then refined by the application of domain independent principles. The refinement process is interspersed with frequent dialogues to help clarify details that are ambiguous or undefined in the initial model. The final model should provide a specification of the system at a high level of abstraction. In particular, it should avoid describing the representation of data within programs and details of the algorithms that have been used to implement the system. A more detailed discussion is given elsewhere [11,58,39,97].

There are both short and long term benefits to be gained in specifying existing systems. In the short term, formal methods help to

(1) uncover inconsistencies in the documentation and highlight ambiguities, redundancy or non-orthogonal properties of functions in the existing system

(2) check the design for consistency and completeness

(3) ensure that the implementation meets its formal specification

(4) highlight anomalies in the existing system and suggest ways in which the system could be improved.

In the longer term formal methods

(1) facilitate user learning, since they provide an appropriate level of abstraction of the functional behaviour of the

system

(2) provide a vehicle for investigating system reimplementations, as the specification is implementation independent

(3) enable modifications to be made with reference to the previous specification; the behaviour of these modifications, and their effects on existing parts of the system can be examined as a whole

(4) provide a suitable prototype for experimenting with the system

(5) enhance the ease with which the system designs can be understood and communicated within the design team.

Formal specifications are usually based on the use of mathematical concepts in describing the requirements of a system. The method concentrates on defining what a system does rather than the implementation specific detail of how the system does it. If the designers are familiar with the notation, formal specifications are easier to reason about as they do not suffer from any of the ambiguities of natural language description. This alone has a significant advantage in that it provides valuable feedback and allows for a more precise communication between designers.

Specification can be seen as complementary to prototyping. The two techniques are particularly closely linked when the formal specification language is directly executable, so that the specif-

ication acts as a prototype [2]. Experimenting with specifications is likely to reveal errors, inconsistencies and omissions at an early stage of the design process. This, therefore, provides a faster and cheaper way of incorporating modifications to the original design. However, since the specification is implementation independent, it is difficult to predict the effect of the alterations on the performance of the system and the difficulties involved in implementing the changes. As the specification is at a high level of abstraction, "it can give a better insight into the interaction of changes with other component of the system; it is just these high level interactions which get lost in the detail of implementation" [39]. Therefore, it is important for the specifier, while experimenting with the specification, to be aware of the implementation consequences before making any decision. Any ambiguities must be further investigated before implementation.

7.2. The Notation

A subset of the applicative language SML, Standard ML [38,72,19], developed at the University of Edinburgh, is the primary notation that has been used in this specification work. ML, which stands for Meta Language, is a statically-scoped functional language that features a flexible but completely secure polymorphic type system [73].

The specifications in this chapter are a combination of formal SML and informal explanatory English. Formal specifications are generally hard to construct, use and understand, which

according to Sannella [89], "may be a blessing in disguise". To build a readable specification, both formal and informal parts are necessary. The formal text, although precise and unambiguous, can be too terse for ease of reading and often its purpose needs to be explained. Conversely, an informal natural language explanation can easily be ambiguous or vague and needs the precision of a formal language to make the intent clear. Therefore, in creating a good specification the composition and style of the informal natural language text together with the structuring of the specification are as important as the formal text.

A specification should reveal the operation of the system a little at a time. These portions can then be combined to give a specification of the whole. This style of presentation, which Wikstrom [113] calls "structured growth", is preferred since giving the specification in small portions can enable each piece to be understood. Furthermore, when these pieces are put together the understanding of the parts that has already been gained can lead more easily to an understanding of the whole. This structural evolution of programs from specifications, by verified refinement steps so that a correct result is guaranteed is "perhaps the most exciting potential application of formal specifications" [89]. However, understanding complex specifications that are developed in this way can be difficult, as one needs to remember the functions of all the parts and understand the way in which they are combined. In such cases, Hayes [39] suggests that it would be useful to provide both a portion by portion development of the

specification and an expanded monolithic specification as well.

To help write specifications a portion at a time, SML provides a facility by which one can describe and name parts of a specification using modules†. A module consists of two parts – an interface (signature) and an implementation (structure), which are defined separately. Every structure has a signature that gives the names of the types and functions defined in the structure. Structures may be built on top of existing structures, thus generating a hierarchy of structures. This hierarchy is also reflected in its signature. Modules are "parameterised" structures [88], since a module can be applied to a structure to return another structure. The following is a simple example of a program in SML with modules that implements an array of integers indexed starting from zero using a list of integers:

```
signature ArraySIG =
    sig
        type array
        val  empty: array
        val  retrieve: (int × array) → int
        val  add: (int × int × array) → array
    end ;

structure Array : ArraySIG =
    struct
        type array = int list
        val empty = nil
        fun retrieve (n, nil) = 0
          | retrieve (n, h::t) =
              if n = 0 then h
              else retrieve (n -1, t)

        fun add (n, h, nil) =
              if n = 0 then h::nil
```

† These extensions to SML for modular programming were first proposed by MacQueen [66].

```
            else 0::add (n -1, h, nil)
        | add (n, h, a::t) =
            if n = 0 then h::t
            else a::add (n -1, h, t)
    end ;
```

The information provided by a signature is limited, and is not sufficient, for example, to prove program correctness or for program documentation. To make the interface more useful, signatures have been extended to include axioms. Axioms (sentences of first order logic in this thesis), place constraints on what operations the operations in a signature are supposed to perform. This extension, with the aim of doing formal development and proofs of SML programs, has been termed "Extended ML" by Sannella and Tarlecki [88]. An example of an axiomatic signature in SML is the following specification to find the largest integer in a given list of integers:

```
signature MaxSIG =
    sig
        val member : ((int list) × int) → bool
        axiom (∀ mem: int)
            => member (nil, mem) = false
        axiom (∀ mem)
            => member (hd::tl, mem) =
                (mem = hd) ∨ member (tl, mem)

        val max : (int list) → int
        axiom (∀ IntList: int list)
            => IntList ≠ nil
                ⇒ member (IntList, max (IntList))
        axiom (∀ a: int, IntList)
            => member (IntList, a)
                ⇒ max (IntList) ≥ a
    end ;
```

In the above signature:

'⇒' denotes implication. The expression on the left hand side of the '⇒' may be an equation (as above) or a boolean-valued expression that can be regarded as abbreviating equations of the form 'expr=true'.

'=>' is used to write unnamed axioms and functions [89]. For example,

```
fn x => []
```

evaluates to a function that, when given an object x, returns an empty list.

Using the features described above, axiomatic specifications have been written in SML to describe some of the more desirable features of the APE system and are discussed below.

7.3. Specification of a Module in Standard ML

The main aim in formally specifying the APE system was to build a model of the system that was consistent with the implementation. In attempting to do so, questions arose that had to be answered in consultation with the source code and the manual. Since the system was developed recently, straightforward questions about its operations were answered by the implementors. Additionally, experienced users of the APE system were available who were willing to help in the specification process.

Building a model for the APE system was an iterative process. It involved forming a crude initial model that was extended to

cover facets of the system not initially dealt with, and used for redesigning or refining inconsistencies that were found. During this design process questions were formulated about the desired behaviour of the system that were then answered from the specification. The final model consisted of informal natural language text and SML, and was comprehensive enough to describe the essential aspects of the system's behaviour, but was sufficiently abstract in that it did not burden the reader with the kind of detail that would appear in the implementation.

It is important that the final model should provide a specification of the system at a high level of abstraction, and avoid giving away any unnecessary implementation details. In particular, it should avoid having to describe the representation of data with programs and algorithms that have been used to implement the system. Some questions led to inconsistencies being highlighted between the model, implementation and documentation. These inconsistencies were generally caused by errors in the implementation or by the informal natural language description of the manual. The specification was then given to people experienced in the use of formal methods who then commented on its style and level of abstraction, and suggested ways in which it may be improved or simplified. These people also had experience of using the APE system that was found to be useful, as inconsistencies between the specification and the implementation could be bought to the notice of the specifier.

The APE system encourages structured programming and imposes a tree structure on the problem under consideration. A tree, in the APE system, is composed of nodes connected by branches that occur singly or in sets and whose members may be sequential or selected, but not both. This contributes to the simplicity of the tree structure because it ensures mutual exclusion in selection. Nodes may be iterated as well, although following the syntax of Jackson's method of program design [52], iterative nodes may belong to sequences alone. The ultimate constituent of the APE system is a node. The type of a node can be formally specified in SML as:

```
signature NodeSIG =
    sig
        type     DATA      ( of type string )
        type     NUMBER    ( of type integer )
        datatype STATUS    = current
                           | not_current
        datatype NODE_TYPE = sequence
                           | selection
                           | iteration
        datatype NODE      = node of (NUMBER ×
                                      STATUS ×
                                      DATA ×
                                      NODE_TYPE ×
                                      NODE list)
    end ;
```

This recursive data type definition for a 'NODE' is a natural and appropriate way of representing trees and parallels the informal description of the tree (Figure-3.1) and its C implementation (Figure-6.13). Thus the tree in Figure-3.1 can be written as:

```
node (1, current, "Menu System", sequence,
  [node (2, not_current, "Menu Display", iteration,
    [node (3, not_current, "Display Menu", sequence, nil),
     node (4, not_current, "Display Prompt", sequence, nil),
```

```
    node (5, not_current, "Get Response", sequence,
     [node (6, not_current, "Invalid Response", iteration, nil)]),
    node (7, not_current, "Valid Response", sequence,
     [node (8, not_current, "Add", selection, nil),
      node (9, not_current, "Delete", selection, nil),
      node (10, not_current, "Re-order", selection, nil),
      node (11, not_current, "Exit", selection, nil)])]) ;
```

An important point to be noticed here is that functions on recursively defined data values, such as the above tree, are defined recursively.

The APE system encourages the designers to split a program into components of manageable size. Each sub-routine in a user program would consist of an individual tree structure. A complete program would consist of a single top-level tree diagram that may contain calls to sub-routine tree diagrams. This can easily be represented as a list of TREE's. The type of a tree can be formally specified in SML as:

```
signature TreeSIG =
    sig
        structure Node : NodeSIG
        datatype TREE = tree of (Node.NUMBER ×
                                 Node.STATUS ×
                                 Node.DATA ×
                                 Node.NODE)
    end ;
```

Hence, trees defined as

```
tree (1, current, "Program Menu",
    node (1, current, "Menu", sequence, nil)) ;

tree (1, not_current, "Module Add",
    node (1, not_current, "Add", sequence, nil)) ;

tree (1, not_current, "Module Delete",
    node (1, not_current, "Delete", sequence, nil)) ;

tree (1, not_current, "Module Re-order",
```

```
    node (1, not_current, "Re-order", sequence, nil)) ;
```

can be represented, using the above definitions, as:

```
[tree (1, current, "Program Menu",
    node (1, current, "Menu", sequence, nil)),
 tree (2, not_current, "Module Add",
     node (2, not_current, "Add", sequence, nil)),
 tree (3, not_current, "Module Delete",
     node (3, not_current, "Delete", sequence, nil)),
 tree (4, not_current, "Module Re-order",
     node (4, not_current, "Re-order", sequence, nil))] ;
```

As an example of specification and program development in SML, and based on the above definitions for a 'NODE', axiomatic specifications are developed that model the *paste* operation in the APE system. The specifications presented here are derived by the specification process discussed in the earlier sections. SML signatures are used in describing this operation of the APE system. The complete implementation for these signatures is presented elsewhere [41].

7.3.1. The Environment

The specification of the APE environment is first presented. A tree in the APE system is a finite sequence of nodes of any length, the definition for which has been given above. The size of the tree is always equal to the number of append operations that have been performed on the current tree since its creation and is independent of all other operations, such as copy.

The initial state of the design diagram is given by a single sequence node called the 'root'. In general, a specification can

be constructed to determine whether a node (n), exists in a single tree;

```
signature NodeSIG =
    sig
        ...          ...
        val node_in_tree : ((NODE list) × NODE) → bool
        axiom (∀ n: NODE)
            => node_in_tree (nil, n) = false
        axiom (∀ n: NODE)
            => node_in_tree (hd::tl, n) =
                 (n = hd)
                ∨ node_in_tree (tl, n)
                ∨ node_in_tree (subtree (hd), n)
    end ;
```

At any time, the system will contain a single node whose STATUS is 'current'. Such a node is called the *current* node and is visually represented in the APE system by a cursor sitting on the node. This is specified by the specification for 'head' as:

```
signature NodeSIG =
    sig
        ...          ...
        val head_in_tree: (NODE list) → bool
        axiom head_in_tree (nil) = false
        axiom (∀ NodeList: NODE list)
            => head_in_tree (NodeList) = (∃ n: NODE)
                => extract_node_status (n) = current
                ∧  node_in_tree (NodeList, n)

        val head : (NODE list) → NODE
        axiom (∀ NodeList: NODE list)
            => ¬head_in_tree (NodeList)
                ⇒ head (NodeList) = NULL
        axiom (∀ NodeList: NODE list)
            => head_in_tree (NodeList)
                ⇒ (∃ n: NODE)
                    => head (NodeList) = n
                        ∧ extract_node_status (n) = current
    end ;
```

and where the NULL node is,

```
val NULL = node (0, not_current, "", sequence, []) ;
```

The *paste* function makes use of two further SML modules, MakeSIG and MoveSIG, that model the operations of generating nodes and traversing the tree diagram in the APE system, respectively. Before proceeding to describe the *paste* operation, an overview of these two modules is presented.

There are three basic types of nodes in the APE system, namely sequence, iteration (or loop), and selection. Nodes in the APE system cannot be mixed, i.e. it is an error to have a selection node alongside a node denoting a sequence. When a node is added to a tree, the original tree is changed. Nodes in the APE system are generated under the *current* node. Additionally, any sub-tree below this node (current) is deleted before new siblings are generated. The SML module MakeSIG specifies such an environment for adding nodes to a tree.

The APE system also provides several methods for traversing the tree design diagram. The simplest way is by moving from one node to another. This is achieved by

(1) using the cursor keys

(2) the four *vi*‡ keys

(3) using the system's pull down menu facilities.

---

‡ *Vi* is the UNIX screen oriented text editor.

In either case, this mode of operation will enable a user to traverse the design diagram one node at a time. The system also provides options for faster traversal of the tree. All traversing commands in the APE system are issued with respect to the *current* node. The SML module MoveSIG specifies the environment used in traversing the design diagram.

7.3.2. The Paste Operation On Trees

Having specified the environments, the specification for the *paste* operation on trees is presented in this section. In the APE system, a node or a sub-tree that was previously saved by using the *copy* or *cut* operations, can be inserted to the left, right or below the current node by the *paste* function.

All operations work directly on a 'NODE list' that denotes a tree (t). The operation *cutit* removes a NODE (together with the sub-tree below it), from a tree (t).

```
signature EditSIG =
    sig
        structure Node: NodeSIG
        structure Make: MakeSIG
            ...    ...
            ...    ...
        val cutit: ((Node.NODE list) × Node.NODE) →
                    (Node.NODE list)
        axiom (∀ n: Node.NODE)
            => cutit (nil, n) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.node_in_tree (t, n)
                ⇒ (∃ t': (Node.NODE list))
                    => cutit (t, n) = t'
                        ∧ Make.creat_nodes (t',
                        Node.extract_node_type (n), 1) = t
    end ;
```

The above specification for *cutit* states that a node (n) can be deleted from t, if t is non-empty and n exists in t. The function *creat_nodes* generates a new node, under the *current* node, whose type is similar to that of n. Any sub-tree of the *current* node is first deleted before the new nodes are created.

The operation for adding a node (n) to the left of the *current* node can be specified as follows:

```
signature EditSIG =
    sig
            ...     ...
            ...     ...
        structure Move: MoveSIG

        val valid_paste: (Node.NODE × Node.NODE) → bool
        axiom (∀ x, y: Node.NODE)
            => valid_paste (x, y) =
                Node.extract_node_type (x) =
                Node.extract_node_type (y)

        val paste_to_left: ((Node.NODE list) × Node.NODE) →
                            (Node.NODE list)
        axiom (∀ t: (Node.NODE list), x: Node.NODE)
            => valid_paste (x, Node.head (t))
                ⇒ (∃ t' : (Node.NODE list))
                    => paste_to_left (t, x) = t'
                        ∧ cutit (t', x) = t
    end ;
```

According to the above signature, when a node (x) is added to a tree (t) it results in a new tree (t'), provided the following conditions hold:

(1) the *current* and x nodes are both of the same type; this is important as nodes of one type cannot be added to a node of another type on the same level

(2) if the node x is deleted from the new tree t' it results in the original tree t

(3) the node x has a younger brother in the new tree t'.

Finally, the operation to specify the *paste* function is given by

```
signature EditSIG =
    sig
        structure Node: NodeSIG
            ...     ...
        datatype SIDE = up
                      | down
                      | left
                      | right
            ...     ...
        val paste: ((Node.NODE list) × Node.NODE × SIDE) →
                    (Node.NODE list)
        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
    end ;
```

The function takes a tree (t) and a node n (that could contain a sub-tree below it) and returns the new tree with the node n added to the left of the *current* node. The operations will fail if there is no tree, or the tree has no current node. The cases when the *paste* operation is not performed are

(1) if the tree is empty, in which case the empty tree is returned

(2) if the tree contains no node marked as *current*, in which case it returns the original tree

(3) if there is nothing to paste, i.e. node n is the NULL node, then the original tree is returned.

The complete environment is given is Figure-7.1.

Following Sannella [89], the specification of types and functions that are intended to be strictly local to the specification of other types and functions in the signature are enclosed in a box.

7.4. Questions and Problems Identified by Interrogating the Specification

The questions that arise during the specification process suggest problems either in documentation or in the implementation of the system. This provides the system designers and maintainers with valuable feedback on possible problem areas. Therefore, once a model of the system was completed, it was reviewed by several people experienced in using formal methods. The reviewers compared the model with the relevant sections of the manual and looked for ambiguities or inconsistencies in the specification. All queries that arose were then answered either by consulting the formal model of the system or by consulting experienced users of the system. Several questions were asked of the system during this review process. The questions can be classified into two categories, namely

```
signature EditSIG =
    sig
        structure Node: NodeSIG
        structure Make: MakeSIG
        structure Move: MoveSIG
        ...     ...
        datatype SIDE = up | down | left | right
        ...     ...
        val paste: ((Node.NODE list) × Node.NODE × SIDE) →
                   (Node.NODE list)

        val cutit: ((Node.NODE list) × Node.NODE) →
                   (Node.NODE list)
        axiom (∀ n: Node.NODE)
            => cutit (nil, n) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.node_in_list (t, n)
                ⇒ (∃ t': (Node.NODE list))
                    => cutit (t, n) = t'
                        ∧ Make.creat_nodes (t',
                          Node.extract_node_type (n), 1) = t

        val valid_paste: (Node.NODE × Node.NODE) → bool
        axiom (∀ x, y: Node.NODE)
            => valid_paste (x, y) =
                Node.extract_node_type (x) =
                Node.extract_node_type (y)

        val paste_to_left: ((Node.NODE list) × Node.NODE) →
                           (Node.NODE list)
        axiom (∀ t: (Node.NODE list), x: Node.NODE)
            => valid_paste (x, Node.head (t))
                ⇒ (∃ t': (Node.NODE list))
                    => paste_to_left (t, x) = t'
                        ∧ cutit (t', x) = t

        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
```

Figure 7.1: The Specification for the Paste Operation

(1) General questions: dealing with the overall system specification

(2) Specific questions: that were asked of individual modules.

Some of the general questions that were asked of the system included:

- Is the model inconsistent with the implementation?
- Is the model inconsistent with the documentation?
- Is the model independent of implementation details?
- Is the chosen model at a suitably high-level of abstraction?
- Is the implementation correct?
- Are there any inconsistencies or ambiguities in the documentation?

Some of the questions that were raised during the specification process of the *paste* operation were:

- Are the default actions consistent throughout the chosen model?
- What happens to the previous siblings of the *current* node after a *paste* operation?
- How often can a sub-tree be added to the *current* node?

- Can any sub-tree be added to the left of the *current* node?

- Can the *root* node have any brothers?

A more detailed examination of the last two questions with respect to the earlier model of the *paste* operation is presented below. That model worked for all cases, except when

(1) the tree had no nodes

(2) the tree had no *current* node

(3) the node to be pasted was the NULL node.

A closer examination of the manual page revealed that the model did not match its informal natural language specification. This suggested that the model was inconsistent as the manual entry specified that the *root* node could not have any brothers either to its left or right. In the earlier model the specification for the *paste* operation did not check whether the *current* node was also the *root* node. The following invalid *paste* specification was used:

```
signature EditSIG =
    sig
        ...     ...
        ...     ...
        val paste: ((Node.NODE list) × Node.NODE × string) →
                    (Node.NODE list)
        ...     ...
        ...     ...
        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
```

```
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
    end ;
```

That is, the *paste* function would invoke the *paste_to_left* function if the tree (t) had a *current* node. The new corrected model contains the axiom

```
axiom (∀ t, n)
    => Move.root (t) = Node.head (t)
        ⇒ paste (t, n, _) = t
```

The altered specification for the *paste* operation is:

```
signature EditSIG =
    sig
        ...     ...
        ...     ...
        val paste: ((Node.NODE list) × Node.NODE × string) →
                    (Node.NODE list)
        ...     ...
        ...     ...
        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
        axiom (∀ t, n)
            => Move.root (t) = Node.head (t)
                ⇒ paste (t, n, _) = t
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
    end ;
```

Several benefits were derived from this review process. The first was that people who were experienced in formal methods but were not part of the design and implementation process would be involved. This is crucial as designers and developers often try to find problems by forming a mental model of the implementation and

the manual. Another important benefit of the review process is that it provides an opportunity to modify parts of the model that have proved to be confusing or misleading. This is important to the system maintainers, since misunderstandings or the bad presentation of a concept can lead to a wrong decision by a user.

7.4.1. Adding New Modules

As a further example, the problems encountered in incorporating a new module in the APE implementation are considered. The APE system has been implemented using modules, each of which does a specific task. Before any new module is integrated in the APE system, an attempt is made to check for correctness. The checks were generally in the form of informal tests of the module on a few input values. No formal proof of correctness is attempted. A substantial amount of testing can be done by independently exercising each module. However, testing individual modules and comprehensively testing the system as a whole involves a big jump in the levels of abstraction and is rarely attempted.

Once the *paste* function passed a set of informal tests, it was incorporated into the system. This led to a few inconsistencies between the mathematical model and the implementation and suggested either that the model was incorrect or that there were bugs in the implementation. In the APE system different types of nodes cannot be mixed on the same level. For example, it is illegal to have a selection node alongside a node denoting a sequence.

In the mathematical model, this constraint was specified by *valid_paste*. The specification for *valid_paste* is simple, yet crucial in determining whether the system should proceed with the operation of adding a sub-tree to the left of the *current* node. However, the effects of not introducing this subtle test into the implementation had disastrous implications. It led to several bugs being generated when the system was in use. Additionally, most of the side effects caused by this omission were not immediately evident as they were triggered when some function, independent of the *paste* operation was attempted, usually resulting in the program being aborted.

This anomaly was highlighted by the mathematical model and resulted in the quick detection and correction of the error in the implementation. One can only speculate at the amount of time and effort that would otherwise have gone into locating and correcting the error had the specification of the system not been present. This example also demonstrates the usefulness of applying the specifications to highlight the interaction between modules, since the functions from one module will need access to components of another. However, any such inconsistencies discovered in the process of specifying a system should be closely examined before changes are carried out.

7.4.2. The Symbol Table

As a final example, the symbol table used by the APE system

in the programming mode of operation is considered. The representation of a program as an overview tree was initially used by the APE system for storing and retrieving declarations, thus enabling the symbol table to be block structured. All declarations local to a particular tree were stored at the root node of each tree, which as a rule displayed the name of the subroutine under it. This was considered a convenient way of keeping track of variables declared in all the blocks enclosing that point. Additionally, the information held in the root node is important in determining the interrelations between the different tree diagrams and is also used in producing the declaration section when generating the final program.

However, whilst specifying the symbol table module of the APE system, doubts were expressed about the complexity of the implementation. It was considered inappropriate to store declarations in separate trees. Experimentation with the formal model generated a simpler representation for the symbol table. In the new representation, symbols are linked together in a list; the only access to it being through the functions *lookup* and *install*. This makes it easy to change the symbol table organisation should it become necessary. The symbol table uses linear search, which is entirely adequate for the APE system, since variables are looked up only during parsing, not execution. *Install* places a new variable with its associated type at the head of the list.

Several topics in the specification of the APE system, such

as the *history* mechanism and the automatic program generator, have not been covered in this thesis. Details of these are included in Heerjee [41] and the user manual [46] of the implemented system. An attempt has also been made at mathematically proving that the implementation in SML supports the suppositions made in the axioms. Correctness proofs of two SML functions, *head* and *paste_to_left*, are presented in Appendix B. These proofs lack complete rigor but adequately shows the derivation of functions from the axioms.

CHAPTER 8

Empirical Evaluation of the APE System

Researchers in the field of human-computer interaction have long held the view that well designed interactive systems increase performance levels over conventional techniques, and intuitively this seems logical. However, little work has been done to empirically support this claim. Increasing effort is being devoted to the development of interactive systems, but their positive impact is seldom evaluated. This is perhaps either because it may be too costly or too difficult.

A computer-based interactive system is a tool like any conventional tool, in the sense that it is built to meet some need. Computer users are various and variable and it would be unusual to have performance levels that were unchanging, or those that were permanently agreed. Nevertheless, evaluation is important if the system is going to meet the needs of the users. Therefore, once a major section of the APE system was implemented, it was decided to evaluate users' ability in accomplishing a simple set of tasks and seek their reaction to using the system.

Several benefits were anticipated from testing the interface of the system. The first was that a different perspective would be obtained, as users would be involved in the evaluation process. This is crucial, as designers and developers often try to find

problems in the human-computer interface by posing as users. This is no substitute for human factors testing with real users. Such testing provides some valuable information on users' critical appraisal of the overall system. Some shortcomings may be unavoidable but others can be changed to accommodate users. Another important benefit of this type of testing is that it provides an opportunity to modify sections or parts that have proved to be confusing or misleading before the final product is released. According to Lund [65], "making an initial good impression is much better than trying to make changes once the product is already being used". Testing also helps in highlighting areas where the users get lost. This is important as the developers should know exactly what led to a wrong decision. Often misunderstanding or bad presentation of a basic concept may be the cause.

The APE system was evaluated in order to investigate the claim that properly designed interactive tools can increase productivity and have advantages over conventional methods of generating code. The empirical evaluation was also intended to provide feedback on possible modifications to be made to the system. The evaluation was carried out once a major section of the APE system was implemented, and consisted of a questionnaire and an experimental comparison with conventional methods of programming.

During the testing process, special emphasis was placed on some specific areas of the interface. First time problems encountered by the user were noted, as initial problems become less

problematic when encountered several times. Eliminating this initial confusion for the user could make a substantial difference. Other types of information that were useful related to the user reaction to specific features of the system, such as an on-line help facility.

The Questionnaire and Comparative evaluation of the APE system is now described in detail in the sections below.

8.1. Questionnaire Evaluation

The questionnaire evaluation was designed to elicit users' general impression about the system and its interface, and their detailed views on more specific aspects of the system.

The points addressed in the questionnaire ranged from ease of use, through to the perceived utility of the system, and even its visual appearance. Some points that were less central to the interface design itself, but which might affect the users' perception of its quality were also included.

8.1.1. Apparatus

Trials were conducted on CIT-101e (VT100 compatible) monitors, linked to a VAX-11/750, or Micro VAX work-stations. UNIX /ULTRIX was the operating system used in these trials, and was the system on which APE was developed and now runs. The input device was the alphanumeric keyboard that was also used as a menu selection device.

8.1.2. Subjects

The questionnaires were given to forty-seven students (mean age 30.5 years, range between nineteen and forty-two years) at Dundee College of Technology. These students were enrolled in one of four courses; post-graduate diploma in Software Engineering, third year undergraduate science degree, fifth year Applicable Mathematics degree, or a Higher National Diploma in Computing.

All candidates had some previous knowledge of computing. Some possessed a knowledge of the UNIX /ULTRIX operating system, although this was not necessary.

8.1.3. Method

The evaluation procedure consisted of three stages. Initially, a brief explanation of the design technique used by the APE system was given. This was found to be necessary because the APE system is conceptually based on Jackson diagrams, and few subjects had experience of this. Subsequently a demonstration was provided to familiarise the users with the basic facilities and tools that were essential for driving the system. The experimenter was always present to answer any queries. The example consisted of using the system to generate a program in Pascal that found the maximum of a set of numbers. This example demonstrated essential features of the system such as the menu interface, on-line help facilities and ways of directly manipulating the design diagram using the command language. First-time users were then allowed to practice and familiarise themselves with the system, following

that they were given instructions for the task that they were required to perform.

The subjects then completed one of three programming tasks in order to familiarise themselves with the system. The tasks were chosen to cover a range of applications in Pascal programming. One task was to write a program that played a simple game of noughts and crosses, while the second task was to write a scanner that produced a list of identifiers in a Pascal program. The final task was to generate a stack implementation that would also handle the two cases of stack overflow and underflow. At the end of the familiarisation, subjects were handed the questionnaire that they completed in their own time.

There were three types of question in the questionnaire. The first set (seven questions) concerned commands offered by the system to the user. Another set (ten questions) dealt with specific questions on existing features, while the third set (six questions) consisted of general open-ended questions that concerned users' opinion on some broader aspects of the system. At the end of the questionnaire, users were encouraged to state their complaints and desired modifications to the system. A conventional five point rating scale was used in the questionnaire [79], and the users indicated their degree of agreement/disagreement with the statements given in the questionnaire. A copy of the questionnaire and detailed results are included in appendices C and D, respectively.

8.1.4. Results

A total of forty-seven questionnaires were administered of which forty-five were completed and returned. The results presented in this section are based on the median ratings given to the questions and are shown in parenthesis.

With respect to their general impression, the subjects found the interface good (5), easy to use (4) and easy to learn (4). Subjects found that the response time of the graphical interface was fast (5), but said that a high level of concentration was required in using the system (4).

Subjects had little or no previous experience of using Jackson diagrams (1). During the learning process, the subjects had some practice sessions with a learning package that was incorporated into the APE system. This was found to be useful (4).

Several questions were asked in order to evaluate the screen design. These ranged from general questions such as the overall graphical interface of the system, to more specific ones, such as the quality of the menu interface.

Subjects agreed that the overall graphical interface was good (5), and were satisfied with the amount of information that was displayed on the screen (4). The help facilities, such as the instructions, on-line help and other accompanied documentation, provided by the system were also found to be good (4). In addition, subjects expressed their satisfaction with the pull down

menu selection system (4) and the advanced editing (such as cut and paste) facilities (4).

Subjects agreed that the different types of tree diagrams provided by the APE system were useful (5), and that sufficient information was displayed in a node of the tree (4). However, subjects were not certain whether more types of tree diagrams would enhance the system's capabilities (3).

Subjects were satisfied (4) with the history mechanism of the APE system. The run-time animation of the design diagram was also found to be useful (4) and all subjects agreed that the overview tree provided by the system was useful.

Two questions also dealt with the windows in the systems. Subjects felt that the number of windows should not be reduced (4). However, they were undecided (3) on whether the size of the text window needed enlargement.

Subjects found that the system did not allow for reasonably quick error correction (3). Answers to this question may have been influenced by the fact that the system being evaluated was a prototype and could sometimes allow illegal commands that took longer to correct.

In summary, the questionnaire revealed a favourable or encouraging response for most aspects of the user interface. This presumably reflects the use of human-factors design principles throughout the development of the system.

8.2. Comparative Evaluation

The second evaluation procedure involved a comparison between the APE system and conventional means of generating code. It was hoped that this would provide an objective test of the value of the system in program development.

8.2.1. Apparatus

The apparatus used in this experiment was the same as described in the questionnaire evaluation.

8.2.2. Subjects

Twenty post-graduate students (one female and nineteen males), from a diploma course in Software Engineering at Dundee College of Technology took part in this experiment. These subjects had an undergraduate college/university degree or diploma and were studying program design techniques, as part of their curriculum.

Subjects in this experiment had previous experience of using the APE system, as all had taken part in the questionnaire evaluation of the system. The subjects were divided into two groups equated for age and performance in class. The latter characteristic was assessed by a mark based on course-work assignments over the term. One subject was omitted from the results because his age and performance in class was markedly different from other subjects in his group. The first group (nine males, mean age 24.1 years, mean class performance 61.1%, SD = 6.5), used the APE system, while the second group (one female and nine males, mean age

25.7 years, mean class performance 64.0%, SD = 9.07), used the screen editor on their machines to produce the pseudocode.

8.2.3. Method

The task was to design a detailed pseudocode algorithm that implemented a restricted form of a decision table. The pseudocode algorithm to be produced had to go through the stages of reading the data into an array from a file. It would ask questions from the user and from the corresponding responses determine which outcome to print. Four pseudocode algorithms were required to be formulated using the conventional pseudocode structuring constructs (e.g. IF..THEN..ELSE, etc.), and a given set of primitives (such as READ, PRINT, Rules and Flags). The decomposition process had to be achieved in a top-down manner and each section was required to be refined until details could be input at the primitive level alone. The complete task is presented in Appendix E.

Both groups undertook their tasks in the same room, but no consultation between subjects was allowed. In addition, no help was given during the experiment and users were unaware that their performance was being timed.

8.2.4. Results and Discussion

The comparative evaluation was designed to compare the APE system with conventional means of programming, as it was hoped that this would provide an objective test of the value of the system in software development.

Statistical analyses (Mann-Whitney U-tests [94] and F [36,62] tests) were performed to examine the differences between the two groups of subjects. In each case, the null hypothesis of no difference between the systems was tested against an appropriate alternative. The Mann-Whitney U-test was preferred over Student's t-test for testing differences in location (mean or median) as it was suspected, and in some cases later shown, that the variances of the two groups were unequal.

8.2.4.1. Quality of the Solution

Since the APE system is conceptually based on Jackson diagrams, it was assumed that if there were any differences between the two systems, then the APE system would produce products of higher quality and/or lower variance. Hence the one-tailed tests were used.

Hypothesis 1

H_0: The solutions produced by the two systems are of equal quality. The median scores under the two systems are the same.

H_a: The solutions produced by the two systems are not of equal quality. The median score under the APE system is higher (one tailed test).

Hypothesis 2

H_0: The variances of the APE and Non APE groups are equal.

H_a: The variance of the Non APE group is greater (one tailed test).

The answer sheets that were returned by all nineteen subjects were marked by three independent lecturers who have experience of teaching program design techniques. The lecturers were not involved with the design, implementation or the evaluation of the APE system. Marks were given on a ten point scale, where ten was the maximum. The three independent sets of marks given by the lecturers X, Y, and Z to the nine subjects who used the APE system and the ten subjects who used the screen editor is shown in Figure-8.1.

The three markers were significantly consistent (Kendall's coefficient of concordance: (18) = 0.71, $p < 0.01$) [94]. Therefore each subject was assigned the mean of three lecturers' marks. These scores gave a median APE user score of 5.67 (S.D. = 1.12), and a median screen editor user score of 6.30 (S.D = 2.08). The small difference between these medians was not statistically significant ($U = 35.5$, $p > 0.05$).

While this result did not show a practical superiority in the quality of the solution produced on the APE system, it was at least no worse than a conventional code generating procedure. Therefore Null Hypothesis 1 was accepted.

The two groups, however, did differ in respect of the uniformity of the code produced by the two groups. Even though there was

Ape Group

Subject	X	Y	Z	$\bar{X}$	
1	5	6	6	5.6	
2	9	8	9	8.6	
3	5	8	8	7.0	
4	6	4	5	5.0	
5	5	6	5	5.3	
6	6	5	6	5.6	
7	6	6	5	5.6	
8	6	5	6	5.6	
9	5	6	6	5.6	
$\bar{X}$	5.8	6.0	6.2	6.0	($= \bar{X}_1$)
Median	6.0	6.0	6.0	5.6	
SD	1.3	1.3	1.4	1.1	($= s_1$)

Non APE Group

Subject	X	Y	Z	$\bar{X}$	
1	5	9	9	7.6	
2	4	4	3	3.6	
3	4	5	6	5.0	
4	8	8	7	7.6	
5	2	2	4	2.6	
6	6	7	6	6.3	
7	8	6	5	6.3	
8	5	5	9	6.3	
9	9	10	9	9.3	
10	8	8	9	8.3	
$\bar{X}$	5.9	6.4	6.7	6.3	($= \bar{X}_2$)
Median	5.5	6.5	6.5	6.3	
SD	2.3	2.5	2.3	2.1	($= s_2$)

Figure 8.1: Marks Assigned to Nineteen Subjects

a wide range of ability in each group the output from the APE group was of a consistent standard where as there was a marked

variation in the quality of the code produced using conventional methods. This was demonstrated by a significant difference in the variances of the scores obtained by each group ($F(9,8) = 3.42$, $p < 0.05$). Therefore Alternative Hypothesis 2 was supported.

8.2.4.2. Speed of Performance

The task completion time was measured, in minutes, for each user from the presentation of the problem to the return of the answer sheet. With respect to the time taken to complete a task, it was predicted that the APE system would produce the above results with less expenditure of time. Figure-8.2 reports the time taken by each user to complete the task.

Hypothesis 3

H_0: The median of the time taken by subjects in each group is the same.

H_a: The median of the time taken by subjects in each group is not the same. The median time under the APE system is less (one tailed test).

The median APE user time was 92.0 (SD = 5.28), and the median screen editor user time was 103.5 (SD = 13.88). The difference between the median times for each group was found to be significant ($U = 20.5$, $p < 0.05$). This result demonstrates that the users of the APE group took less time to complete their task as compared with the screen editor group. Therefore Alternative Hypothesis 3

Ape Group

Subject	Time (Mins)
1	88
2	92
3	99
4	84
5	88
6	92
7	97
8	99
9	90
$\bar{X}$	92.1
Median	92.0
SD	5.28

Non APE Group

Subject	Time (Mins)
1	107
2	80
3	100
4	120
5	90
6	93
7	93
8	120
9	109
10	117
$\bar{X}$	102.9
Median	103.5
SD	13.88

Figure 8.2: Task Completion Times (in minutes)

was supported.

The variance of the distribution of the times taken was also

considered. However, since it was difficult to predict which group would have the greater variance, if there were a difference, a two-sided alternative hypothesis (two tailed test) was used.

Hypothesis 4

H_0: The variances of the APE and Non APE groups are equal.

H_a: The variances of the two groups are unequal (two tailed test).

There was a marked difference in the variation for the time taken to produce code using the two methods. The screen editor group were significantly more variable. This was demonstrated by a significant difference in the variance of the scores obtained by each group ($F(9,8) = 6.91$, $p < 0.05$). Therefore Alternative Hypothesis 4 was supported.

8.2.4.3. Correlation Between Quality and Speed

Figure-8.3 and Figure-8.4 summarise the correlation between quality and time taken to complete the task by the APE and screen editor groups respectively.

Hypothesis 5

H_0: In the APE group, there is no correlation between the time taken to complete a task and quality of the solution.

H_a: In the APE group, there is a correlation between the time taken to complete a task and quality of the solution.

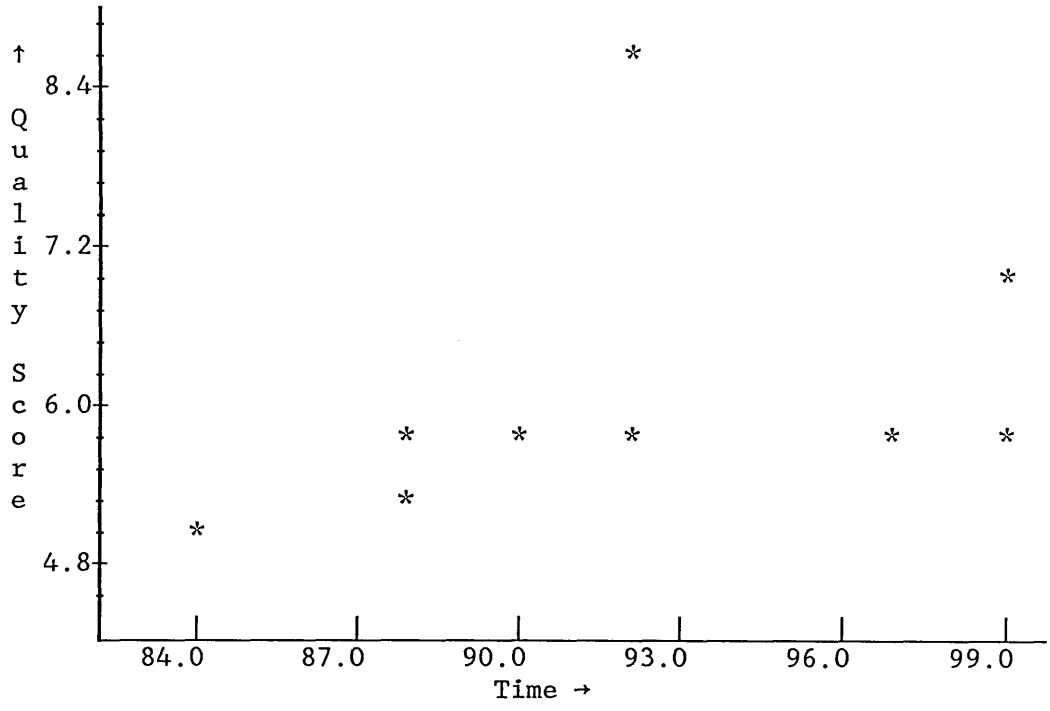


Figure 8.3: Quality vs. Time Taken for the APE Group

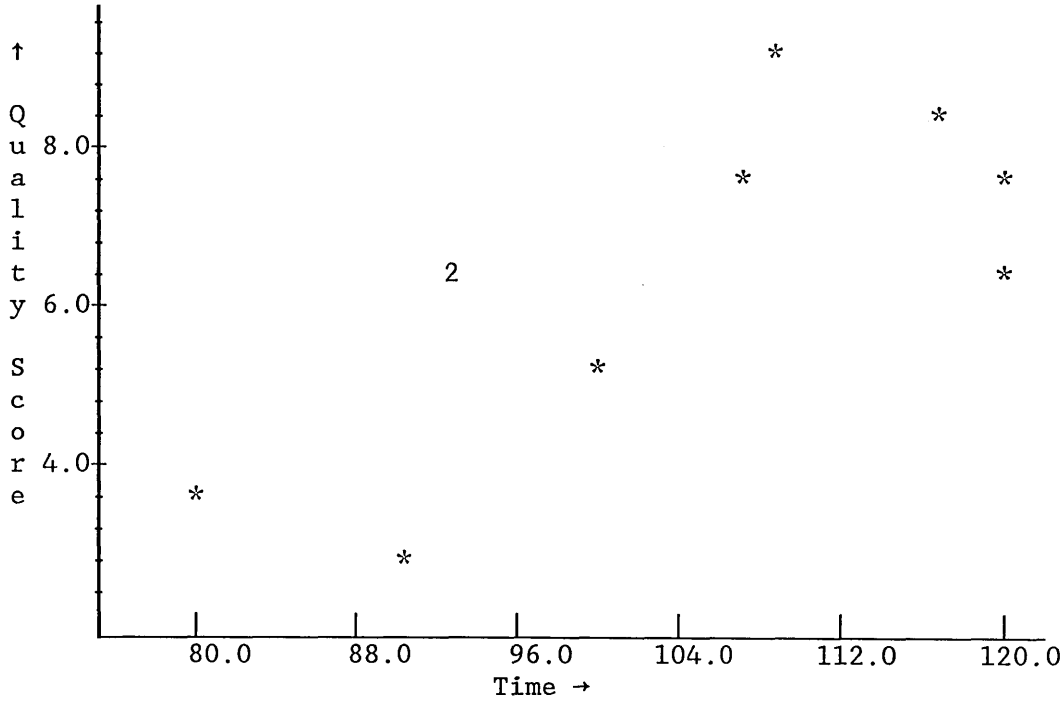


Figure 8.4: Quality vs. Time Taken for the Non-APE Group

Hypothesis 6

H_0: In the non-APE group, there is no correlation between the time taken to complete a task and quality of the solution.

H_a: In the non-APE group, there is a correlation between the time taken to complete a task and quality of the solution.

As can be observed from the graphs, there is no correlation between the time taken to complete the task and the quality of the score for the APE group (correlation = 0.33, $p > 0.05$). Thus Null Hypothesis 5 was accepted. However, there was a significant correlation between the quality and the time taken by users in the screen editor group (correlation = 0.72, $p < 0.05$). Therefore Alternative Hypothesis 6 was accepted.

This result demonstrated that the quality of the solution produced by users in the APE group is independent of the time taken to generate the solution. This could be partly because the range of times spent and qualities produced was so narrow. Hence, the lack of correlation between quality and time spent could, in some instances, be an effect of the uniformity of the product and the time taken.

On the other hand, the quality of the solution of the screen editor group was dependent on the amount of time spent in generating the solution. The results show that those users who took longer produced a better product.

Based on the results, a trade-off can be observed between speed and accuracy in the non-APE group. However, no such effect

can be observed for the APE group. Research has shown that skilled programmers produce consistent results irrespective of the time taken and do not demonstrate any speed accuracy trade-offs [111]. The results of the user trials suggests that the users of the APE system behave as if they were skilled programmers as they exhibit the same consistency irrespective of the time taken.

CHAPTER 9

Conclusions

In this thesis the APE system has been presented, and is a flexible software development system conceptually based on Jackson diagrams. Programs can rapidly be created and manipulated and then transformed automatically into source code. Since control constructs derive directly from the program structures, the only source code required from the user is that concerned with elementary actions and conditions. The user trials of the APE system have demonstrated the acceptability of its human-computer interface and have shown that the system provides productivity gains in software development. Further gains will accrue during the maintenance phase as only program specifications need to be changed, not the code.

The use of diagrams is spreading as an interfacing technique. Diagrams are generally considered to provide a less error prone method of representing information, although they may hide ambiguities in certain cases. The advantage of using diagrams is that they enable a user to work with a two-dimensional representation of a program, instead of a linear text string representation. Specifying a program pictorially exhibits the meaning of a program more clearly and thus results in better understanding and easier modification of the program. This in turn reduces the time and effort for software maintenance. Diagrams offer a higher-level

description of the desired actions, often de-emphasizing issues of syntax and providing a higher level of abstraction. This may be especially true during debugging, where graphics can be used to present more information about the program state, such as current variables and data structures, than is possible with purely textual displays. For these reasons the APE system was specifically designed around an extended set of Jackson structure diagrams and has proved to be a user friendly interface.

9.1. Experiences in Applying Formal Methods

The specification language SML was used to build a mathematical model of the APE system. The short term benefits have been an increased understanding of the system and the detection of some errors in the implementation. By using a mixture of natural language and SML the specification of a module of the APE system is presented that is comprehensive enough to describe the essential aspects of the system's behaviour, but is sufficiently abstract in that it will not burden the reader with the kind of detail that will appear in the implementation. This has offered the possibility of showing the advantages of applying formal methods to capture important aspects of the behaviour of a large system.

One reason for successfully applying formal methods to the APE system was its good modular structure. This made it possible to concentrate on individual modules in relative isolation.

The questions and problems identified by interrogating the specification are important as they highlight ambiguities either in the documentation or in the implementation of the system. This provides system designers and maintainers with valuable feedback on possible problem areas.

Executable specification languages, such as SML, have several uses; allowing dynamic reviews of the design among the design team and, demonstrating the model to potential users to obtain comments, feedback, and clarification of user requirements. A common criticism of these specification languages is that they do not provide the same level of abstraction as, say, VDM or Z. However, these disadvantages can be overcome in SML by generating axiomatic specifications and using modules. Furthermore, SML has the added advantage of being executable (although the axioms cannot be executed), thus enabling the specification to be checked for correctness.

In this thesis, an attempt has also been made at proving mathematically that the SML functions satisfies each of the axioms in the specification. Correctness proofs of two SML functions are presented in Appendix B, one each for a recursive and non-recursive function. These proofs adequately show the derivation of functions from axioms. It is hoped that a future implementation of SML will involve a theorem proving element to check implementations automatically against axioms.

The difficulties encountered in developing a model were in learning the language and in achieving the right level of abstraction. The abstraction level is important since the specifications of the modules of a system should avoid implementation details. In the APE system this was achieved by hierarchically structuring SML modules.

The principal benefit of constructing a formal model is the development of a framework to aid communication between personnel involved with system maintenance. The model can also be used to investigate future changes, and since this framework provides relevant abstraction of user and system behaviour it should facilitate improved documentation and user learning.

9.2. Empirical Evaluation of the APE System

The results presented here concern two issues, namely, getting user responses to evaluate the interface of the APE system and the empirical evaluation of its usefulness as a tool.

Evaluation is crucial in improving design and performance and should therefore pervade the implementation process. Validation should occur in stages as the system develops over time, from a preliminary feasibility demonstration to formal testing of the interface through field tests with subjects solving real world problems.

The questionnaire resulted in a generally positive response to the system, with most answers indicating approval of its

characteristics. This showed that the principle of an animated programming environment was acceptable to users, and that design decisions regarding the user interface were well-founded. After the questionnaire was analysed, changes were made to the system. These included improved error recovery routines, modifications to the on-line help facilities, and extensions to the skeleton tree diagram.

Questionnaires used in evaluating an interface should be carefully prepared as the end users of a system are not always aware of all the features of the interface that influenced their perception of quality. Additionally, problems may exist in attitude measurement because of varying attitudes over time.

There were some advantages and disadvantages in using the questionnaire as an evaluation tool. Specific questions were found to be the most useful as they resulted in a list of areas that needed attention. However, such questions cannot hope to evaluate the entire interface, because this would result in an over-lengthy questionnaire. On the other hand, users with little or no computer experience may be unable to give a critical response, particularly if these responses must be based either on comparison with other systems or on the user's own conception of what the interface should look like. So it was also found inadvisable to ask users to pass judgement on questions that proposed new features to a system, without giving them experience with it. This view suggests that users might not be aware of what constitutes a good system,

but might only be aware of what they did not like when they had to use it.

The results from specific questions indicated how little known some commands were, which is important since an unknown command is of little use in a system. Measuring the users dissatisfaction through these questions also yielded a measure of problem areas in the existing system. This should help in future work of the system's interface.

During the evaluation it was noted that users tended to switch from the menu selection system to the command language. In part this may have been because of the time penalty involved in requesting a display, waiting for it to be displayed, and then searching for the desired item. Also, the large number of commands required the use of multiple menus, which was designed to give help whether or not it was requested. As expertise was developed, the command language was preferred, although it was difficult to learn, and there were no on-line reminders. This supports the contention that a system should have two or more levels of dialogue, appropriate to the user's level of competence.

In the comparative evaluation, two groups of users generated code to solve a problem, using either the APE system or a screen editor. Although the median quality of the solutions was the same for both groups, there was significantly more variability in those produced with the screen editor.

As regards performance, the results show that the median time taken to generate code under the APE system was less in comparison to that taken by a screen editor. In addition, there was a significantly greater variability in the time taken to complete a task with the screen editor. This result was expected as the APE system aims to relieve the programmer of many of the tedious and repetitious tasks in designing, writing, documenting and maintaining programs.

Finally, the correlation between quality and times for each group demonstrates that the APE system imposes some uniformity in quality and in time taken, and there is no indication that spending longer with the APE system results in a better product. However, the screen editor group who produced much more variable products in varying times showed that those who spent longer produced better results. This result suggests the interesting possibility that users of the APE system behave like skilled programmers as they do not show any speed accuracy trade-offs.

These results have demonstrated potential benefits of a structured programming environment. The results also suggest that using such systems helps to produce standard quality products, and reduces the dependence on the experience and ability of the practitioner. This type of comparative evaluation should be extended to larger and more diverse groups of users, where it is possible that differences in performance may be demonstrated.

Clearly, it is time-consuming to carry out such a test, but the results are a surer basis for decisions regarding the introduction of a new system. A questionnaire can provide information helpful for adjusting the interface characteristics to the preferences of users, and in identifying potential areas of difficulty in use. The value of a new system can of course be identified after prolonged use in real applications, but this is not likely to be efficient given the costs of installation and training.

The results of any user trials depend on the user population sampled and the tasks used in the trials. Consequently, the generalisations based on the results of the user trials presented in this thesis require circumspection. Nevertheless, one can draw conclusions about relationships between various characteristics, problems and techniques that should remain valid for other larger samples. Of course, these results need to be investigated further in order to provide confirmation and explain their implications. In this sense, the empirical work described in this thesis raises more questions than it answers, and should provide a starting point for additional research.

Similar criticism applies to the inferences based on the formal model of the APE system, since the correspondence between the model and the system is suggested but is not formally established. This approach although not rigorous does highlight inconsistencies between the specification, implementation and documentation, and provides the designers and implementors with a set of critical

test cases.

9.1. Future Work

Further work remains to be done in the area of visual programming and program visualisation. In particular two areas that need to be more fully addressed are providing support for data abstraction and establishing methods which ensure that dynamics enhance the clarity of visualisation. A more flexible code view and a user-oriented view of data structures and algorithms would be desirable enhancements. Recent work on the graphical simulation of real-time systems could be of significant benefit in enhancing the animation of software systems.

A future aim is to develop and integrate a testing procedure within the APE system. One area currently under investigation is the use of grammars to model the input and output. This will provide a means of generating input test data and predicting expected results without having to resort to the traditional method of solving systems of predicates.

Testing of future versions of the APE system should use a similar controlled evaluation with expert/industrial user groups in respect of its application to document preparation and as a program design aid. In general, it is hoped that the method of design and evaluation, as described in this thesis, will have an application to a range of future projects.

In summary, some areas of further work and research are:

- consideration of the reverse problem starting from existing programs and, hence, automatic structuring, documentation, even translation;
- the development of a tool to verify the consistency of design diagrams;
- the incorporation of a knowledge based system to guide the user through various stages of the development life cycle;
- additional research to assess the effectiveness of user interfaces;
- statistical evaluation of other visually appealing systems with respect to user friendliness and other attributes; and
- the formulation of standards to assist in the design and development of interactive systems.

APPENDIX A

The Curses Library Package

The Curses library is a collection of terminal-independent, low-level input/output functions. These subroutines enable a programmer to do the most common type of terminal dependent functions, such as moving the cursor around on the screen, creating and deleting windows, writing text and seeking to specific window-relative cursor positions.

The curses functions use a terminal's features to the best of a program's abilities by decoding the information in the termcap terminal database. The database contains definitions for the various escape sequences needed to get around on specific terminals. Moreover, these functions talk to terminals efficiently in that they always send the minimum amount of characters necessary to modify the current screen.

A window in curses is purely an internal representation. It provides and store a potential image of a portion of the terminal and does not bear any necessary relation to what is really on the terminal screen.

To update the screen optimally, curses maintains two images of a screen - one of these reflects what the screen currently looks like; the other is a scratch space that a programmer modifies using the various curses functions. When the curses *refresh*

function is invoked, the scratch buffer is synchronised with the terminal screen by sending the minimum number of characters necessary to match the two images. This enables the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Furthermore, changes can be made to these windows in any order without regard to motion efficiency.

The screen package is initialised by a call to *initscr*. This function determines terminal characteristics and allocates space for two windows. One of these (called *curscr* for current screen) is a screen image of what the terminal currently looks like. Another screen (called *stdscr*, for standard screen) is the default scratch pad provided to make changes on.

Several functions are provided to deal with *stdscr* as the default screen. For example the basic functions *addch* and *move*‡, are used to change what will go on a window. The former adds a character at the current (y,x) co-ordinates, returning ERR if it would cause the window to scroll illegally. The *move* function returns ERR if a user tries to move off the window and scrolling is not allowed, otherwise it changes the current (y,x) co-ordinates to whatever the user had in mind. As it is often desirable to first move and then do some input/output operation, most curses functions can be preceded by the prefix 'mv'. The desired (y,x) co-ordinates can then be added to the argument of the function. For example, the calls

‡ The complete list of functions is presented in Arnold [5].

```
move(y, x);
addch(ch);
```

can be replaced by

```
mvaddch(y, x, ch);
```

and

```
wmove(win, y, x);
waddch(win, ch);
```

can be replaced by

```
mvwaddch(win, y, x, ch);
```

After changes have been made to a window, a call is made to *refresh* (or *wrefresh* if the window is not *stdscr*) to make the terminal screen look like the window covered by it. All other higher level output routines such as *addstr* and *printw* call *addch* to add characters to a window.

Input to a window is a mirror image of the output. The complementary function to *addch* is *getch* which, if echo is set, will call *addch* to echo characters as they are typed.

The screen package must know what is on the terminal at all times. If characters are being echoed on the terminal screen, then tty must be in raw or cbreak mode. If not, *getch* sets it to cbreak mode before reading a character.

The curses package gets confused when echo is enabled. The problem here is that the package does not know about any characters that it has not written to the screen itself. Since

characters are echoed by the operating system and not by curses, the package does not know that they are there. As a consequence, when the screen is refresh'ed, curses will not delete the characters that it does not know about and the screen rapidly fills with unwanted and unerasable characters. Therefore, all input and output must be through curses routines.

Before exiting, the clean up routine *endwin* must be called. It restores tty modes to what they were when *initscr* was first called.

APPENDIX B

Correctness Proofs

of Standard ML Functions from its Axioms

In order to ensure that a Standard ML function satisfies its specification it is necessary to prove that the function satisfies each of the axioms in the specification. As examples of program verification, two proofs are presented, one each for a recursive and non-recursive function. These proofs lack rigor but adequately show the derivation of functions from axioms.

B.1. Correctness Proof of a Recursive Standard ML Function

In this section, a proof is presented for obtaining the *head* element (described in Chapter 7) in a 'NODE list'. For convenience it is presented here again:

```
signature NodeSIG =
    sig
        ...          ...
        val head_in_tree: (NODE list) → bool
        axiom head_in_tree (nil) = false
        axiom (∀ NodeList: NODE list)
            => head_in_tree (NodeList) = (∃ n: NODE)
                => extract_node_status (n) = current
                ∧  node_in_tree (NodeList, n)

        val head : (NODE list) → NODE
        axiom (∀ NodeList: NODE list)
            => ¬head_in_tree (NodeList)
                ⇒ head (NodeList) = NULL
        axiom (∀ NodeList: NODE list)
            => head_in_tree (NodeList)
                ⇒ (∃ n: NODE)
                    => head (NodeList) = n
```

```
                    ∧ extract_node_status (n) = current
    end ;
```

An ML function which satisfies this specification is the following:

```
functor NodeFun() : NodeSIG =
    struct
        ...     ...
        ...     ...
        fun head_in_tree (nil) = false
           | head_in_tree ((h::t): NODE list) =
               (extract_node_status (h) = current) orelse
               (head_in_tree (subtree (h))) orelse
               (head_in_tree (t))

        fun head (nil) = NULL
           | head ((h::t): NODE list) =
                if extract_node_status (h) = current then
                   h
                else if head_in_tree (subtree (h)) then
                    head (subtree (h))
                else
                    head (t)
    end ;
```

Since the definition of *head* is recursive, the method of induction is used to prove that *head* satisfies each of the axioms in the specification. The specification makes use of other functions, namely *extract_node_status*, *head_in_tree* and *subtree*. Although a rigorous proof would make reference to the definitions of these functions, it is beyond the scope of this thesis. Instead it will simplify matters slightly if various facts about these functions are used without proving that they follow from the definition. These facts include:

- the function *subtree* returns the siblings of a node

- the function *head_in_tree* returns a boolean which is true if a node whose status is *current* exists in a tree

- the function *extract_node_status* returns the status associated with a node.

Proof A: (head satisfies ¬ head_in_tree (t) ⇒ head (t) = NULL)

We assume ¬head_in_tree (t) and prove by induction that head (t) = NULL)

Base Case: Suppose t = nil. Then head (t) = NULL (by definition).

Step Case: Suppose head (T) = NULL for any T of order p such that ¬head_in_tree (T). We show that for any T′ (where T′ = a::T) of order p+1, ¬head_in_tree(T′) ⇒ head(T′) = NULL.

There are two cases to consider.

Case A.1: (extract_node_status (a) = current)

This case is not possible because if the status of node a was current then head_in_tree (T′) = true and head(T′) = a.

This is a contradiction since head_in_tree(T′) = false implies that there cannot exist a node whose status is current. Hence head (T′) = NULL.

Case A.2: (extract_node_status (a) ≠ current)

In this case head(T′) = head (subtree (a)) (and so head (subtree (a)) = NULL by inductive assumption) or head (T′) = head

(T) (and so head (T) = NULL) by inductive assumption).

□

Proof B (head satisfies head_in_tree (t) ⇒ ∃ a => head (t) = a ∧ extract_node_status (a) = current)

We assume that head_in_tree (t) = true and prove by induction that there exists an A such that head (t) = A and extract_node_status (A) = current

Base Case: Suppose t = A::nil

Then since head_in_tree (t) is true, there exists an A in t such that extract_node_status (A) = current.

Since t has just a single element, A, it must be of status current.

Therefore head (t) = A

Step Case: We suppose that for any T of order p or less head_in_tree (T) = true ⇒ ∃ a => head (T) = a ∧ extract_node_status (a) = current and prove by strong induction that for any T′, where T′ = b::T, of order p+1 head_in_tree (b::T) ⇒ ∃ A => head (b::T) = A ∧ extract_node_status (A) = current.

There are three cases to consider:

Case B.1: (b = current)

In this case head (b::T) = b (by definition)

Thus A = b.

Case B.2: (b ≠ current ∧ head_in_tree (subtree b) = true)

In this case
head (b::T)
= head (subtree b) by definition
= head (TT') where TT' is the subtree of b of order p or less
= A for some A whose status is current by inductive assumption

Case B.3: (b ≠ current ∧ head_in_tree (T) = true)

In this case
head (b::T)
= head (T) by definition
= A for some A by inductive assumption

□

B.2. Correctness Proof of a Non-Recursive Standard ML Function

As a final example, an instance of a non-recursive proof is presented. Once again, the proof lacks complete rigor but adequately shows that the Standard ML implementation supports the suppositions made in the axioms.

The function *paste_to_left*, described in Chapter 7, is proved by demonstrating that the implementation supports the axioms. The specification and Standard ML implementation for the EditSIG signature is given below:

```
signature EditSIG =
    sig
        structure Node: NodeSIG
        structure Make: MakeSIG
        structure Move: MoveSIG
        ...     ...
        datatype SIDE = up | down | left | right
        ...     ...
        val paste: ((Node.NODE list) × Node.NODE × SIDE) →
                    (Node.NODE list)

        val cutit: ((Node.NODE list) × Node.NODE) →
                    (Node.NODE list)
        axiom (∀ n: Node.NODE)
            => cutit (nil, n) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.node_in_list (t, n)
                ⇒ (∃ t': (Node.NODE list))
                    => cutit (t, n) = t'
                        ∧ Make.creat_nodes (t',
                          Node.extract_node_type (n), 1) = t

        val valid_paste: (Node.NODE × Node.NODE) → bool
        axiom (∀ x, y: Node.NODE)
            => valid_paste (x, y) =
                Node.extract_node_type (x) =
                Node.extract_node_type (y)

        val paste_to_left: ((Node.NODE list) × Node.NODE) →
                            (Node.NODE list)
        axiom (∀ t: (Node.NODE list), x: Node.NODE)
            => valid_paste (x, Node.head (t))
                ⇒ (∃ t': (Node.NODE list))
                    => paste_to_left (t, x) = t'
                        ∧ cutit (t', x) = t

        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
        axiom (∀ t, n)
            => Move.root (t) = Node.head (t)
                ⇒ paste (t, n, _) = t
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
```

```
functor EditFun (Node: NodeSIG, Move: MoveSIG, Make: MakeSIG
                 sharing structure Node = Move.Node = Make.Node
                ): EditSIG =
 struct
    structure Node = Node
    structure Move = Move
    structure Make = Make

    datatype SIDE = left | right | bottom | up

    (* check to ensure that the node types are valid for pasting *)
    fun valid_paste (x: Node.NODE_TYPE, y: Node.NODE_TYPE) =
        (x = y) andalso (x <> Node.iteration)

    (* paste x to the left of head in l *)
    fun paste_to_left (l: Node.NODE list, x: Node.NODE) =
        if valid_paste (Node.extract_node_type
           (Node.head (l)), Node.extract_node_type (x))
        then
            Node.replace_list_in_list (l, Node.head (l),
            Node.re_number ([x], Node.last_node_no
            (l, 0) +1) @ [Node.head (l)])
        else
            (print ("Left: this is not a valid paste.\n"); l)

    (* paste node x to the side y of head in l *)
    fun paste (nil, x, y) = nil
      | paste (l: Node.NODE list, x: Node.NODE, left: SIDE) =
            if Node.head (l) <> Node.NULL then
                paste_to_left (l, x)
            else
                (print ("There is no head in this list.\n"); l)

    (* delete node x from the list *)
    fun cutit (nil, x) = nil
      | cutit ((hd :: tl): Node.NODE list, x: Node.NODE) =
            if x = hd then
                tl
            else if Node.node_in_tree(Node.subtree(hd),x)
            then
                let val Node.node (a,b,c,d,e) = hd
                in
                    Node.node (a,b,c,d, cutit (e, x))::tl
                end
            else
                hd :: cutit (tl, x)
 end;
```

- The function *Node.replace_list_in_list(a, b, c)* finds the first instance of a Node.NODE (b) in the 'Node.NODE list'

(a). If such a node exists, it is replaced by the the 'Node.NODE list' (c) and the new list is returned.

- The function *Node.last_node_no(t)* returns the highest generation number associated with a Node.NODE in a 'Node.NODE list' t.

- The function *Node.re_number(t, no)* re-numbers the generation numbers in a 'Node.NODE list' beginning with 'no'.

- The function *Node.node_in_tree(t, n)* is similar to the function *Node.head_in_tree* (specified in the above section). It returns a boolean depending on whether a Node.NODE n exists in a 'Node.NODE list' t.

- The function *Node.extract_node_type(n)* returns the type of a Node.NODE n.

Some facts of various functions are assumed in this proof. In particular it is assumed that the functions *cutit*, *valid_paste* and the *NodeSIG* structure have been defined.

Proof C: (valid_paste(x, Node.head(t)) ⇒ (∃ t' : (Node.NODE list))
=> paste_to_left(t, x) = t' ∧ cutit(t' , x) = t))

Since valid_paste(x, Node.head(t)) is true,

paste_to_left(t, n)
= Node.replace_list_in_list(t, Node.head(t), n)
where n is a 'Node.NODE list' obtained by re-numbering the generation numbers in the 'Node.NODE list' [x] and concatenating

[Node.head(t)] to it.)

= t′

where the number of nodes in t′ = number of nodes in t + number of nodes in [x]

Also, by definition of cutit, cutit (t′, x) = t

□

APPENDIX C

The Questionnaire

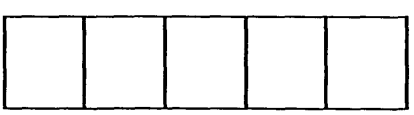
Name _____ Group _____ Date _____

The following questionnaire is used to summarise peoples' attitudes towards using the *Animated Programming Environment* (APE) system.

The questionnaire is in the form of a number of statements with which you can either agree or disagree, ranging from agreement through to full disagreement.

Here is an example:

The system is easy to use.

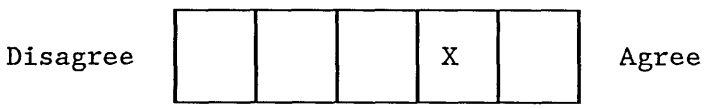
Disagree  Agree

The first and last box on the scale indicate full disagreement/agreement.

The two boxes on either side of the centre box indicate moderate disagreement/agreement.

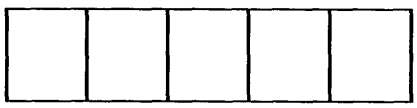
The centre box indicates a neutral position.

If you agree moderately with the statement would you please mark the fourth box (from the left) with a cross ('X').

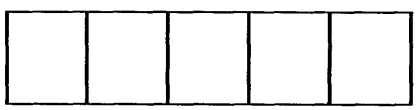


If you do not understand the meaning of a statement please look in the glossary at the end of this questionnaire.

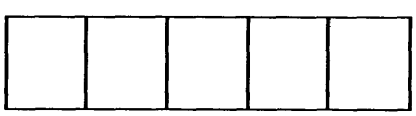
(1) The system is easy to use.

Disagree  Agree

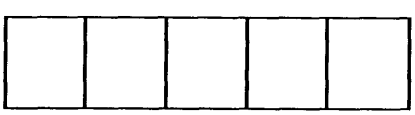
(2) A lot of concentration is required for using the system.

Disagree  Agree

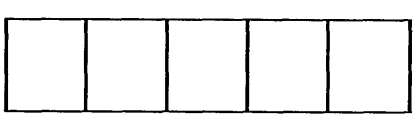
(3) I have previous experience of programming with the Jackson's methodology.

Disagree  Agree

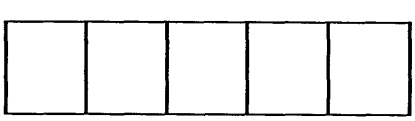
(4) The system is too complicated.

Disagree  Agree

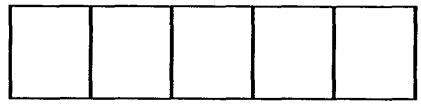
(5) The response time of the graphical interface is fast.

Disagree  Agree

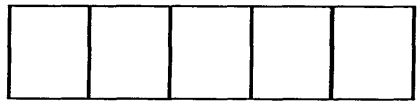
(6) The instructions, on-line help and other accompanied documentation provided with the system are good.

Disagree  Agree

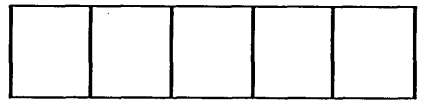
(7) The system allows for quick error correction.

Disagree  Agree

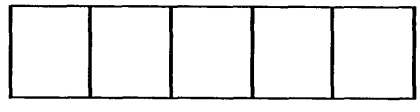
(8) The overall graphical interface is good.

Disagree  Agree

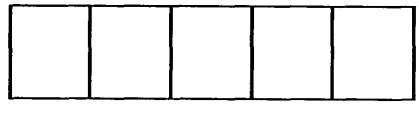
(9) The system will be further enhanced by the proper use of colour.

Disagree  Agree

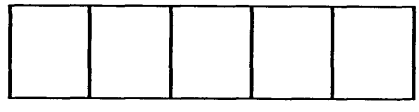
(10) The system menu facilities are satisfactory.

Disagree  Agree

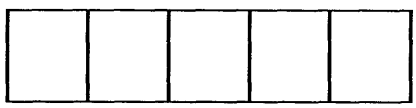
(11) The editing facilities (e.g. cut, paste, copy, etc.) of the system are good.

Disagree  Agree

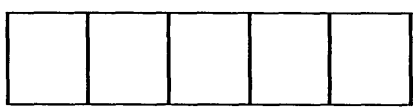
(12) The system displays enough information on the screen.

Disagree  Agree

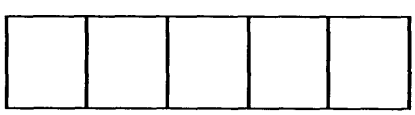
(13) The system's *history* mechanism is easy to understand. (Moderate to full disagreement would indicate you think that the *history* mechanism needs to be modified.)

Disagree  Agree

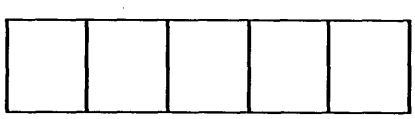
(14) The system was easy to learn to use.

Disagree  Agree

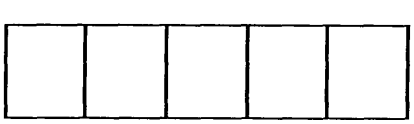
(15) The *learn* program/package of the system is useful.

Disagree  Agree

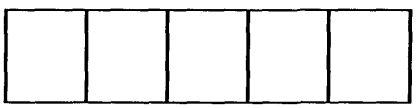
(16) Sufficient information is displayed in a node of a tree.

Disagree  Agree

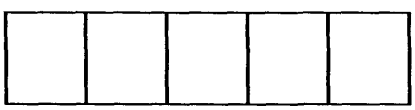
(17) The various tree diagrams provided by the system are useful.

Disagree  Agree

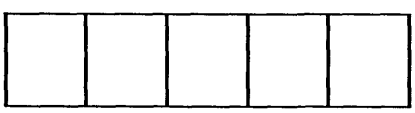
(18) More types of tree diagrams are required, as they will enhance the system's capabilities.

Disagree  Agree

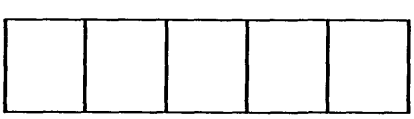
(19) The *overview* tree is useful. (An *overview* tree is one which displays the bare skeleton of the user program and the inter-relations of the current tree with other tree diagrams that are stored in *history*.)

Disagree  Agree

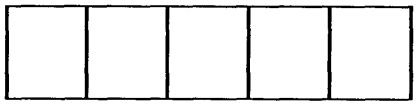
(20) The option provided by the system for changing the size of the default node is useful. (A standard node is the one which appears when you enter the system for the first time.)

Disagree  Agree

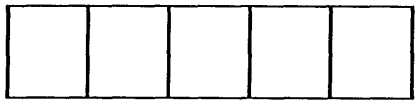
(21) The run-time animation of the tree diagram is useful. (Run-time animation means animating the tree diagram so that when the program runs you can see the cursor move between the nodes when a particular node is executed.)

Disagree  Agree

(22) The number of windows in the system should be reduced.

Disagree  Agree

(23) The size of the text window, needs to be made larger. A text window is where a user inputs program text.

Disagree  Agree

If there is anything else which you would like to add concerning the system in question, or about this questionnaire, please feel free to do so in the space provided below.

Glossary

(1) Refers to the extent to which the user feels at ease with using the system and does not consider it difficult to use.

(2) Refers to the amount of concentrated thought required by a user in using the system.

(3) Refers to the extent to which a user is familiar with the Jackson's method of program design.

(4) Refers to the feeling that the system could have been made much simpler.

(5) Refers to the speed of the system.

(6) Refers to the the help facilities provided by the system. These include on-line help as well as any accompanied documentation.

(7) Refers to the time taken in eliminating error/bug(s) in the task which the user is undertaking.

(8) Refers to the appearance of the graphics which the system produces.

(9) Refers to the degree to which the system's capabilities will improve with the use of colour.

(10) Refers to the usefulness of the system's menu facilities.

(11) Refers to the quality of editing facilities offered by the system.

(12) Refers to the amount of information displayed by the system on the terminal screen.

(13) Refers to the extent to which the system's *history* mechanism is easy to use.

(14) Refers to the complexity involved in learning the system.

(15) Refers to the usefulness of the system's *learn* package.

(16) Refers to the amount of information displayed in a node of the system's tree diagram.

(17) Refers to the usefulness of the different types of tree diagrams provided by the system.

(18) Refers to the feeling that more tree diagrams are required which will enhance the system.

(19) Refers to the usefulness of the overall tree diagram, i.e. the diagram which display's the connections of the current tree with other tree diagrams which are saved in the user's *history* mechanism.

(20) Refers to the usefulness of changing the size of the default node.

(21) Refers to the feeling that the run-time animation of the tree

diagram will enhance the system.

(22) Refers to the feeling that the number of windows in the system should be decreased.

(23) Refers to the to the extent that the small size of the text window hinders the users performance.

APPENDIX D

Questionnaire Analysis

The responses of the subjects to the above questionnaire are tabulated below (figure-D.1) together with the median ratings (figure-D.2).

Subject Responses to the Questionnaire

Subject	1	2	3	4	5	6	7	8	9	10
1	5	2	5	1	5	5	5	5	5	4
2	4	3	1	1	5	2	5	4	4	1
3	4	3	1	3	2	3	3	4	5	4
4	3	5	2	3	4	2	3	4	2	3
5	3	1	2	2	4	4	5	5	5	4
6	2	3	1	3	3	4	4	5	5	4
7	2	4	3	3	5	4	3	5	5	4
8	3	4	1	3	2	5	4	5	2	5
9	4	4	4	2	5	4	5	5	3	3
10	4	4	5	3	5	4	4	4	4	4
11	4	5	4	2	5	5	3	5	5	5
12	4	4	1	3	4	4	5	5	1	4
13	3	5	1	3	4	5	4	5	2	5
14	4	3	1	2	4	3	4	4	2	4
15	3	4	5	4	5	4	4	4	3	4
16	4	5	1	1	5	5	4	5	5	5
17	5	3	1	1	5	5	3	4	2	5
18	5	3	5	1	5	5	4	5	5	5
19	4	3	4	3	4	3	2	4	5	3
20	4	4	5	2	4	3	3	5	4	4
21	4	5	1	1	2	4	1	4	5	5
22	4	5	1	2	4	4	4	4	3	3
23	4	2	4	1	4	5	2	4	2	3
24	4	2	5	2	5	4	3	5	1	4
25	2	5	1	4	5	5	2	5	1	4
26	2	4	3	3	5	4	2	4	5	2
27	3	4	1	2	5	4	3	4	3	5
28	4	2	4	2	4	4	5	4	2	5
29	4	5	2	2	5	3	2	5	3	4
30	2	5	1	3	3	4	2	5	3	4
31	4	4	1	2	4	5	4	5	5	4
32	4	3	1	2	4	4	2	4	3	4

Subject Responses to the Questionnaire

Subject	1	2	3	4	5	6	7	8	9	10
33	4	4	1	2	5	5	4	5	5	5
34	4	5	1	1	4	4	1	5	5	5
35	2	5	1	4	4	4	2	4	5	3
36	4	2	1	2	5	5	3	4	3	4
37	3	3	4	2	5	4	2	4	2	4
38	4	3	3	5	5	4	4	5	4	5
39	4	5	1	1	5	5	3	5	5	5
40	2	5	3	3	5	2	1	5	5	2
41	3	3	3	3	3	4	2	3	5	3
42	3	4	5	1	5	4	2	4	2	5
43	3	4	2	3	3	3	4	3	3	4
44	1	5	1	1	2	2	1	4	1	5
45	4	5	1	2	5	2	2	5	5	4

Subject Responses to the Questionnaire

Subject	11	12	13	14	15	16	17	18	19	20	21	22	23
1	5	5	5	4	5	4	5	5	5	5	5	1	3
2	2	1	2	5	5	5	5	3	4	3	4	2	2
3	3	4	2	4	4	5	4	4	5	4	3	2	4
4	3	2	4	2	3	4	4	4	4	3	3	2	4
5	5	3	4	2	4	4	5	3	5	5	5	1	2
6	3	4	3	2	3	3	4	3	5	3	4	2	3
7	4	4	2	2	3	4	4	4	5	5	5	2	3
8	3	1	4	5	3	5	5	1	5	3	3	2	2
9	4	4	3	4	3	3	4	3	3	4	4	2	2
10	3	4	5	4	5	4	4	2	5	4	4	3	3
11	5	5	4	4	5	4	5	4	5	3	3	3	2
12	4	3	4	3	4	4	5	2	5	4	4	3	4
13	5	4	3	4	5	3	5	3	5	5	4	2	5
14	4	2	4	2	4	4	4	2	5	5	4	4	5
15	3	3	3	4	3	4	3	3	5	3	3	2	3
16	4	5	4	5	5	5	4	3	3	5	5	4	1
17	5	2	4	5	3	4	4	2	4	3	5	1	4
18	5	5	4	5	4	5	5	4	5	5	5	1	4
19	4	4	3	5	4	3	4	3	5	4	5	1	3
20	4	3	4	4	4	4	2	5	4	5	4	1	2
21	1	2	5	5	5	2	5	5	5	4	5	1	4
22	5	4	4	4	3	4	5	3	3	4	3	1	3
23	5	3	3	4	3	3	4	3	3	3	3	3	2
24	5	4	4	5	3	5	5	3	4	5	4	3	4
25	3	3	5	2	5	2	5	2	5	5	5	2	4
26	3	4	5	1	4	4	4	3	5	5	5	2	3
27	4	3	5	4	3	3	5	2	5	5	5	1	1
28	5	3	2	3	5	3	5	2	5	5	5	1	5
29	3	2	5	1	4	2	4	1	4	5	5	3	4

Subject Responses to the Questionnaire

Subject	11	12	13	14	15	16	17	18	19	20	21	22	23
30	5	5	5	1	3	5	5	3	5	4	5	3	4
31	3	4	4	4	3	2	5	4	4	4	5	2	3
32	4	4	3	4	4	4	5	3	5	5	5	2	2
33	5	2	4	4	3	2	5	1	5	3	4	1	1
34	5	5	3	5	4	5	5	4	3	3	3	1	1
35	3	2	3	1	4	2	4	4	4	4	3	3	4
36	5	5	4	5	5	5	5	4	5	3	3	4	2
37	3	4	3	4	5	5	5	3	3	4	3	3	2
38	4	4	4	5	5	4	5	2	5	1	5	1	3
39	4	2	5	4	5	5	5	4	5	3	5	1	5
40	2	2	3	1	3	1	3	4	5	5	5	2	5
41	2	3	4	3	4	3	4	2	5	4	5	3	3
42	5	4	4	4	3	5	5	2	4	2	5	1	1
43	3	4	4	4	4	2	3	3	5	4	5	2	1
44	1	1	3	2	5	1	4	3	5	3	4	3	5
45	4	2	4	4	4	1	4	1	4	5	4	2	3

Figure D.1: User Responses to the Questionnaire

Median Ratings

Question No.	MEDIAN
1	4.0
2	4.0
3	1.0
4	2.0
5	5.0
6	4.0
7	3.0
8	5.0
9	4.0
10	4.0
11	4.0
12	4.0
13	4.0
14	4.0
15	4.0
16	4.0
17	5.0
18	3.0
19	5.0
20	4.0
21	4.0
22	2.0
23	3.0

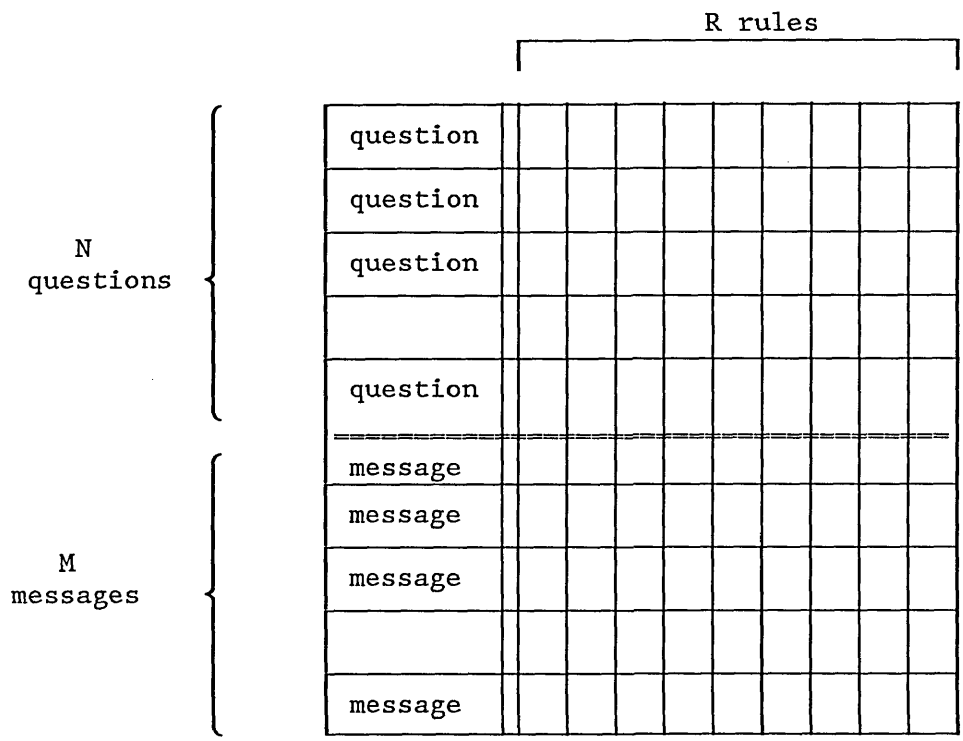
Figure D.2: Median Ratings

APPENDIX E

Decision Table Implementation

Overview

Design a detailed pseudocode algorithm to implement a restricted form of decision table:



The data for such a decision table is held in a file in the following sequence:

```
N,                              /* No. of questions */
M,                              /* No. of messages  */
question, ..., question,        /* N questions      */
message, ..., message,          /* M messages       */
R,                              /* No. of rules     */
```

```
        { outcome, ..., outcome,   exec flag, ..., exec flag,
          ^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^^^^^^^^^^
  R     Question outcome rule 1        Actions to perform
Times                                  if rule 1 selected
        { outcome, ..., outcome,   exec flag, ..., exec flag,
        { ...
        { outcome, ..., outcome,   exec flag, ..., exec flag,
```

Example

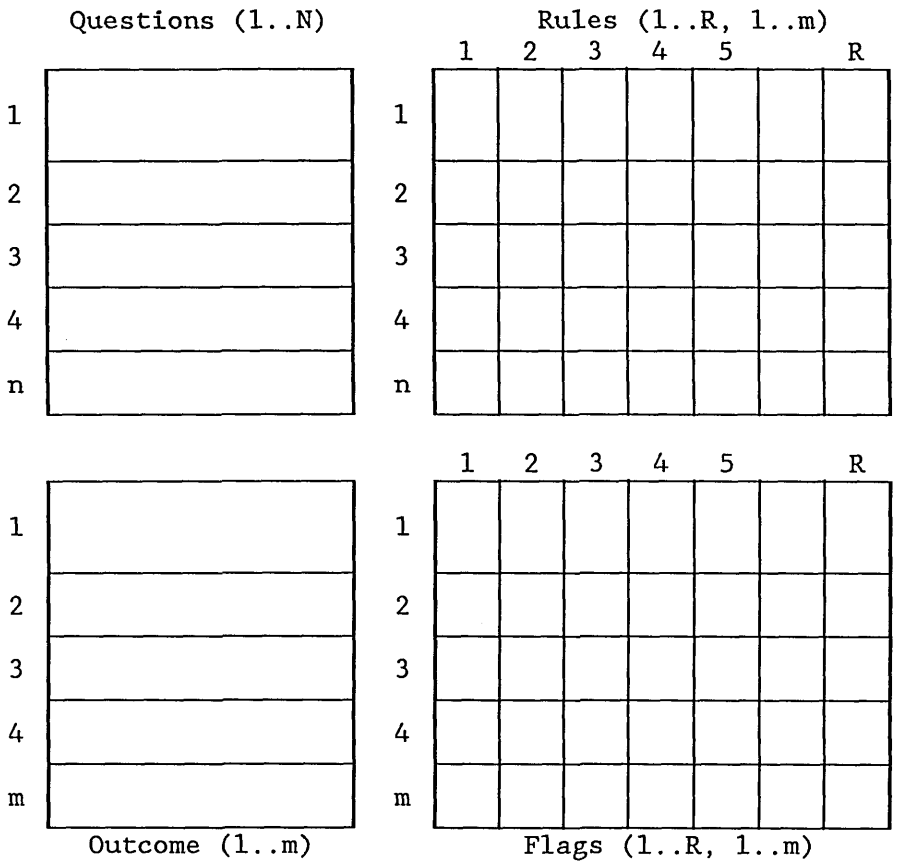
Age 17?	N	Y	Y	Y	Y	Y
O Grade Maths?	-	N	Y	N	Y	Y
O grade English?	-	N	N	Y	Y	Y
>2 Highers?	-	-	-	-	N	Y
Wait till 17	X					
Resit Maths		X		X		
Resit English		X	X			
Apply for HND					X	
Apply for Degree						X

```
4, 5, 'Age 17', 'O grade Maths?', 'O grade English?', '>2
     Highers?', 'Wait till 17', 'Resit Maths', 'Resit English',
     'Apply for HND', 'Apply for Degree',

     6, 'N', '-', '-', '-', 'X', ' ', ' ', ' ', ' ',
        'Y', 'N', 'N', '-', ' ', 'X', 'X', ' ', ' ',
        'Y', 'Y', 'N', '-', ' ', ' ', 'X', ' ', ' ',
        'Y', 'N', 'Y', '-', ' ', 'X', ' ', ' ', ' ',
```

'Y', 'Y', 'Y', 'N', ' ', ' ', ' ', 'X', ' ',
'Y', 'Y', 'Y', 'Y', ' ', ' ', ' ', ' ', 'X'

The information from the file will be read into a set of four arrays:



The pseudocode algorithm you produce should go through the stages of reading the data into the arrays from the file and asking the users questions and from the responses determining which Outcome message to print.

Four pseudocode algorithms should be formed using the conventional pseudocode structuring constructs (IF..THEN, BEGIN..END, etc) and the following list of primitives:

File Input
- Read (Question (x))
- Read (Outcome (x))
- Read (Rules (x, y))
- Read (Flags (x, y))

Screen Output
- Print (Question (z))
- Print (Outcome (z))

You may choose to store the results of asking the questions in a working array Responses (1..N).

Remember, start the process in a top-down manner and refine each section until you are putting detail in at the level of these primitives alone.

References

1. Abrial, J. -R., "The Specification Language Z: Basic Library", Oxford University Programming Research Group internal report, April 1980.

2. Alexander, Heather, "Formal Specification and Rapid Prototyping Techniques for Human-Computer Interaction", Technical Report TR. 26, University of Stirling, Stirling, Scotland, August 1985.

3. Allman, Eric and Ingres, Project, "An Introduction to the Source Code Control System", in *The UNIX Programmer's Manual*, vol. 2C, University of California, Berkeley, CA 94720, 1980.

4. Archibald, J. L., Leavenworth, B. M., and Power, L. R., "Abstract Design and Program Translator: New Tools for Software Design", *IBM Systems Journal*, vol. 22, no. 3, pp. 170-187, 1983.

5. Arnold, Kenneth C. R., "Screen Updating and Cursor Movement Optimization: A Library Package", in *The UNIX Programmer's Manual, Vol. 2C*, Computer Science Division, Department of Electrical Engineering and Computer Studies, University of California, Berkeley, California 94720, August 1983.

6. Baecker, Ron, "Sorting out Sorting", 16mm colour, sound film, 25min, Dynamics Graphics Project, Computer Systems Research Institute, University of Toronto, Toronto, Ontario, Canada,

1981.

7. Beichter, Freidrich W., Herzog, Otthein, and Petzsch, Heiko, "SLAN-4 – A Software Specification and Design Language", *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, pp. 155-162, March 1984.

8. Benbasat, Izak, Dexter, Albert S., and Masulis, Paul S., "An Experimental Study of the Human/Computer Interface", *Communications of the ACM*, vol. 24, no. 11, pp. 752-762, ACM, November 1981.

9. Benest, I. D. and Took, R. K., "Interactive techniques in software engineering environments", *Computing and Control Division Colloquium on Advanced Workstations II: Reviews and Exhibition*, pp. 3/1-3/15, IEE, Savoy Place, London WC2R 0BL, 19 November 1985.

10. Bergland, G. D., "A Guided Tour of Program Design Methodologies", *IEEE Computer*, pp. 13-37, IEEE, October 1981.

11. Bjørner, Dines and Jones, Cliff B., *Formal Specification and Software Development*, Prentice-Hall International Series in Computer Science, Prentice-Hall International, Inc., Englewood Cliffs, USA, 1982.

12. Blum, M. T., "Information Systems Development", in *Automated Tools for Information Systems Design*, ed. H. -J. Schneider and A. I. Wasserman, North Holland, Amsterdam, 1982.

13. Booch, G., "Object-Oriented Design", in *Tutorial: Software Design Techniques*, pp. 420-436, IEEE Computer Society Press, Washington D.C., 1983.

14. Brown, Gretchen P., Carling, Richard T., Herot, Christopher F., Kramlich, David A., and Souza, Paul, "Program Visualization: Graphical Support for Software Development", *IEEE Computer*, vol. 17, no. 3, pp. 27-35, IEEE, August 1985.

15. Brown, Marc H. and Sedgewick, Robert, "A System for Algorithm Animation", *Computer Graphics*, vol. 18, no. 3, pp. 177-186, ACM, July 1984.

16. Caine, Stephen H. and Gordon, E. Kent, "PDL - A tool for software design", *Proceedings of 1975 National Computer Conference*, vol. 44, pp. 272-276, 1975.

17. Campbell, R. H., "SAGA: A project to automate the management of software production systems", *Proceedings of a Conference on Software Engineering Environments*, pp. 182-201, IEE, London, 1986.

18. Carbonell, J. R., Elkind, J. I., and Nickerson, R. S., "On the psychological importance of time in a timesharing system", *Human Factors*, vol. 10, no. 2, pp. 135-142, 1968.

19. Cardelli, Luca, "ML under UNIX", Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1984.

20. Clarisse, Olivier and Chang, Shi-Kuo, "VICON: A Visual Icon Manager", *Visual Languages*, pp. 151-190, Plenum Press, New York, 1986.

21. Cuff, R. M., "On casual users", *Internation Journal of Man-Machine Studies*, vol. 12, pp. 163-187, 1980.

22. Dahl, O. -J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming*, Academic, London, 1972.

23. Date, C. J., *An Introduction to Database Systems, Volume I, Fourth Edition*, Addision-Wesley, Reading, Mass., 1986.

24. DeRemer, Frank and Kron, Hans H., "Programming-in-the-Large Versus Programming-in-the-Small", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, pp. 80-86, June 1976.

25. Du Boulay, B., O'Shea, T., and Monk, J., "The black box inside the glass box: presenting concepts to novices", *International Journal of Man-Machine Studies*, vol. 14, pp. 237-249, 1981.

26. Edwards, Jack L., "Engineering Advanced Human-Computer Interaction: A Methodology for Design and Evaluation", *Proceedings of IEEE Electronicom'85*, vol. 2, pp. 398-401, IEEE, New York, USA, 1985.

27. Eisensdadt, Marc and Brayshaw, Mike, "AORTA Diagrams As An Aid To Visualising The Execution Of Prolog Programs", *International State of the Art Seminar on Graphics Tools for*

Software Engineering: *Visual Programming* & *Program Visualisation*, p. 19, The British Computer Society, London, 16 March 1988.

28. Everhart, C. R., "A Unified Approach to Software System Engineering", *Proceedings of COMPSAC'80*, IEEE Computer Society, Los Alamitos, Calif., October 1980.

29. Fairley, R. E., *Software Engineering Concepts*, McGraw-Hill, New York, 1985.

30. Fitter, M., "Towards more "natural" interactive systems", *International Journal of Man-Machine Studies*, vol. 11, pp. 339-350, 1979.

31. Foley, James D., "The Design and Evaluation of User Interfaces", Technical Report GWU-11ST-81-07, pp. 225-228, Department of Electrical Engineering and Computer Science, The George Washington University, Washington D.C. 20052, 1981.

32. Freeman, P. and Wasserman, A. I., *Tutorial*: *Software Design Techniques*, IEEE Computer Society Press, Washington, DC, 1983.

33. Gane, C. and Sarson, T., *Structured Systems Analysis*: *Tools and Techniques*, IST Databooks, New York, 1977.

34. Gilbert, P., *Software Design and Development*, Chicago: SRA, 1983.

35. Grafton, Robert B., "Visual Programming", *IEEE Computer*, vol. 17, no. 3, pp. 11-25, IEEE, August 1985.

36. Groebner, David F. and Shannon, Patrick W., *Business Statistics: A Desision-Making Approach, 2nd ed*, Charles E. Merrill Publishing Company, USA, 1985.

37. Hansen, Wilfred J., "User Engineering Principles for Interactive Systems", *Proceedings of 4th FIPS Fall Joint Conference*, vol. 39, pp. 523-532, 1971.

38. Harper, Robert, MacQueen, David, and Milner, Robin, "Standard ML", Internal Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh (UK), March 1986.

39. Hayes, Ian, *Specification Case Studies*, Prentice-Hall International Series in Computer Science, Prentice-Hall International, Inc., UK, 1987.

40. Heerjee, Kaizad B., Miller, Colin J., Samson, William B., and Swanston, Michael T., "The Design, Validation and Evaluation of a Software Development Environment", Submitted for publication to *Software Engineering Journal*, 1988.

41. Heerjee, Kaizad B., "The Formal Specification of an Animated Programming Environment in Standard ML", Technical Bulletin No. 14, Dundee Institute of Technology, Dundee, UK, 1988.

42. Heerjee, Kaizad B., Swanston, Michael T., Miller, Colin J., and Samson, William B., "The Design and Evaluation of an

Animated Programming Environment", *People and Computers IV, Proceedings of the HCI '88 Conference of the British Computer Society Human-Computer Interaction Specialist Group*, Cambridge University Press, Manchester (UK), September 1988.

43. Heerjee, Kaizad B., Miller, Colin J., and Swanston, Michael T., "Experiences in Applying Formal Methods to Existing Software", *Proceedings of a Formal Methods Workshop*, Teeside Polytechnic, Cleveland (UK), July 1988.

44. Heerjee, Kaizad B., *A Survey of Diagraming and Visual Programming Techniques and Tools*, Dundee College of Technology, Dundee, UK, 1988. Under Publication

45. Heerjee, Kaizad B., "Doctype and Sped: UNIX Text Processing Tools", Submitted for publication to *Dr. Dobb's Journal*, 1988.

46. Heerjee, Kaizad B., "APE Programmer's Manual", Technical Bulletin No. 12, Dundee College of Technology, Dundee, UK, 1988.

47. Heerjee, Kaizad B., "APE - An Animated Programming Environment", Technical Bulletin No. 13, Dundee College of Technology, Dundee, UK, 1988.

48. Heerjee, Kaizad B. and Sadeghi, Rubik, "Rapid Implementation of SQL - A Case Study Using YACC and LEX", *Information and Software Technology*, vol. 30, no. 4, pp. 228-236, Butterworth & Co. (Publishers) Ltd., Guildford, England, May 1988.

49. Hoare, C. A. R., "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, ACM, August 1978.

50. Hoare, C. A. R., "Programming is an Engineering Profession", Technical Monograph No. 27, Oxford University Programming Research Group, 1982.

51. IBM,, "HIPO - A Design Aid and Documentation Technique", Document GC20-1851, Data Processing Division, IBM Corporation, White Plains, NY, 1974.

52. Jackson, M. A., *Principles of Program Design*, Academic Press, London, 1975.

53. Jones, C. B., *Software Development*, Prentice-Hall International, 1980.

54. Jones, C. B., *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

55. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs., New Jersey, 1978.

56. Kiger, John I., "The depth/breadth trade-off in the design of menu-driven user interfaces", *International Journal of Man-Machine Studies*, vol. 20, pp. 201-213, 1984.

57. Kilgour, A. C. and Earnshaw, R. A., "Graphics Tools for

Software Engineering: Visual Programming & Program Visualisation", *International State of the Art Seminar*, The British Computer Society, London, 16 March 1988.

58. Knuth, E. and Neuhold, E. J., *Program Specification: Proceedings of a Workshop*, Lecture Notes in Computer Science, 152, Springer-Verlag, New York, 1983.

59. Kunth, D. E., *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

60. Leintz, B. P. and Swanson, E. B., *Software Maintenance Management*, Addison Wesley, Reading, Mass., 1980.

61. Liebelt, Linda S., McDonald, James E., Stone, Jim D., and Karat, John, "The Effect of Organization on Learning Menu Access", *Proceedings of Human Factors Society – 26th Annual Meeting*, pp. 546-550, 1982.

62. Lindley, D. V. and Scott, W. F., *New Cambridge Elementary Statistical Tables*, Cambridge University Press, UK, 1984.

63. Liskov, B., "A Design Methodology for Reliable Software Systems", *Proceedings of the AFIPS 1972 FJCC*, vol. 41, no. Part 1, pp. 191-200, 1972.

64. Lu, P. M., Yau, S. S., and Hong, W., "A formal methodology using attributed grammars for multiprocessing system software development, Part I: Design representation; Part II: Validation", *J. Inform. Sci.*, vol. 30, pp. 79-123, 1983.

65. Lund, Michelle A., "Evaluating the User Interface: The Candid Camera Approach", *Proceedings of CHI'85 – Human Factors in Computer Systems*, pp. 107-113, ACM, 1985.

66. MacQueen, D. B., "Modules for Standard ML", *Proceedings of 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, 1984.

67. Macaulay, L. A. and Norman, M. A., "Designing Interfaces for Different Types of Users - Experimental Criteria", *Proceedings of INTERACT'84 – First IFIP Conference on Human-Computer Interaction*, vol. 1, pp. 643-647, Elseiver Science Publishers B.V. (North-Holland), 1985.

68. Martin, James, *Program Design which is Provably Correct*, Savant Press, Carnforth, Lancs., 1982.

69. Miller, Dwight P., "The depth/breadth tradeoff in heirarchical computer menus", *Proceedings of Human Factors Society – 25th Annual Meeting*, pp. 296-300, 1981.

70. Miller, L. A. and Thomas, J. C., "Behavioural Issues in the use of Interactive Systems", *Internation Journal of Man-Machine Studies*, vol. 9, pp. 509-536, 1977.

71. Mills, H. D., "Mathematical Foundation for Structured Programming", Technical Report FSC 72-6012, IBM Federal Systems Division, Gaithersburg, MD, 1972.

72. Milner, Robin, "The Standard ML Core Language", Report CSR-168-84, Department of Computer Science, University of Edinburgh, Edinburgh (UK), 1984.

73. Milner, R. G., "A theory of type polymorphism in programming", *Journal of Computer and System Sciences*, vol. 17, pp. 348-375, 1978.

74. Myers, Brad A., "INCENSE: A System for Displaying Data Structures", *Computer Graphics*, vol. 17, no. 3, pp. 115-125, IEEE, August 1985.

75. Myers, Brad A., "State of the Art in Visual Programming & Program Visualisation", *International State of the Art Seminar on Graphics Tools for Software Engineering: Visual Programming & Program Visualisation*, pp. 1-24, The British Computer Society, London, 16 March 1988.

76. Nassi, I. and Shneiderman, B., "Flowchart Techniques for Structured Programming", *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, August 1973.

77. Nickerson, R. S., "Man-computer interaction: A challenge for human factors research", *IEEE Transactions on Man-Machine Systems (pt. 2)*, vol. MMS-10, no. 4, pp. 164-180, December 1969.

78. Norman, Donald A., "Design Principles for Human-Computer Interfaces", *Proceedings of CHI'83 - Human Factors in Comput-*

ing Systems, pp. 1-10, ACM, New York, December 1983.

79. Oppenheim, A. N., *Questionnaire Design and Attitude Measurement*, Gower Publishing Company Limited, Aldershot, England, 1986.

80. Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, ACM, 1972.

81. Paterson, James L., *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs., 1981.

82. Poulson, D. F., "Towards Simple Indices of the percived quality of software interfaces", *Computing and Control Division Colloquium on Evaluation Techniques For Interactive Systems Design: I*, pp. 3/1-3/4, IEE, Savoy Place, London WC2R OBL, 27 March 1987.

83. Pressman, R. S., *Software Engineering: a practitioner's approach*, McGraw-Hill, 1982.

84. Raeder, Georg, "A Survey of Current Graphical Programming Techniques", *IEEE Computer*, vol. 17, no. 3, pp. 11-25, IEEE, August 1985.

85. Reingold, Edward M. and Tilford, J. S., "Tidier Drawing of Trees", *IEEE Transactions on Software Engineering*, vol. 7, pp. 223-228, 1981.

86. Rochkind, Marc J., "The Source Code Control System", *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364-370, December 1975.

87. Ross, Douglas T., "Structured Analysis (SA): A language for communicating ideas", *IEEE Transactions on Software Engineering*, vol. SE-3, pp. 16-34, IEEE, January 1977.

88. Sannella, Donald and Tarlecki, Andrzej, "Program Specification and Development in Standard ML", *Conference Proceedings on Principles of Programming Languages*, pp. 67-77, ACM, New Orleans, 1985.

89. Sannella, Donald, "Formal specification of ML programs", Report, pp. 1-19, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh (UK), 1986.

90. Shaw, A. C., "A model for document preparation systems", Technical Report 80-04-02, Department of Computer Science, University of Washington, Seattle, April, 1980.

91. Shneiderman, Ben, *Software Psychology: Human Factors in Computer and Information Systems*, Little, Brown and Co., Boston, MA, 1980.

92. Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, 1987.

93. Shneiderman, B., "Human factors experiments in designing interactive systems", *IEEE Computer*, pp. 9-19, IEEE, December 1979.

94. Siegel, Sidney, *Nonparametric Statistics for the Behavioural Sciences*, McGraw-Hill Book Company, Inc., Tokyo, 1956.

95. Smith, David C., *Pygmalion: A Computer Program to Model and Stimulate Creative Thoughts*, p. 187, Birkhauser, Basel, Stuttgart, 1977.

96. Smith, Sidney L. and Mosier, Jane N., "Design Guidelines for User-System Interface Software", Report ESD-TR-84-190, p. 448 pages, The Mitre Corp., Bedford, MA, September 1984.

97. Staunstrup, J., *Program Specification: Proceedings of a Workshop*, Lecture Notes in Computer Science, 134, Springer-Verlag, New York, 1982.

98. Stay, J. F., "HIPO and Integrated Program Design", *IBM Systems Journal*, vol. 15, no. 2, pp. 143-154, IBM, 1976.

99. Stephens, S. A. and Tripp, L. L., "Requirements Expression and Verification Aid", *Proceedings of the 3rd International Conference on Software Engineering*, pp. 101-108, IEEE, New York, 1978.

100. Supowit, Kenneth J. and Reingold, Edward M., "The Complexity of Drawing Trees Nicely", *Acta Informatica*, vol. 18, no. 4, pp. 377-392, Germany, 1983.

101. Teichrow, D. and Hershey, E. A., "PSL/PSAA Computer Aided Methods of Systems Analysis and Specification", *IEEE Transactions on Software Engineering*, vol. 3, no. 1, pp. 41-48, 1977.

102. Teiltelbaum, T. and Reps, T., "The Cornell Program Synthesiser: A Syntax Directed Programming Environment", *Communications of the ACM*, vol. 24, no. 9, pp. 563-573, 1981.

103. Telecom, British, *Kindra*, British Telecom Research Laboratories, Martlesham Heath, England, 1986.

104. Thompson, K. and Ritchie, D. M., *UNIX Programmer's Manual*, Bell Laboratories, 1978. Seventh Edition.

105. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System", *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Japan, September 1982.

106. Tichy, Walter F., "A Data Model for Programming Support Environments", in *Automated Tools for Information System Design*, ed. H. -J. Schneider and A. I. Wasserman, pp. 31-48, IFIP, 1982.

107. Tse, T. H., "An Automation of Jackson's Structured Programming", *The Australian Computer Journal*, vol. 17, no. 4, November 1985.

108. Warnier, J., *Logical Construction of Systems*, pp. 11-38, Van Nostrand Reinhold Company, New York, 1981.

109. Wasserman, Anthony I., "The Design of "Idiot-Proof" Interactive Programs", *Proceedings of AFIPS National Computer Conference*, vol. 42, pp. M34-M38, 1973.

110. Wasserman, Anthony I., "Information System Design Methodology", *Journal of the American Society for Information Science*, pp. 5-24, John Wiley & Sons, Inc., January 1980.

111. Welford, A. T., *Fundamentals of Skill*, Methuen and Co. Ltd., London, 1968.

112. Wetherell, C. and Shannon, A, "Tidy Drawing of Trees", *IEEE Transactions on Software Engineering*, vol. 5, pp. 514-520, 1979.

113. Wikstrom, Ake, *Functional Programming Using Standard ML*, Series in Computer Science, Prentice-Hall International (UK) Ltd, 1987.

114. Wirth, N., "The Programming Language Pascal", *Acta Informatica*, vol. 1, no. 1, pp. 35-63, 1971.

115. Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, pp. 221-227, ACM, 1971.

116. Wirth, N., *Algorithms + Data Structures = Programs*,

Prentice-Hall, Englewood Cliffs, NJ, 1976.

117. Wood-Harper, A. T., Antill, L., and Avison, D. E., *Information Systems Definition: The Multiview Approach*, Blackwell Scientific, Oxford, 1985.

118. Woody, C. A., Fitzgerald, M. P., Scott, F. J., and Powers, D. L., "A Subject-Content-Retriever-for-Processing-Information-On-Line (SCORPIO)", *AFIPS Conference Proceedings*, vol. 46, 1977.

119. Yamamoto, Y., "An Approach to the Generation of Software Life Cycle Support Systems", Ph.D. Thesis, University of Michigan, Ann Arbor, Michigan, 1981.

120. Yau, Stephen S. and Grabow, Paul C., "A Model for Representing Programs Using Hierarchical Graphs", *IEEE Transactions on Software Engineering*, vol. SE-7, no. 6, pp. 556-574, November 1981.

121. Yau, Stephen S., Yang, Chen-Chan, and Shatz, Sol M., "An Approach to Distributed Computing System Software Design", *IEEE Transactions on Software Engineering*, vol. SE-7, no. 4, pp. 427-436, July 1981.

122. Yau, S. S. and Tsai, J. P., "A Graph Description Language for Large-Scale Software Specification in a Maintenance Environment", *Proceedings of COMPSAC'84*, pp. 397-407, IEEE, November 1984.

123. Yau, Stephen S. and Tsai, Jeffrey J., "A Survey of Software Design Techniques", *IEEE Transactions on Software Engineering*, vol. SE-12, no. 6, pp. 713-721, June 1986.

124. Yourdon, E. and Constantine, L. L., in *Structured Design*, Yourdon Inc., New York, 1975.

The Design and Evaluation of an Animated Programming Environment†

Kaizad B. Heerjee
Michael T. Swanston
Colin J. Miller
William B. Samson

Dundee College of Technology
Bell Street, Dundee
Scotland

ABSTRACT

APE, an Animated Programming Environment, is an interactive, graphical, program design and development system, that embodies structured programming and top-down design. The system supports the development of programs for a variety of block structured languages whilst working conceptually at the level of Jackson diagrams. Evaluation of APE has been carried out during the design and implementation stages of the development life-cycle. The evaluation was based on responses to a questionnaire and a comparison with conventional means of generating code. The questionnaire evaluation elicited users' general impressions about the system and its interface, and their detailed views on more specific aspects of the system. The comparative evaluation showed no difference in the mean quality of the solution to a programming problem, but a significantly reduced variance in quality compared to conventional methods.

Keywords: Human-Computer Interface, Software Evaluation, Programming Environments

---

† To appear in the Proceedings of HCI '88. Conference of the British Computer Society Human-Computer Interaction Specialist Group, Manchester, September 1988.

The Design and Evaluation of an Animated Programming Environment†

Kaizad B. Heerjee
Michael T. Swanston
Colin J. Miller
William B. Samson

Dundee College of Technology
Bell Street, Dundee
Scotland

1. Introduction

Researchers in the field of human-computer interaction have long held the view that well designed interactive systems increase performance levels over conventional techniques, and intuitively this seems logical. Increasing effort is being devoted to the development of interactive systems, but their positive impact is seldom evaluated. Although lists of goals have been proposed [10,3,13,11], to assist the design of interactive systems, they can often be criticised for being contradictory or too imprecisely stated to be defined and measured.

The interactive system evaluated in this study was the Animated Programming Environment, APE. The characteristics of the system are described, followed by a description of the method used for evaluation and its results.

2. Overview of the APE System

APE is an interactive, graphical, program and text design and development system which enables users to develop programs whilst working conceptually at the level of Jackson diagrams [9]. The APE system uses a hierarchical structure (Figure-1) to represent a program.

Algorithms are generated by refining a problem into simpler components. At higher levels, problems can be partitioned into logically disjoint sub-problems, that can be tackled independently. Each level of refinement simplifies the sub-problem further until

† To appear in the Proceedings of HCI '88. Conference of the British Computer Society Human-Computer Interaction Specialist Group, Manchester, September 1988.

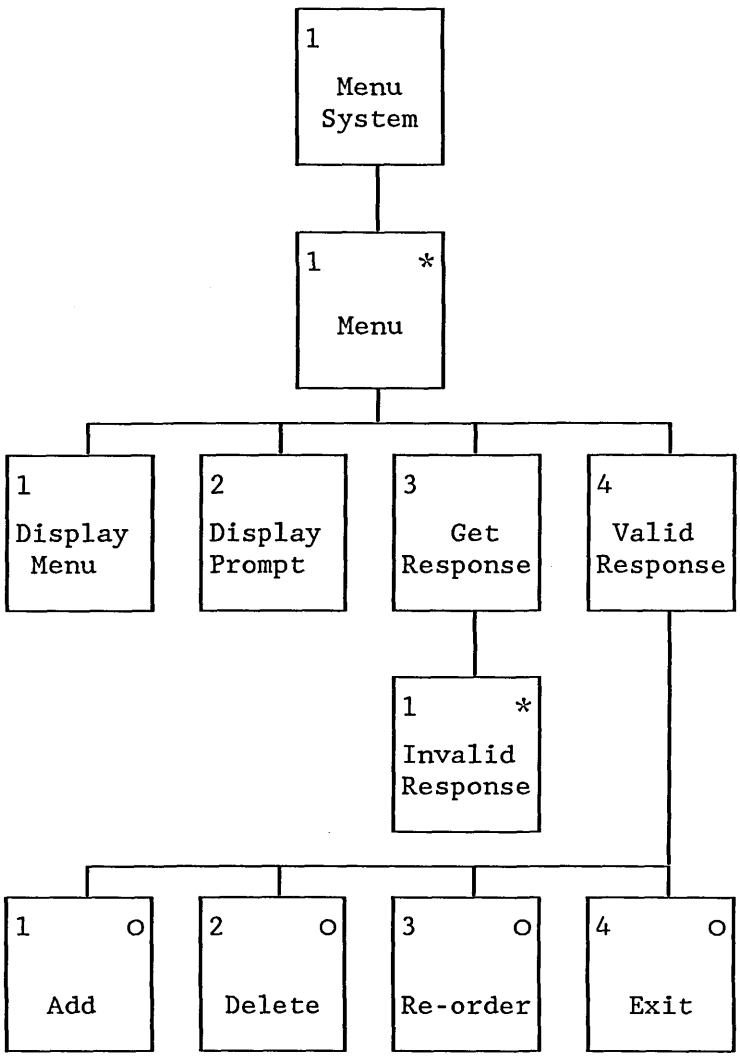


Figure 1: A Jackson Structured Tree Diagram in APE

it can be solved directly. This method of hierarchical program development embodies the approach of structured programming and top-down design.

APE has been designed to be a diagram driven system with a user interface that models the development process. The system has been written in C using the UNIX built-in Curses library functions [1], and has been implemented on a VAX-11/750 under ULTRIX Version 1.2. For a detailed description of the complete system, see Heerjee [6,7,8,4,5].

The system generates nodes for the tree diagram which are based on Jackson structured diagrams. The system provides facilities for traversing and editing the design diagram. The user may input commands either by using the pull down menus or by typing them

directly at the keyboard. The system also has a history mechanism that is useful for storing, retrieving and traversing between tree diagram structures. Furthermore, the interactive learn package gives Computer Aided Instruction courses and practice to help the novice user operate the system.

Several views of the design diagram are possible for the user. The enlarged diagram is the default and the system also provides a perspective view of the design diagram, displaying more levels/nodes on the screen. A hard copy listing of the design diagram can also be obtained.

The APE system provides support for the various software development stages, namely system design, programming, testing and debugging, coding, documentation, maintenance and modification. First the design specification of a program is entered in a system tree using comments and underlying control structures. Each leaf of this tree, representing a module, is then refined to produce code. Thus the complete source code is a refinement of the system tree and contains design information such as the functional specification and the designer's original comments.

Modules are written using the system's history mechanism. The inter-relationships between modules, based on their interface definitions, is displayed graphically in the form of an overview tree. The overview tree displays the bare skeleton of the users' program and the inter-relations of the current tree with other tree diagrams in history.

The programming phase in essence consists of refining the coding trees, that correspond to nodes on the overview tree, into a complete program. Each node of these coding trees can be refined in one of three ways; into a sequence of tasks, one of several selections, or a task that has to be iterated zero or more times. Programming statements and data definitions are inserted using the systems syntax directed editor. If the input is syntactically incorrect a one line error message is displayed. The system prompts for the types of any undeclared identifiers thus enabling a user to concentrate on the structure of a program. All declarations are stored in a symbol table and are output when the program is generated. The generated program is formatted, and comments are automatically inserted. Stubs are generated for each undefined module, thus enabling the program to be compiled and executed.

The APE system provides two methods for testing and debugging programs. The first is a direct call to the Pascal interpreter and executor. The second option enables a user to view the run-time animation of the design diagram. Breakpoints can be set and values of variables traced as in a conventional debugger.

3. User Interface

The characteristics of the user interface, as described below, were based as far as possible on available human factors information. In addition a number of users provided reports, new suggestions and ideas during the implementation stage of the system.

3.1. Screen Layout

For most interactive systems, the layout of information on a screen is important, since densely packed or cluttered screens can often overwhelm and distract users. Smith and Mosier [15] highlight the complexity of this issue by suggesting 162 guidelines for displaying data on screens.

The display was organised in pages and made use of windows. Sequences of screens were similar throughout the system for similar tasks. Within a sequence, the information on the screen gave an indication to the user of the current position. The option of going backwards in a sequence was also provided. System messages provided guidance which employed a user-centered phrasing [13]. Furthermore, all system messages were displayed in a consistent format at a fixed location on the screen. In addition, the use of multiple overlapped windows was limited because of the size of the screen and the time taken to update it.

The APE system created three windows when invoked; the menu, text and node windows. A diagrammatic view of the overall terminal screen is shown in figure-2.

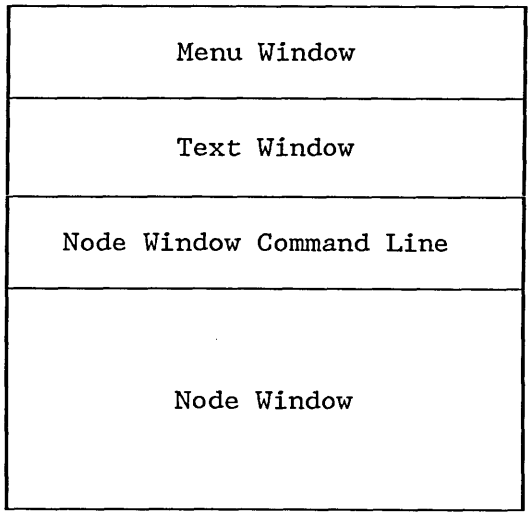


Figure 2: Diagrammatic View of the Screen Layout

The menu window was a single line window at the top of the screen, and highlighted the menu bar options, in inverse video. The coding

tree (the default), was displayed in the node window. The number and size of the nodes on this tree could all be varied to a certain degree by the user when invoking the system. The first row of this window was the command line, as all interaction with the system, except with the menu window, occurred there. All system prompts and messages were generated on the command line as well.

The text window was made up of the remaining portion of the screen which consisted of the menu window at the top, and the node window. The text window was the only window in the system in which scrolling was allowed. All input into a node took place here as the system did not provide any direct access to a node except through this window.

3.2. Command Input

Command language and direct manipulation were the two interaction styles initially used in implementing the APE system. The primary input device for manipulating the interface was the alphanumeric keyboard; occasionally the cursor control keys could be used. The commands were usually a single keystroke. For example, the 'S' key was used for generating nodes, while the cursor keys were used for traversing the tree diagram.

As the system was developed the command set became more extensive and complex. This resulted in a strategy employing single letters, shifted single letters, and 'CTRL' key plus single letters. Additionally, some commands stood alone, whereas others had to be combined in different patterns. This profusion of commands enabled the expert users to achieve a complex task with very few keystrokes. However the large command repertoire made the task of the novice and intermittent user more difficult. Therefore, it was decided to incorporate a menu interface into the system. The commands in the menu system were designed in a manner where there was a one-to-one correspondence with the options present in the menus to those that a user would normally type directly at the keyboard.

The APE system makes use of single and tree structured menu systems. In the former case, a combination of pull down, or pop-up, and extended menus is used. The style is similar to that of an Apple Macintosh. The tree structured menus enabled a large collection of data to be made available to a novice or intermittent user. An index was displayed on the screen to enable the user to maintain a sense of position.

4. Empirical Evaluation the APE System

The APE system was evaluated in order to investigate the claim that properly designed interactive tools can increase productivity and have advantages over conventional methods of generating code. The empirical evaluation was also intended to provide feedback on

possible modifications to be made to the system. The evaluation was carried out once a major section of the APE system was implemented, and consisted of a questionnaire and an experimental comparison with conventional methods of programming.

4.1. Questionnaire Evaluation

The questionnaire evaluation was designed to solicit users' general impression about the system and its interface, and their detailed views on more specific aspects of the system.

The points addressed in the questionnaire ranged from ease of use, through to the perceived utility of the system and its visual appearance. Some points that were less central to the interface design itself, but which might affect the users' perception of its quality, were also included.

4.1.1. Apparatus

Trials were conducted on CIT-101e (VT100 compatible) monitors, linked to a VAX-11/750, or Micro VAX work-stations. UNIX/ULTRIX was the operating system used in these trials, and was the system on which APE was developed and now runs.

4.1.2. Subjects

The questionnaires were given to forty-one students (mean age 30.5 years, range between nineteen and forty-two years) at Dundee College of Technology. These students were enrolled in one of four courses; post-graduate diploma in Software Engineering, third year undergraduate science degree, fifth year Applicable Mathematics degree, or a Higher National Diploma in Computing.

All candidates had some previous knowledge of computing. Some possessed a knowledge of the UNIX/ULTRIX operating system, although this was not necessary.

4.1.3. Method

The evaluation procedure consisted of three stages. Initially, a brief explanation of the design technique used by the APE system was given. This was found to be necessary because the APE system is conceptually based on Jackson diagrams, and few subjects had experience of this. Subsequently a demonstration was provided to familiarise the users with the basic facilities and tools that were essential for driving the system. The experimenter was always present to answer any queries. The example consisted of using the system to generate a program in Pascal that found the maximum of a set of numbers. This example demonstrated essential features of the system such as the menu interface, on-line help facilities and ways of directly manipulating the design diagram using the command language. First-time users were then allowed to practice and

familiarise themselves with the system, following which they were given instructions for the task that they were required to perform.

The subjects then completed one of three programming tasks in order to familiarise themselves with the system. The tasks were chosen to cover a range of applications in Pascal programming. One task was to write a program that played a simple game of noughts and crosses, while the second task was to write a scanner that produced a list of identifiers in a Pascal program. The final task was to generate a stack implementation that would also handle the two cases of stack overflow and underflow. At the end of the familiarisation, subjects were handed the questionnaire which they completed in their own time.

There were three types of question in the questionnaire. The first set (seven questions) concerned commands offered by the system to the user. Another set (ten questions) dealt with specific questions on existing features, while the third set (six questions) consisted of general open-ended questions that concerned users' opinion on some broader aspects of the system. At the end of the questionnaire, users were encouraged to state their complaints and desired modifications to the system. A conventional five point rating scale was used in the questionnaire [12], and the users indicated their degree of agreement/disagreement with the statements given in the questionnaire. A copy of the questionnaire and detailed results are available elsewhere [7].

4.1.4. Results

A total of forty-one questionnaires were administered of which thirty-nine were completed and returned. The results presented in this section are based on the median ratings given to the questions and are shown in parenthesis.

With respect to their general impression, the subjects found the interface good (5) and easy to use (4). Subjects found that the response time of the graphical interface was fast (5), but said that a high level of concentration was required in using the system (4).

Subjects had little or no previous experience of using Jackson diagrams (1). During the learning process, the subjects had some practice sessions with a learning package that was incorporated into the APE system. This was found to be useful (4).

Several questions were asked in order to evaluate the screen design. These ranged from general questions such as the overall graphical interface of the system, to more specific ones, such as the quality of the menu interface.

Subjects agreed that the overall graphical interface was good (5), and were satisfied with the amount of information that was displayed on the screen (4). The help facilities, such as the instructions, on-line help and other accompanied documentation, provided by the system were also found to be good (4). In addition, subjects expressed their satisfaction with the pull down menu selection system (4).

Subjects agreed that the different types of tree diagrams provided by the APE system were useful (5), and that sufficient information was displayed in a node of the tree (4).

Subjects were satisfied (4) with the history mechanism of the APE system. The run-time animation of the design diagram was also found to be useful (4).

Two questions also dealt with the windows in the systems. Subjects felt that the number of windows should not be reduced (4). However, they were undecided (3) on whether the size of the text window needed enlargement.

Subjects found that the system did not allow for reasonably quick error correction (3). Answers to this question may have been influenced by the fact that the system being evaluated was a prototype and could sometimes allow illegal commands that took longer to correct.

In summary, the questionnaire revealed a favourable or encouraging response for most aspects of the user interface. This presumably reflects the use of human-factors design principles throughout the development of the system.

4.2. Comparative Evaluation

The second evaluation procedure involved a comparison between the APE system and conventional means of generating code. This was designed to provide an objective test of the value of the system in program development.

4.2.1. Apparatus

The apparatus used in this experiment was the same as described in the questionnaire evaluation.

4.2.2. Subject Categories

Nineteen post-graduate students (one female and eighteen males), from a diploma course in Software Engineering at Dundee College of Technology took part in this experiment. These subjects had an undergraduate college/university degree or diploma and were studying program design techniques, as part of their curriculum.

Subjects in this experiment had previous experience of using the APE system, as all had taken part in the questionnaire evaluation of the system. The subjects were divided into two groups equated for age and performance in class. The latter characteristic was assessed by a mark based on course-work assignments over the term. The first group (nine males, mean age 24.1 years, mean class performance 61.1%, SD = 6.5), used the APE system, while the second group (one female and nine males, mean age 25.7 years, mean class performance 64.0%, SD = 9.07), used the screen editor on their machines to produce the pseudocode.

4.2.3. Method

The task was to design a detailed pseudocode algorithm that implemented a restricted form of a decision table. The pseudocode algorithm to be produced had to go through the stages of reading the data into an array from a file. It would ask questions from the user and from the corresponding responses determine which outcome to print. Four pseudocode algorithms were required to be formulated using the conventional pseudocode structuring constructs (e.g. IF..THEN..ELSE, etc.), and a given set of primitives (such as READ, PRINT, Rules and Flags). The decomposition process had to be achieved in a top-down manner and each section was required to be refined until details could be input at the primitive level alone. The complete task is presented elsewhere [7].

Both groups undertook their tasks in the same room, but no consultation between subjects was allowed and no help was given during the experiment.

4.2.4. Results

The answer sheets were returned by all nineteen subjects, and were marked by three independent lecturers with experience of teaching program design techniques. They were not involved with the design, implementation or the evaluation of the APE system. Marks were given on a ten point scale, where ten was the maximum.

The three markers were significantly consistent (Kendall's coefficient of concordance (18) = 0.71, $p < 0.01$) [14]. Therefore each subject was assigned the mean of the three lecturers' marks. These scores gave a mean APE score of 6.0 (S.D. = 1.12), and a mean screen editor score of 6.3 (S.D. 2.08), which were not significantly different.

While this result did not demonstrate a practical superiority in the quality of the solution produced on the APE system, it was at least no worse than a conventional code-generating procedure. However the two methods did differ in respect of the uniformity of the code produced by the two groups. Even though there was a wide range of ability in each group, the output from the APE group was of a more consistent standard, compared to the quality of the code

produced using conventional methods. This result was predicted for Jackson diagrams [9], and was demonstrated by a significant difference in the variance of the scores obtained by each group ($F(9,8) = 3.42$, $p < 0.05$) [2].

5. Conclusions

The results presented in this paper concern two issues, namely, obtaining user responses to the interface of the APE system, and the empirical evaluation of its usefulness as a tool.

The questionnaire resulted in a generally positive response to the system, with a majority of answers indicating approval of its characteristics. This showed both that the principle of an animated programming environment was acceptable to users, and that design decisions regarding the user interface were well-founded. After the questionnaire was analysed, changes were made to the system. These included improved error recovery routines, modifications to the on-line help facilities, and extensions to the skeleton tree diagram.

There were some advantages and disadvantages in using the questionnaire as an evaluation tool. Specific questions were found to be the most useful as they resulted in a list of areas that needed attention. However, such questions cannot hope to evaluate the entire interface, because this would result in an over-lengthy questionnaire. On the other hand, users with little or no computer experience may be unable to give a critical response, particularly if these responses must be based either on comparison with other systems or on the user's own conception of what the interface should look like. For this reason, it was also found inadvisable to ask users to pass judgement on questions that proposed new features to a system, without giving them experience with it. This view suggests that users might not be aware of what constitutes a good system, but might only be aware of what they did not like when they had to use it.

During the course of the evaluation it was noted that users tended to switch from the menu selection system to the command language. In part this may have been due to the time penalty involved in requesting a display, waiting for it to be displayed, and then searching for the desired item. Also, the large number of commands required the use of multiple menus, which was designed to give help whether or not it was requested. As expertise was developed, the command language was preferred, although it was relatively difficult to learn, and there were no on-line reminders. This supports the contention that a system should have two or more levels of dialogue, appropriate to the user's level of competence.

In the comparative evaluation, two groups of users generated code to solve a problem, using either the APE system or a screen editor. Although the mean quality of the solutions was the same for

both groups, there was significantly more variability in those produced with the screen editor. This demonstrated one of the potential benefits of a structured programming environment. These results also suggest that using such systems helps to produce standard quality products, and reduces the dependence on the experience and ability of the practitioner. It is intended to extend this type of comparative evaluation to larger and more diverse groups of users, where it is possible that differences in performance may be demonstrated. The present study was concerned with the quality of the solution to a problem; it would be of interest to obtain measures of speed as well in future trials.

Future testing of the APE system will employ controlled evaluation, with industrial user groups and in respect of its application to document preparation and as a program design aid. In general, it is hoped to establish a methodology for design and evaluation which will have an application to a range of future projects.

References

1. K C R Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*, Computer Science Division, Department of Electrical Engineering and Computer Studies, University of California, Berkeley, California 94720, August 1983.

2. D F Groebner & P W Shannon, *Business Statistics: A Decision-Making Approach, 2nd ed*, Charles E. Merrill Publishing Company, USA, 1985.

3. W J Hansen, "User Engineering Principles for Interactive Systems", *Proc. 4th FIPS Fall Joint Conf. Proceedings*, vol. 39, pp. 523-532, 1971.

4. K B Heerjee, "APE - An Animated Programming Environment", Technical Bulletin No. 13, Dundee College of Technology, Dundee, UK, 1988.

5. K B Heerjee, "APE Programmer's Manual", Technical Bulletin No. 12, Dundee College of Technology, Dundee, UK, 1988.

6. K B Heerjee, Colin J. Miller, William B. Samson & Michael T. Swanston, "The Design, Validation and Evaluation of a Software Development Environment", Submitted for publication to Software Engineering Journal, 1988.

7. K B Heerjee, "An Interactive, Graphical, Program Design and Development Tool", Unpublished Ph.D. Thesis, Council for National Academic Awards (CNAA), UK, 1988.

8. K B Heerjee, Colin J. Miller & Michael T. Swanston, "Experiences in Applying Formal Methods to Existing Software", *Proceedings of a Formal Methods Workshop*, Teeside Polytechnic, Cleveland (UK), July 1988.

9. M A Jackson, *Principles of Program Design*, Academic Press, London, 1975.

10. M A Lund, "Evaluating the User Interface: The Candid Camera Approach", *Proc. CHI'85 Human Factors in Computer Systems*, pp. 107-113, ACM, 1985.

11. D A Norman, "Design Principles for Human-Computer Interfaces", *Proc. of CHI'83, Human Factors in Computing Systems*, pp. 1-10, ACM, New York, December 1983.

12. A N Oppenheim, *Questionnaire Design and Attitude Measurement*, Gower Publishing Company Limited, Aldershot, England, 1986.

13. B Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, 1987.

14. S Siegel, *Nonparametric Statistics for the Behavioural Sciences*, McGraw-Hill Book Company, Inc., Tokyo, 1956.

15. S L Smith & J N Mosier, "Design Guidelines for User-System Interface Software", Report ESD-TR-84-190, p. 448 pages, The Mitre Corp., Bedford, MA, September 1984.

Experiences in Applying Formal Methods to Existing Software†

Kaizad B. Heerjee
Colin J. Miller
Michael T. Swanston

Dundee College of Technology
Bell Street, Dundee
Scotland

ABSTRACT

This paper reports experience gained in successfully applying formal methods to an Animated Programming Environment (APE). These techniques were applied in order to specify precisely some of the features of the APE system.

The benefits of applying formal methods particularly to existing software are described, as is the specification process itself. Axiomatic specifications are written and a prototype of a module in the APE system is developed in Standard ML.

The formal specification was developed by interrogating the implementation and documentation. A number of problems, exposed as a consequence, are presented.

† Appeared in the proceedings of a Formal Methods Workshop, Teeside Polytechnic, Middlesbrough (UK) on July 18-19, 1988.

Experiences in Applying Formal Methods to Existing Software†

Kaizad B. Heerjee
Colin J. Miller
Michael T. Swanston

Dundee College of Technology
Bell Street, Dundee
Scotland

1. Introduction

The use of natural language in defining computer systems is prone to ambiguity and vagueness, whereas mathematical notations, although more precise, are rather cryptic. Formal specification techniques are becoming increasingly important for defining systems and, moreover, they enable the use of formal methods in subsequent phases of the development cycle. The potential benefits of applying formal methods to the design and production of software is currently a topic of much discussion [16,17,14].

The value of a specification, such as that presented in this paper, is that it defines the system in question and so enables its properties to be determined by reasoning rather than by performing experiments. The latter approach can be difficult (if the system is complex) and costly (if it has not yet been built). Specifications can be constructed that answer the questions raised and which adopt a certain level of abstraction. If these specifications are presented in a mathematical notation, the question of their meaning and consistency can then be answered using mathematical means.

This paper reports on the experiences gained in applying formal methods to describe APE, an existing interactive program support environment. An overview of the APE system is presented in the next section. The use of formal methods with particular reference to the description of an existing piece of software is outlined in Section 3. The specification process itself is described and a sample specification presented in Section 4. In Section 5, a module of the APE system is specified in Standard ML following which the questions that were raised during the specification process are discussed. Conclusions are presented in Section 7.

† Appeared in the proceedings of a Formal Methods Workshop, Teeside Polytechnic, Middlesbrough (UK) on July 18-19, 1988.

2. Overview of the APE System

APE, an acronym for an Animated Programming Environment, is an interactive, graphical, program and text design and development system. The system enables users to develop programs whilst working conceptually at the level of Jackson diagrams [15]. The system is designed to generate code in Pascal, incorporate automatic syntax checking and generate data definitions automatically where possible. A run-time animation component has been incorporated to enable program monitoring and debugging. The APE system also includes facilities for document structuring and the automatic invocation of text-processing facilities.

The APE system uses a hierarchical structure (Figure-1)

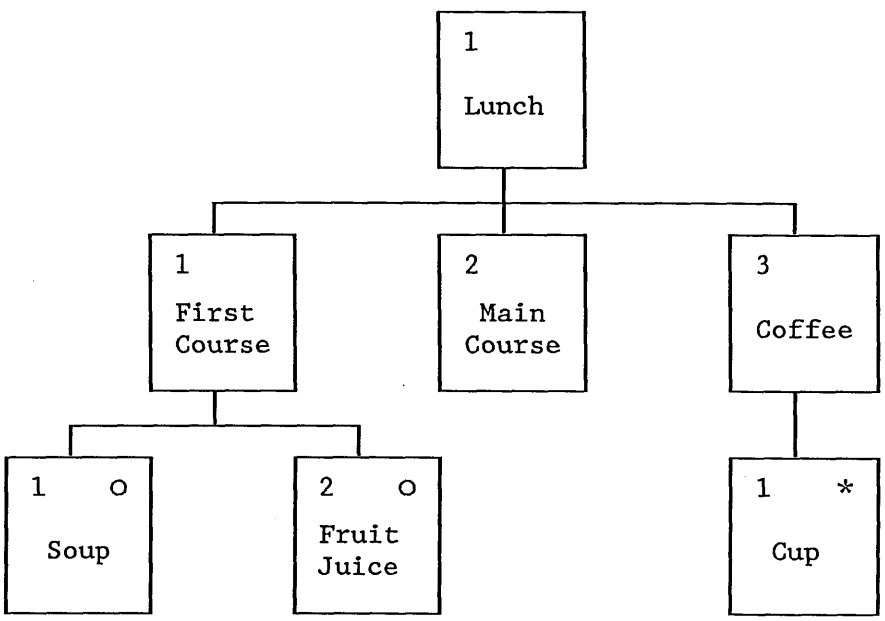


Figure 1: A Tree Structure Diagram in APE

to represent a program structure. At higher levels, problems can be partioned into logically disjoint sub-problems, that can then be tackled independently. Each level of refinement simplifies the sub-problem further until it can be solved directly. This method of hierarchical program development embodies the approach of structured programming and top-down design.

APE has been designed to be a diagram driven system with a user interface that models the development process. Although the system is interactive, and incorporates an optional use of menu facilities, it is not hardware specific and can be run with any cursor control terminal. The system has been written in C [18] using the UNIX† built-in Curses library functions [2], and has

† UNIX is a trademark of Bell Laboratories.

been implemented on a VAX-11/750 under ULTRIX‡ Version 1.2. The APE system was primarily designed as a programming tool, and a brief outline of some of its general capabilities applicable in all modes of operation are outlined below. A detailed description of the system is given elsewhere [11,12,13,8,9].

The system generates nodes for the tree diagram. In the programming mode it generates selection, sequence and iteration nodes. These nodes are based on Jackson structured diagrams. The system provides facilities for traversing and editing the design diagram. The user may input commands either by using the pull down menus or by typing them directly at the keyboard. The system also has a **history** mechanism that is useful for storing, retrieving and traversing between tree diagram structures. Furthermore, the system provides a learn package for helping the user operate the system.

Several views of the design diagram are possible. The enlarged diagram is the default. Another diagram in the system provides a perspective view of the design diagram by displaying more levels/nodes on the screen. A hard copy listing of the design diagram can also be obtained.

In the programming mode of operation, the system helps to design, test and debug programs in Pascal. Program text can be parsed or written into a node with a screen editor. In the parse mode, the system issues prompts for undefined identifiers and declarations. Sub-routines in Pascal are written using the systems history mechanism. Each sub-routine is written on a separate tree. The system knows about the inter-relation between various tree structures that are saved in history. An option is provided for viewing these inter-relations as an **overview** tree diagram. The **overview** tree displays the bare skeleton of the users' program and the inter-relations of the current tree with other tree diagrams in history.

The system generates a formatted output of the users Pascal program from the design diagram. Stubs are generated for incomplete sub-routines. Two options are provided by the system for running programs. The first is a direct call to the Pascal interpreter and executor **pix**. The second option allows a user to animate the design diagram and view the execution of the program as a tree. A user can set breakpoints and trace the value of variables as in a conventional debugger.

APE has been designed to be a diagram driven system with a user interface that models the development process. Although the system is interactive, and incorporates an optional use of menu facilities, it is not hardware specific and can be run with any cursor control terminal. The APE system's interface has been

‡ ULTRIX and VAX are a trademark of Digital Equipment Corporation.

implemented by linking several relatively independent modules. The formal specification work concentrated in formally specifying these individual modules of the system. The work reported in this paper deals with specification of parts of the APE system. The complete specification is given in Heerjee [10].

3. The Uses of Formal Methods in Existing Systems

Formal methods can be used throughout the software development life-cycle; to describe precisely the behaviour of a system, as an unambiguous starting point for producing system documentation, to check for correctness, and, for testing possible changes, modifications or reimplementations of the system. Using formal methods involves developing a precise description of the system so that its structure and detail can be understood and communicated within a design team, and between designers and clients. This kind of information is best obtained by building a model of a system using formal methods. The task of constructing such a model is an iterative process. An initial model is put forward and then refined by the application of domain independent principles. The refinement process is interspersed with frequent dialogues to help clarify details that are ambiguous or undefined in the initial model. The final model should provide a specification of the system at a high level of abstraction. In particular, it should avoid describing the representation of data within programs and details of the algorithms that have been used to implement the system. For a more detailed discussion see [3,19,6,25].

There are both short and long term benefits to be gained in specifying existing systems. In the short term, formal methods help to

(1) uncover inconsistencies in the documentation and highlight ambiguities, redundancy or non-orthogonal properties of functions in the existing system;

(2) check the design for consistency and completeness;

(3) ensure that the implementation meets is formal specification; and

(4) highlight anomalies in the existing system and suggests ways in which the system could be improved.

In the longer term formal methods

(1) facilitate user learning, since they provide an appropriate level of abstraction of the functional behaviour of the system;

(2) provide a vehicle for investigating system reimplementations, as the specification is implementation independent;

(3) enable modifications to be made with reference to the previous specification; the behaviour of these modifications, and their effects on existing parts of the system can be examined as a whole;

(4) provide a suitable prototype for experimenting with the system, and

(5) enhance the ease with which the system designs can be understood and communicated within the design team.

Formal specifications are usually based on the use of mathematical concepts in describing the requirements of a system. The method concentrates on defining what a system does rather than the implementation specific detail of how the system does it. Provided that the designers are familiar with the notation, formal specifications are easier to reason about as they do not suffer from any of the ambiguities of natural language description. This alone has a significant advantage in that it provides valuable feedback and allows for a more precise communication between designers.

Specification can be seen as complementary to prototyping. The two techniques are particularly closely linked when the formal specification language is directly executable, so that the specification acts as a prototype [1]. Experimenting with specifications is likely to reveal errors, inconsistencies and omissions at an early stage of the design process. This, therefore, provides a faster and cheaper way of incorporating modifications to the original design. However, since the specification is implementation independent, it is difficult to predict the impact of the alterations on the performance of the system and the difficulties involved in implementing the changes. As the specification is at a high level of abstraction, "it can give a better insight into the interaction of changes with other component of the system; it is just these high level interactions which get lost in the detail of implementation" [7]. Therefore, it is important for the specifier, while experimenting with the specification, to be aware of the implementation consequences before making any decision. Any ambiguities must be further investigated before implementation.

4. *The Notation*

A subset of the applicative language SML, Standard ML [5,21,4], developed at the University of Edinburgh, is the primary notation that has been used in this specification work. The name 'ML' stands for Meta Language and is the name in the LCF system, a system for proving the correctness of programs. SML is a

statically-scoped functional language which features a flexible but completely secure polymorphic type system [22].

The specifications in this document are a combination of formal SML and informal explanatory English. Formal specifications are generally hard to construct, use and understand, which according to Sannella [24], "may be a blessing in disguise". To construct a readable specification, both, formal and informal parts are necessary. The formal text, although precise and unambiguous, can be too terse for ease of reading and often its purpose needs to be explained. Conversely, an informal natural language explanation can easily be ambiguous or vague and needs the precision of a formal language to make the intent clear. Therefore, to create a good specification the structuring of the specification and the composition and style of the informal prose are as important as the formal text [6].

A specification should reveal the operation of the system a little at a time. These portions can then be combined to give a specification of the whole. This style of presentation, which Wikstrom [26] calls "structured growth", is preferred since giving the specification in small portions can enable each piece to be understood. Furthermore, when these pieces are put together the understanding of the parts that has already been gained can lead more easily to an understanding of the whole. This structural evolution of programs from specifications, by means of verified refinement steps so that a correct result is guaranteed is, according to Sannella [24], "perhaps the most exciting potential application of formal specifications" [24]. However, understanding complex specifications that are developed in this way can be quite difficult, as one needs to remember the functions of all the parts and understand the way in which they are combined. In such cases, Hayes [7] suggests that it would be useful to provide both a portion by portion development of the specification and an expanded monolithic specification as well.

To help write specifications a portion at a time, SML provides a facility by which one can describe and name parts of a specification using modules†. A module consists of two parts – an interface (signature) and an implementation (structure), which are defined separately. Every structure has a signature which gives the names of the types and functions defined in the structure. Structures may be built on top of existing structures, thus generating a hierarchy of structures. This hierarchy is also reflected in its signature. Modules are "parameterised" structures [23], since a module can be applied to a structure to return another structure. The following is a simple example of a program in SML with modules that determine whether an element (x) is a member of a list (hd::tl):

† These extensions to SML for modular programming were first proposed by MacQueen [20].

```
signature MemberSIG =
    sig
        val member : (('a list) × 'a) → bool
        axiom (∀ x: 'a)
            => member (nil, x) = false
        axiom (∀ x)
            => member (hd::tl, x) =
                (x = hd) ∨ member (tl, x)
    end ;

structure Member : MemberSIG =
    struct
        fun member (nil, item) = false
          | member (hd::tl, item) =
            (hd = item) orelse member (tl, item)
    end ;
```

Here '=>' is used to write unnamed functions. For example,

```
        fn x => []
```

evaluates to a function that, when given an object x, returns an empty list.

The information provided by a signature is rather limited, and is not sufficient, for example, to prove program correctness or for program documentation. To make the interface more useful, signatures have been extended to include axioms. Axioms (sentences of first order logic in this paper), place constraints on what operations the operations in a signature are supposed to do. This extension, with the aim of doing formal development and proofs of SML programs, has been termed "Extended ML" by Sannella and Tarlecki [23]. An example of an axiomatic signature in SML is the following specification to find the largest integer in a given list of integers:

```
signature MaxSIG =
    sig
        val max : (int list) → int
        axiom (∀ IntList: int list)
            => IntList ≠ nil
                ⇒ member (IntList, max (IntList))
        axiom (∀ a: int, IntList)
            => member (a, IntList)
                ⇒ max (IntList) ≥ a

        val member : ((int list) × int) → bool
        axiom (∀ mem: int)
            => member (nil, mem) = false
        axiom (∀ mem)
            => member (hd::tl, mem) =
                (mem = hd) ∨ member (tl, mem)
    end ;
```

In the above signature, '⇒' is used to denote implication. The expression on the left hand side of the '⇒' may be an equation (as above) or a boolean-valued expression that can be regarded as abbreviating equations of the form 'expr=true'.

Using the features described above, axiomatic specifications have been written in SML to describe some of the more desirable features of the APE system and are discussed below.

5. Specification of APE in Standard ML

The main aim in formally specifying the APE system was to build a model of the system that was consistent with the implementation. In attempting to do so, questions arose that had to be answered in consultation with the source code and the manual. Since the system was developed recently, straightforward questions about its operations were answered by the implementors. Additionally, experienced users of the APE system were available who were willing to help in the specification process.

Building a model for the APE system was an iterative process. It involved forming a crude initial model which was extended to cover facets of the system not initially dealt with, and used for redesigning or refining inconsistencies that were found. During this design process questions were formulated about the desired behaviour of the system which were then answered from the specification. Some questions led to inconsistencies being highlighted between the model, implementation and documentation. These inconsistencies were generally caused by errors in the implementation or by the informal natural language description of the manual. The specification was then given to people experienced in the use of formal methods who then commented on its style and level of

abstraction, and suggested ways in which it may be improved or simplified. These people also had experience of using the APE system which was found to be useful, as inconsistencies between the specification and the implementation could be bought to the notice of the specifier. This iterative process was carried out until the specifiers were satisfied with the model that was built.

The APE system encourages structured programming and imposes a tree structure on the problem under consideration. A tree, in the APE system, is composed of nodes connected by branches that occur singly or in sets and whose members may be sequential or selected, but not both. This contributes to the simplicity of the tree structure because it ensures mutual exclusion in selection. Nodes may be iterated as well, although following the syntax of Jackson's method of program design [15], iterative nodes may belong to sequences alone. The ultimate constituent of the APE system is a node. The type of a node can be formally specified in SML as:

```
signature NodeSIG =
    sig
        type     NODE_DATA      ( of type string )
        type     NODE_NO        ( of type integer )
        datatype NODE_STATUS    = current
                                | not_current
        datatype NODE_TYPE      = sequence
                                | selection
                                | iteration
        datatype NODE           = node of (NODE_NO ×
                                           NODE_STATUS ×
                                           NODE_DATA ×
                                           NODE_TYPE ×
                                           NODE list)
    end ;
```

This recursive data type definition for a 'NODE' is a natural and appropriate way of representing trees and parallels the informal description of the tree (Figure-1) and its C implementation (Figure-2). Thus the tree in Figure-1 can be written as:

```
node (1, current, "Root", sequence,
    node (2, not_current, "Son 1", sequence,
        node (3, not_current, "Son 1.1", selection, nil),
        node (4, not_current, "Son 1.2", selection, nil)),
    node (5, not_current, "Son 2", sequence, nil),
    node (6, not_current, "Son 3", sequence,
        node (7, not_current, "Son 3.1", iteration, nil))) ;
```

An important pattern to be noticed here is that functions on recursively defined data values are defined recursively.

```
typedef struct _TREE {
    int             node_no;
    char            node_type;
    char           *node_data;
      ...             ...
      ...             ...
    struct _TREE   *father;
    struct _TREE   *son;
    struct _TREE   *youngbrother;
    struct _TREE   *oldbrother;
} _TREE ;
```

Figure 2: The C data structure for a Node

The APE system encourages the designers to split a program into components of manageable size. Each sub-routine in a user program would consist of an individual tree structure. Therefore, a complete program consist of a single top-level tree diagram that would contain calls to other sub-routine tree diagrams. This can easily be represented as a list of NODE's. Hence, two trees defined as

```
node (1, current, "Main Program", sequence, nil) ;
node (1, not_current, "Function Max", sequence, nil) ;
```

can be represented, using the above definitions, as:

```
[node (1, current, "Main Program", sequence, nil),
 node (2, not_current, "Function Max", sequence, nil)] ;
```

As an example of specification and program development in SML, and based on the above definitions for a 'NODE', we develop axiomatic specifications that model the **paste** operation in the APE system. The specifications presented here are derived by the specification process discussed in the earlier section. SML signatures are used in describing this operation of the APE system. The implementation for these signatures is presented in the Appendix.

5.1. The Environment

We begin with a specification of the APE environment. A tree in the APE system is a finite sequence of nodes of any length, the definition for which has been given above. The size of the tree is always equal to the number of append operations that have been performed on the current tree since its creation – independent of the number of other (cut, copy and paste) operations.

The initial state of the design diagram is given by a single sequence node called the 'root'. In general, a specification can be constructed to determine whether a node (n), exists in a 'NODE list';

```
signature NodeSIG =
    sig
        ...          ...
        val node_in_tree : ((NODE list) × NODE) → bool
        axiom (∀ n: NODE)
            => node_in_tree (nil, n) = false
        axiom (∀ n: NODE, hd, tl: NODE list)
            => node_in_tree (hd::tl, n) =
                  (n = hd)
                ∨ node_in_tree (n, tl)
                ∨ node_in_tree (n, subtree (hd))
    end ;
```

At any time, the system will contain a single node whose NODE_STATUS is 'current'. Such a node is called the **current** node and is visually represented in the APE system a cursor sitting on the node. This is specified by the specification for 'head' as:

```
signature NodeSIG =
    sig
        ...          ...
        val head : (NODE list) → NODE
        axiom head (nil) = NULL
        axiom (∀ l: NODE list)
            => head (l) = ∃ n
                => (extract_node_status (n) = current
                      node_in_tree (l, n))

        val head_in_tree: (NODE list) → bool
        axiom head_in_tree (nil) = false
        axiom (∀ l: NODE list)
            => head_in_tree (l) = ∃ n
                => extract_node_status (n) = current
                ∧ node_in_tree (l, n)
    end ;
```

and where the NULL node is defined as,

```
val NULL = node (0, not_current, "", sequence, []) ;
```

Before proceeding to describe the **paste** operation, we present an overview of two SML modules, MakeSIG and MoveSIG, that model operations of generating nodes and traversing the tree diagram in the APE system, respectively.

There are three basic types of nodes in the APE system, namely, sequence, iteration (or loop), and selection. Nodes in the APE system cannot be mixed, i.e. it is an error to have a selection node along side a node denoting a sequence. When a node is added to a tree, the original tree is changed. Nodes in the APE system are generated under the **current** node. Additionally, any sub-tree below this node (current) is deleted before new siblings are generated. The SML module MakeSIG specifies such an environment for adding nodes to a tree.

The APE system also provides several methods for traversing the tree design diagram. The simplest way is by moving from one node to another. This is achieved by

(1) using the cursor keys; or

(2) the four **vi**† keys; or

(3) using the systems pull down menu facilities.

In either case, this mode of operation will enable a user to traverse the design diagram one node at a time. The system also provides options for faster traversal of the tree. Additionally, all traversing commands in the APE system are issued with respect to the **current** node. The SML module MoveSIG specifies the environment used in traversing the design diagram.

5.2. The Paste Operation On Trees

Having specified the environments, we proceed to specify the **paste** operation for trees. In the APE system, the **paste** function is used for inserting a node or a sub-tree that has been previously saved by using the **copy** or **cut** operations, to the left, right or below the current node.

All operations work directly on a 'NODE list' that denotes a tree (1). The operation **cutit** removes a NODE (together with the sub-tree below it), from a tree (1).

```
signature EditSIG =
    sig
        structure Node: NodeSIG
        structure Make: MakeSIG
            ...     ...
            ...     ...
        val cutit: ((Node.NODE list) × Node.NODE)
            → (Node.NODE list)
        axiom (∀ n: Node.NODE)
            => cutit (nil, n) = nil
```

† **Vi** is the UNIX screen oriented text editor.

```
        axiom (∀ l: (Node.NODE list), n)
            => Node.node_in_list (l, n)
                ⇒ cutit (l, n) = ∃ l'
                    => Make.creat_nodes (l',
                            Node.extract_node_type (n), l) = l
    end
```

The above specification for **cutit** states that a node (n) can be deleted from l, if l is not empty and n exists in l. **Creat_nodes** generates new nodes under the **current** node with type (t). Any sub-tree of the **current** node is first deleted before the new nodes are created.

The operation for adding a node (n) to the left of the **current** node can be specified as follows:

```
signature EditSIG =
    sig
            ...     ...
            ...     ...
        structure Move: MoveSIG

        val valid_paste: (Node.NODE × Node.NODE) → bool
        axiom (∀ x, y: Node.NODE)
            => valid_paste (x, y) =
                Node.extract_node_type (x) =
                Node.extract_node_type (y)

        val paste_to_left: ((Node.NODE list) × Node.NODE)
                        → (Node.NODE list)
        axiom (∀ l: (Node.NODE list), x: Node.NODE)
            => valid_paste (x, Node.head (l))
                ⇒ paste_to_left (l, x) = ∃ l'
                    => cutit (l', x) = l
                    ∧ Move.has_younger_brother (l', x)
    end
```

According to the above signature, when a node (x) is added to a tree (l) it results in a new tree (l'), provided the following conditions hold:

(1) the **current** and x nodes are both of the same type; this is important as nodes of one type cannot be added to a node of another type on the same level;

(2) if the node x is deleted from the new tree l' it results in the original tree l; and

(3) the node x has a younger brother in the new tree l'.

Finally, the operation to specify the **paste** function is given by

```
signature EditSIG =
    sig
        structure Node: NodeSIG
            ...     ...
            ...     ...
        val paste: ((Node.NODE list) × Node.NODE ×
                            string) → (Node.NODE list)
        axiom (∀ n: Node.NODE, a: string)
            => paste (nil, n, a) = nil
        axiom (∀ l: Node.NODE list), n, a)
            => Node.head (l) = Node.NULL
                ⇒ paste (l, n, a) = l
        axiom (∀ l, a)
            => paste (l, Node.NULL, a) = l
        axiom (∀ l, n, a)
            => a = "left"
                ⇒ paste (l, n, a) =
                    paste_to_left (l, n)
    end
```

The function takes a tree (l) and a node n (that could contain a sub-tree below it) and returns the new tree with the node n added to the left of the **current** node. The operations will fail if there is no tree, or the tree has no current node. The cases when the paste operation is not performed are

(1) if the tree is empty, in which case the empty tree is returned; or

(2) if the tree contains no node marked as **current**, in which case it returns the original tree; or

(3) if there is nothing to paste, i.e. node n is the NULL node, then the original tree is returned.

The complete environment is given below:

```
signature EditSIG =
    sig
        structure Node: NodeSIG
        structure Make: MakeSIG
        structure Move: MoveSIG

        val paste: ((Node.NODE list) × Node.NODE × string)
                                        → (Node.NODE list)
```

```
val valid_paste: (Node.NODE × Node.NODE) → bool
axiom (∀ x, y: Node.NODE)
    => valid_paste (x, y) =
        Node.extract_node_type (x) =
        Node.extract_node_type (y)

val paste_to_left: ((Node.NODE list) × Node.NODE)
                        → (Node.NODE list)
axiom (∀ l: (Node.NODE list), x: Node.NODE)
    => valid_paste (x, Node.head (l))
        ⇒ paste_to_left (l, x) = ∃ l'
            => cutit (l', x) = l
            ∧ Move.has_younger_brother (l', x)

val cutit: ((Node.NODE list) × Node.NODE)
            → (Node.NODE list)
axiom (∀ n: Node.NODE)
    => cutit (nil, n) = nil
axiom (∀ l: (Node.NODE list), n)
    => Node.node_in_list (l, n)
        ⇒ cutit (l, n) = ∃ l'
            => Make.creat_nodes (l',
                Node.extract_node_type (n), l) = l
```

```
axiom (∀ n: Node.NODE, a: string)
    => paste (nil, n, a) = nil
axiom (∀ l: Node.NODE list), n, a)
    => Node.head (l) = Node.NULL
        ⇒ paste (l, n, a) = l
axiom (∀ l, a)
    => paste (l, Node.NULL, a) = l
axiom (∀ l, n, a)
    => a = "left"
        ⇒ paste (l, n, a) =
            paste_to_left (l, n)
```

Following Sannella [24], the specification of types and functions that are intended to be strictly local to the specification of other types and functions in the signature are enclosed in a box.

6. Questions and Problems Identified by Interrogating the Specification

The questions that arise during the specification process indicate problems either in documentation or in the implementation of the system. This provides the system designers and maintainers with valuable feedback on possible problem areas. Therefore, once a model of the system was completed, it was reviewed by several

people experienced in using formal methods. The reviewers compared the model with the relevant sections of the manual and looked for ambiguities or inconsistencies in the specification. All queries that arose were then answered either by consulting the formal model of the system or by consulting experienced users of the system. Several questions were asked of the system during this review process. The questions can be classified into two categories, namely

(1) General questions: dealing with the overall system specification.

(2) Specific questions: that were asked of individual modules.

Some of the general questions that were asked of the system included:

(1) Is the model inconsistent with the implementation?

(2) Is the model inconsistent with the documentation?

(3) Is the model independent of implementation details?

(4) Is the chosen model at a suitably high-level of abstraction?

(5) Is the implementation correct?

(6) Are there any inconsistencies or ambiguities in the documentation?

Some of the questions that were raised during the specification process of the **paste** operation were

(1) Are the default actions consistent throughout the chosen model?

(2) What happens to the previous siblings of the **current** node after a **paste** operation?

(3) How often can a sub-tree be added to the **current** node?

(4) Can any sub-tree be added to the left of the **current** node?

(5) Can the **root** node have any brothers?

We shall now look in detail at the last two questions with respect to the earlier model of the **paste** operation. That model works for all cases, except when

(1) the tree has no nodes, or

(2) the tree has no **current** node, or

(3) the node to be pasted is the NULL node.

A closer examination of the manual page revealed that the model did not match its informal natural language specification. This suggested that the model was inconsistent as the manual entry specified that the **root** node could not have any brothers either to its left or right. In the earlier model the specification for the **paste** operation did not check whether the **current** node was also the **root** node. The following incorrect **paste** specification was used:

```
signature EditSIG =
    sig
        ...    ...
        ...    ...
        val paste: ((Node.NODE list) × Node.NODE × string) →
                    (Node.NODE list)
        ...    ...
        ...    ...
        axiom (∀ n: Node.NODE)
            => paste (nil, n, _) = nil
        axiom (∀ t: (Node.NODE list), n)
            => Node.head (t) = Node.NULL
                ⇒ paste (t, n, _) = t
        axiom (∀ t)
            => paste (t, Node.NULL, _) = t
        axiom (∀ t, n)
            => paste (t, n, left: SIDE) =
                paste_to_left (t, n)
    end ;
```

That is, the **paste** function would invoke the **paste_to_left** function provided that the tree (t) had a **current** node. The new corrected model contains the axiom

```
axiom (∀ t, n)
    => Move.root (t) = Node.head (t)
        ⇒ paste (t, n, _) = t
```

The altered specification for the **paste** operation is:

```
signature EditSIG =
    sig
        ...    ...
        ...    ...
        val paste: ((Node.NODE list) × Node.NODE × string) →
                    (Node.NODE list)
        ...    ...
        ...    ...
        axiom (∀ n: Node.NODE)
```

```
        => paste (nil, n, _) = nil
    axiom (∀ t: (Node.NODE list), n)
        => Node.head (t) = Node.NULL
            ⇒ paste (t, n, _) = t
    axiom (∀ t)
        => paste (t, Node.NULL, _) = t
    axiom (∀ t, n)
        => Move.root (t) = Node.head (t)
            ⇒ paste (t, n, _) = t
    axiom (∀ t, n)
        => paste (t, n, left: SIDE) =
            paste_to_left (t, n)
end ;
```

Several benefits were derived from this review process. The first was that people who were experienced in formal methods but were not part of the design team would be involved. This is crucial as designers and developers often try to find problems by forming a mental model of the implementation and the manual. Another important benefit of the review process is that it provides an opportunity to modify parts of the model that have proved to be confusing or misleading. This is important to the system maintainers, since misunderstandings or the bad presentation of a concept can lead to a wrong decision by a user.

6.1. Adding New Modules

As another example we consider the problems caused by incorporating a new module in the APE implementation. The APE system has been implemented using modules, each of which does a specific task. Before any new module is integrated with the APE system, an attempt is made to check for correctness. The checks were generally in the form of informal tests of the module on a few input values. No formal proof of correctness is attempted. A substantial amount of testing can be done by independently exercising each module. However, testing individual modules and comprehensively testing the system as a whole involves a big jump in the levels of abstraction and is rarely attempted.

Once the **paste** function passed a set of informal tests, it was incorporated into the system. This led to a few inconsistencies between the mathematical model and the implementation and suggested either that the model was incorrect or that there were bugs in the implementation. In the APE system different types of nodes cannot be mixed on the same level. For example, it is illegal to have a selection node alongside a node denoting a sequence. In the mathematical model, this constraint was specified by **valid_paste**. The specification for **valid_paste** is simple, yet crucial in determining whether the system should proceed with the operation of adding a sub-tree to the left of the **current** node.

However, the effects of not introducing this subtle test into the implementation had disastrous implications. It led to several bugs being generated when the system was in use. Additionally, most of the side effects caused by this omission were not immediately evident as they were triggered when some function, independent of the **paste** operation was attempted, usually resulting in the program being aborted.

This anomaly was highlighted by the mathematical model and resulted in the quick detection and correction of the error in the implementation. One can only speculate at the amount of time and effort that would otherwise have gone into locating and correcting the error had the specification of the system not been present. This example also demonstrates the usefulness of applying the specifications to highlight the interaction between modules, since the functions from one module will need access to components of another. However, any such inconsistencies discovered in the process of specifying a system should be closely examined before changes are carried out.

6.2. The Symbol Table

As a final example, we consider the symbol table used by the APE system in the programming mode of operation. The representation of a program in the form of an overview tree was initially used by the APE system for storing and retrieving declarations, thus enabling the symbol table to be block structured. All declarations local to a particular tree were stored at the root node of each tree, which as a rule displayed the name of the subroutine under it. This was considered a convenient way of keeping track of variables declared in all the blocks enclosing that point. Additionally, the information held in the root node is important in determining the inter-relations between the different tree diagrams and is also used in producing the declaration section when generating the final program.

However, whilst specifying the symbol table module of the APE system, doubts were expressed with regard to the complexity of the implementation. It was considered inappropriate to store declarations in separate trees. Experimentation with the formal model generated a simpler representation for the symbol table. In the new representation, symbols are linked together in a list; the only access to it being through the functions **lookup** and **install**. This makes it easy to change the symbol table organisation should it become necessary. The symbol table uses linear search, which is entirely adequate for the APE system, since variables are looked up only during parsing, not execution. **Install** places a new variable with its associated type at the head of the list.

7. Conclusions

The specification language SML has been used to build a mathematical model of the APE system. The immediate benefits being an increased understanding of the system and the detection of some errors in the implementation. By using a mixture of natural language and SML the specification of a module of the APE system is presented which is comprehensive enough to describe the essential aspects of the system's behaviour, but is sufficiently abstract in that it will not burden the reader with the kind of detail that will appear in the implementation. This has offered the possibility of showing the advantages of applying formal methods to capture important aspects of the behaviour of a large system. However, it must be pointed out that one reason we have been successful in applying formal methods to the APE system was its good modular structure which has enabled us to concentrate on individual modules in relative isolation.

A common criticism of executable specification languages is that they do not provide the same level of abstraction as, say, VDM or Z. However, these disadvantages can be overcome in SML by generating axiomatic specifications and using modules. Furthermore, SML has the added advantage of being executable (although the axioms cannot be executed), thus enabling the specification to be checked for correctness. An executable specification has several uses; allowing dynamic reviews of the design among the design team and demonstrating the model to potential users in order to obtain comments, feedback, and clarification of user requirements. It is to be hoped that a future implementation of SML will involve a theorem proving element to check implementations automatically against axioms.

There are many aspects of the APE system, such as the **history** mechanism and the specification of the automatic program generator, that have not been covered in this paper. Details of these are included in Heerjee [10] and the user manual of the implemented system [8]. An attempt has also been made at proving mathematically that the axioms satisfy the implementation [13].

The difficulties encountered in developing a model were in learning the language, SML in this case, and in achieving the right level of abstraction. The latter characteristic is very important since the specifications of the modules of a system should avoid implementation details. In the case of the APE system this was achieved by hierarchically structuring SML modules.

The principal benefit of constructing a formal model is the development of a framework to aid communication between people involved with system maintenance. The model can also be used to investigate future changes, and since this framework provides relevant abstraction of user and system behaviour it should facilitate improved documentation and user learning.

Acknowledgement

The authors wish to thank William Samson for numerous discussions and valuable suggestions on this paper during its preparation.

References

1. Heather Alexander, "Formal Specification and Rapid Prototyping Techniques for Human-Computer Interaction", Technical Report TR. 26, University of Stirling, Stirling, Scotland, August 1985.

2. Kenneth C. R. Arnold, "Screen Updating and Cursor Movement Optimization: A Library Package", in *The UNIX Programmer's Manual, Vol. 2C*, Computer Science Division, Department of Electrical Engineering and Computer Studies, University of California, Berkeley, California 94720, August 1983.

3. Dines Bjorner and Cliff B. Jones, *Formal Specification and Software Development*, Prentice-Hall International Series in Computer Science, Prentice-Hall International, Inc., Englewood Cliffs, USA, 1982.

4. Luca Cardelli, "ML under UNIX", Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1984.

5. Robert Harper, David MacQueen, and Robin Milner, "Standard ML", Internal Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh (UK), March 1986.

6. Ian Hayes, *Specification Case Studies*, Prentice-Hall International Series in Computer Science, Prentice-Hall International, Inc., UK, 1987.

7. Ian J. Hayes, "Applying Formal Specification to Software Development in Industry", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 169-178, IEEE, February 1985.

8. Kaizad B. Heerjee, "APE Programmer's Manual", Technical Bulletin No. 12, Dundee College of Technology, Dundee, UK, 1988.

9. Kaizad B. Heerjee, "APE - An Animated Programming Environment", Technical Bulletin No. 13, Dundee College of Technology, Dundee, UK, 1988.

10. Kaizad B. Heerjee, "The Formal Specification of an Animated Programming Environment in Standard ML", Technical Bulletin No. 14, Dundee College of Technology, Dundee, UK, 1988.

11. Kaizad B. Heerjee, Michael T. Swanston, Colin J. Miller, and William B. Samson, "The Design and Evaluation of an Animated Programming Environment", *People and Computers, Proceedings of the HCI '88 Conference of the British Computer Society Human-Computer Interaction Specialist Group*, Manchester (UK), September 1988.

12. Kaizad B. Heerjee, Colin J. Miller, William B. Samson, and Michael T. Swanston, "The Design, Validation and Evaluation of a Software Development Environment", Submitted for publication to *Software Engineering Journal*, 1988.

13. Kaizad B. Heerjee, "An Interactive, Graphical, Program Design and Development Environment", Unpublished Ph.D. Thesis, Council for National Academic Awards (CNAA), UK, 1988.

14. C. A. R. Hoare, "Programming is an Engineering Profession", Technical Monograph No. 27, Oxford University Programming Research Group, 1982.

15. M. A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.

16. C. B. Jones, *Software Development*, Prentice-Hall International, 1980.

17. C. B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.

18. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs., New Jersey, 1978.

19. E. Knuth and E. J. Neuhold, *Program Specification: Proceedings of a Workshop*, Lecture Notes in Computer Science, 152, Springer-Verlag, New York, 1983.

20. D. B. MacQueen, "Modules for Standard ML", *Proceedings of 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, 1984.

21. Robin Milner, "The Standard ML Core Language", Report CSR-168-84, Department of Computer Science, University of Edinburgh, Edinburgh (UK), 1984.

22. R. G. Milner, "A theory of type polymorphism in programming", *Journal of Computer and System Sciences*, vol. 17, pp. 348-375, 1978.

23. Donald Sannella and Andrzej Tarlecki, "Program Specification and Development in Standard ML", *Conference Proceedings on Principles of Programming Languages*, pp. 67-77, ACM, New

Orleans, 1985.

24. Donald Sannella, "Formal specification of ML programs", Report, pp. 1-19, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh (UK), 1986.

25. J. Staunstrup, *Program Specification: Proceedings of a Workshop*, Lecture Notes in Computer Science, 134, Springer-Verlag, New York, 1982.

26. Ake Wikstrom, *Functional Programming Using Standard ML*, Series in Computer Science, Prentice-Hall International (UK) Ltd, 1987.

Rapid Implementation of SQL: A Case Study Using YACC and LEX

Kaizad B. Heerjee
Rubik Sadeghi‡

Department of Mathematics and Computer Studies
Dundee College of Technology
Bell Street, Dundee
Scotland

ABSTRACT

YACC and LEX are two powerful UNIX† tools, that are largely ignored by all but compiler writers, indeed, while considerable time and effort is being devoted to software reuse, little immediate interest has yet been raised on the part of the ordinary UNIX user.

This paper demonstrates how a subset of the SQL query language has been implemented as an interface to a relational database system (PRECI/C in this case) using YACC and LEX.

Keywords: software development, query languages, relational databases.

‡ Currently working with Oracle (UK).
† UNIX is a trademark of Bell Laboratories.