Formal functional testing of graphical user interfaces.

Stephen Wai-Leung Yip

FORMAL FUNCTIONAL TESTING OF GRAPHICAL USER INTERFACES

Stephen Wai-Leung Yip

October 1992

Submitted for the Degree of Doctor of Philosophy

School of Engineering & Computer Science

University of Durham

Abstract

Graphical user interface software has acquired a high degree of popularity in a relatively short time. This thesis investigates the software testing of graphical or window-based user interfaces. It proposes an original validation approach called Formal Functional Testing (FFT). This approach tests a user interface by its conformance to the required functions as stated in a formal functional specification. A specification language (called WinSpec) has been developed, using states and state predicates to specify functions of graphical user interfaces. A special form of state transition diagram called WinSTD is introduced to capture the visual appearance of display objects, and the control flow of interactions. Functional test cases are then derived from specifications. The problem of test case selection is addressed by analysing function paths into interaction sequences. The graph theoretic algorithms of the Euler tour and the postman tour have been applied to derive optimal test cases. This new validation approach is explored in the specification and testing of a number of user interfaces. These include a logon interface and a window editor. A 100% function coverage criterion is used, producing relatively short test sequences that can be executed manually in about 10 minutes. The test sequences derived from formal specifications are evaluated with seeded error detection and code coverage measurements. The results obtained show a 80% success rate in the detection of seeded errors and a 70% code coverage.

Acknowledgements

I would like to thank my supervisor Dr. Dave Robson, for his kindness and support. I am also in debt to Prof. Keith Bennett for the facilities in the department, sponsorship to attend technical conferences and critical comments on my work. I am grateful to Dr. John Welford, Mr. Barry Cornelius and Miss Jenny Newton for comments on an earlier draft of this thesis. Jenny checked Chapters 5 and 6, Barry gave useful comments to improve the WinSpec notations, and John gave me support as a fellow Christian in addition to technical comments. Thanks also go to fellow research students in the Computer Science Division, especially the "testers", for many heated discussions as "iron sharpens iron" (Proverbs 27:17).

A remote "thank you" to Rick Kuhn of the US National Bureau of Standards, whom I never met, who read a very early draft of the first of my papers (quite voluntarily) and sent encouraging comments. This served as a compass to cast the direction of my research, when my knowledge and experience of the area was too immature to be sure on what to focus. For similar reasons I would like to thank the referees who gave comments on [Yip91a] to [Yip91d]. Sincere thanks are also due to Abbas, Ian, Jim and Hu, of the Open Software Foundation (Cambridge, Massachusetts); Microsoft (Seattle); Hewlett Packard (Oregon); and DEC Western Research Lab (Palo Alto). They allowed me to visit their companies during April 1991. The visits enlightened me to the development and testing work on window systems and GUIs at these locations. Abbas (Dr.Birjandi) also called, organized and chaired the session on GUI testing in the Hawaii International Conference in Jan 1991, and made some useful comments on my work. A number of USENET news group messages have assisted my research, particularly ones from Tim Endres of Ice Engineering Inc. (MI 48189, USA), which provided practical help on the JRR tool.

Acknowledgement is given to Symantec Corporation (Bedford, Massachusetts, USA) for permission to use some of the GUI programs in their THINK Pascal package as test objects in this thesis. Grants from the UK Science and Engineering Research Council (SERC) and the British Telecom Laboratories (Martlesham Heath, Ipswich) are gratefully acknowledged. Mr. Colin Archibald, Mr. Stuart Birchall, Mr. Ray Lewis and Mr. Ray Watts of the BT Laboratories have given me support. Mr. Mike Cooper, Miss Yvette Rooke and Miss Valerie Walker have helped to improve the English in this thesis. The examiners' comments have led to the last and vital improvements to this thesis. I am alone responsible for any remaining errors in the thesis, grammatical or technical.The last word of thanks are taken from 1 Corinthians 4:7, "What do I have that I have not received" from the Lord, my late parents and so many who sojourn with me concurrently at many of life's rendezvous.

Synopsis

This thesis describes the research that the author has undertaken for a PhD in Computer Science. Chapter 1 begins with an introduction to graphical user interfaces (GUIs), justifying their importance and the need for proper validation. Chapter 2 serves as an overview of existing software testing techniques. The various problems in GUI validation are analysed in Chapter 3. These problems are classified into 3 categories : functional; structural; and environmental issues. The largest functional difficulty identified is the lack of a formal specification method suitable for validation purposes. The main structural problem is deciding on which of the software levels (i.e. window systems, toolkits, user interface management system (UIMS) or applications) to target tests. The environmental issues involve human testers, automation, input synthesis and output visual verification.

Based on the findings of the problem analysis (Chapter 3) and knowledge of existing testing techniques (Chapter 2), it was decided to develop a functional testing approach. Chapter 4 details a literature survey of specification methods for user interfaces, as functional tests are specification based. The literature survey reveals that none of the existing specification methods are suitable for the derivation of functional test cases. In Chapter 5, an original specification approach for graphical user interfaces is developed. All display objects are enumerated in a special state diagram called WinSTD. Interaction functions relating display objects are specified in a set of formal notations called WinSpec. Chapters 6 and 7 give details of the test case derivation process. Chapter 6 deals with the selection and grouping of individual functions to form effective test sequences, as testing budgets are limited. Chapter 7 addresses the actual mechanics of generating tests from specifications. In essence the 'Inputs' clauses in specifications of functions provide the basis of test input generation. The other vital part of test cases, test oracles for checking output of functions, is obtained from the state predicates.

Chapter 7 uses a small user interface, the Logon interface, to illustrate the derivation of test cases. Chapters 8 and 9 together present the case study of ThinkEdit, a relatively larger user interface. The two chapters cover the specification and testing of ThinkEdit respectively. Specification for a number of other GUIs is discussed in Chapter 10.

Chapter 11 examines the issues of automating the proposed approach, and reports on wider automation work on GUI validation being pursued in industry. Chapter 12 presents an analysis and review of the results of the case studies. Chapter 13 gives conclusions and future directions. Specific and technical terms used in this thesis are printed in *italics* at their first occurrence, and their respective meanings collected together in a glossary in appendix A.

Contents

Chapter 1

Introduction

This thesis is the outcome of an investigation into the development of approaches to the validation of graphical user interfaces (GUIs). It begins by introducing graphical user interfaces.

1.1 The advent of graphical user interfaces

The concept of using windows, icons, pop-up menus and a mouse as a user interface originated from Xerox PARC (Palo Alto Research Centre) in the late 1960s, in projects such as SmallTalk and Star [Myers88]. The first use of icons was due to [Smith82] in the design of the Star User Interface at Xerox. The graphical user interface concept was developed as part of the preparation for and expectation of the shift from mainframe to distributed computing. It was not until the mid to the late 1980s, when more powerful CPUs in workstations and PCs coincided with the lower cost of memory and bitmap displays, that window systems eventually became generally available to users of a wider range of vendors' hardware. Since then, window user interfaces have become popular, and now play an important role within many software packages. Surveys of artificial intelligence applications, for example, report that 40% to 50% of the code and run time memory are devoted to aspects of the user interface [Bobrow86]. Another survey [Took90] reports that 50% to 80% of interactive systems are devoted to user interface considerations.

Graphical user interfaces are sometimes called *WIMPs*, for Windows, Icons, Menus and Pointers (or Window, Icon, Mouse and Pull-down/Pop-up menus). With the advent of graphical user interfaces, a new style of user interaction called *direct manipulation* has emerged [Shneiderman83]. Instead of using a command language to describe operations on objects that are invisible, users perform (or request) operations by manipulating objects that are visible on a computer screen. Alongside a new class of word processors called WYSIWYG ("What You See Is What You Get", which require no embedded formatting commands), users are given graphical visual feedback and a sense of control over what is happening on a graphic display. From the direct manipulation of a spacecraft in a video game, to the deletion of a file by placing its icon onto the trash-can icon, the user interaction is direct, visible and graphical. However, as user interfaces are becoming more graphical, interactive and easier to use, their development costs are also higher. It is now recognized that user interface software is often large, complex, and difficult to create, test and maintain [Myers89].

Over the last decade, research and development efforts towards a better or more formalized design of user interface software have been making advances. Since the Graphical Input Interaction Technique (GIIT) Workshop at Seattle (1982) and the User Interface Management Systems (UIMS) Workshop at Seeheim (Germany, Nov. 1983), a number of models and specification methods have been published. The term *User Interface Management System* (UIMS) was first coined at the Seattle workshop. Today, in the early 1990s, implemented UIMSs have been emerging and they are promoting the systematic and automatic creation of user interfaces [Lewis89T], [Lee90].

Graphical user interfaces have been promoted through vendor products. The first Apple Macintosh systems, complete with their window user interfaces, were delivered in 1984. The Macintosh was the first of such systems commercially available to the public and soon gained popularity [Crabb89]. Subsequently, the Sun workstation, with its NeWs windowing system [Leler89] and user interfaces also became popular. By 1988, the new IBM OS/2 systems were delivered with its native Presentation Manager graphics. Whilst the existing PC DOS systems were enhanced with an additional layer of software called Windows, to support graphical user interfaces. The X Window System [Scheifler86] from MIT was first released in 1986, for a nominal charge similar to the spread of Unix in its early days. It is based on the design of X, technically speaking the name of a protocol for sending graphics around a computer network. With the advantages of being device-independent and network-transparent, the X window system emerged as the de facto standard window system [Anderson87]. An X consortium and a company called the Open Software Foundation (OSF) were formed in 1988 to promote X and related software, such as the Motif user interface for Unix. Members of OSF include IBM, DEC, Hewlett Packard and other leading manufacturers. The idea of open software encourages the use of graphical user interfaces amongst software producers. The portability of applications is enhanced, simply by virtue of user interfaces being built on top of the de facto standard window system X [Malhortra89] .

1.2 The need for proper validation

In contrast to the effort made to develop user interfaces, very little effort has been directed towards developing means for their systematic and automated testing. Prototyping has become the only usability assessment practice [Myers89], in both the industrial and academic worlds. The aim of prototyping is to allow users to try out prototypes and to introduce modifications according to their comments [Ehrlich89]. Prototyping is useful as a means of testing the specification of user requirements. It is designed to obtain feedback about the overall usability and acceptability of the user interface. However, the final implementation could be quite different from the prototype. Proper testing is needed to uncover bugs and to establish an acceptable level of confidence in the conformance of the user interface to its specification. Conformance testing is important, as an example, consider the user interface of a fly-by-wire aircraft. An error in the user interface could cause the left engine to be shut down when the pilot meant to give instruction to shut down the right engine! This is not just an imaginary scenario. The simple fact is that a user interface works like a switch box in relaying user requests to hardware functions. The "shutting down of the wrong engine" scenario represents a common class of "cross-wired" faults in switch boxes.

Another example is a new cash dispenser system. During prototyping, users may find the system "works" as it is fairly easy to use, and the right amounts of cash are given during the trial runs. Yet the final system has to be tested systematically and thoroughly to ensure that the correct amount of cash is dispensed at all possible request levels, taking into account cash stocks. It should always produce accurate slip print outs, debit accounts correctly, and cope with various possible user errors. In short, all functions must be tested.

Until now, the testing of graphical user interfaces has usually been undertaken by human testers to exercise the systems' functionality. Often these tests are managed in an ad hoc manner [Winston91]. When a symptom is observed, it may have arisen out of previous interactions, and human testers easily forget such earlier events. Thus the exact cause of the problem is very difficult to determine. It is not an interesting task for any human tester to try to check through a large number of windows and menus. Therefore, it is important that the problems of graphical user interface testing be investigated, with the goal of finding ways towards systematic and automated testing.

1.3 Importance of the subject matter

The study of user interfaces, also known as Man-Machine Interfaces (MMI) or Human Computer Interfaces (HCI), has been recognized as a significant research area in information technology. This is evident in the Alvey program [Talbot85] which aimed to stimulate advances in a number of identified key enabling areas :

- VLSI and CAD
- Man-Machine Interfaces
- Intelligent Knowledge Based Systems
- Software Engineering

Moreover, the validation of user interfaces also falls within the discipline of software engineering, which is another key area in the above list.

Testing and Software Engineering

It is important to understand how software testing fits into the wider field of other software engineering processes. This is best illustrated in the waterfall model [Boehm88], [Royce70] of a software life cycle, reproduced with some simplification in Figure 1.1.

It can be seen that some form of testing is necessary at all stages of the life cycle. The terms "validation", "verification" and "revalidation" are defined fully in Chapter 2. For now they can be looked at as various forms of testing.

This thesis develops an approach to testing that would be conducted after "integration", but prior to software release. It would probably be nearest to the "Product verification" phase in the waterfall model. It is worth noting that "specification" is also a major phase within the life cycle. An important part of this thesis is in the development of a specification approach for the validation of graphical user interfaces.

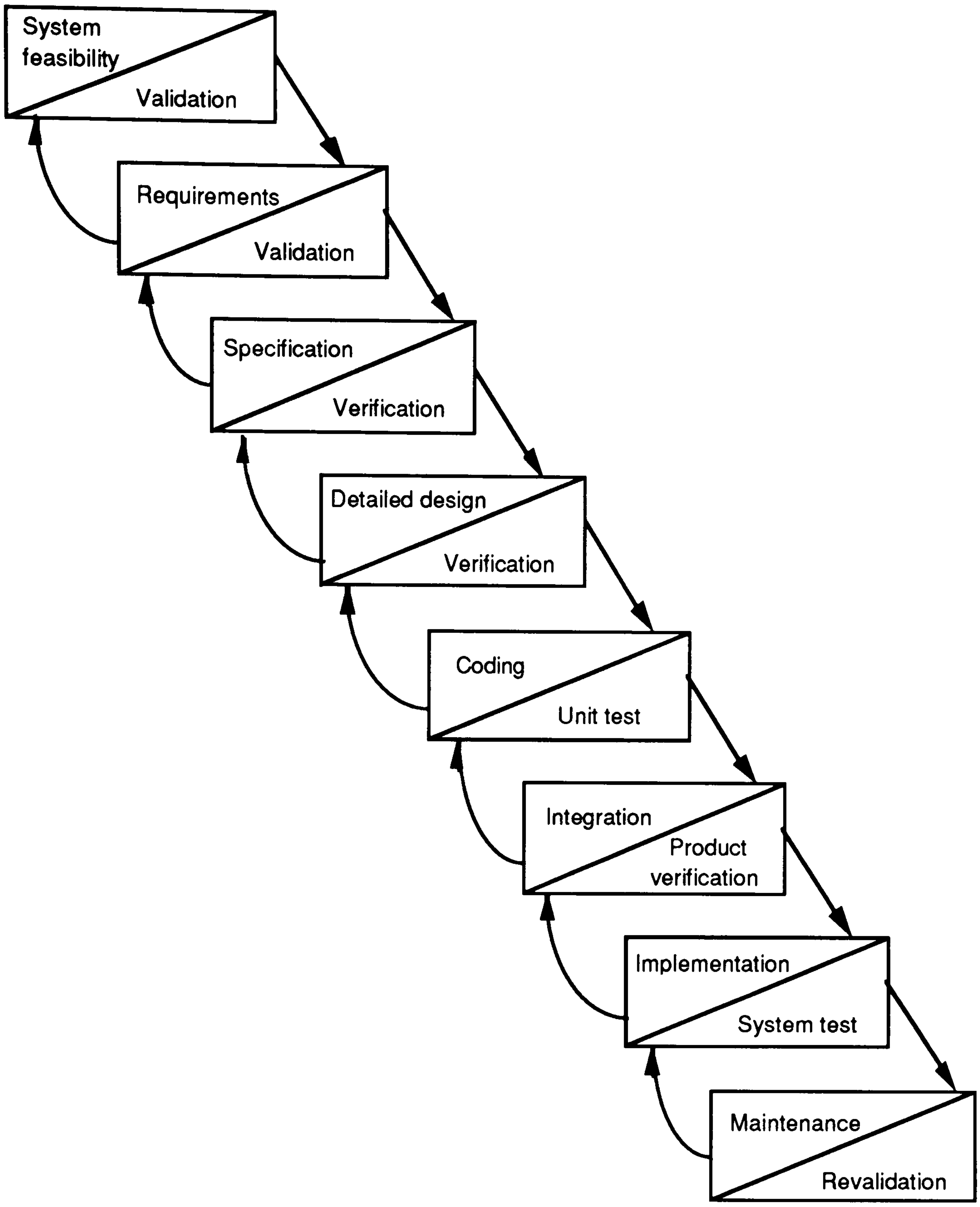


Figure 1.1 The waterfall model of software engineering life cycle

1.4 Originality and contribution of research

Although there has been a rapid growth in the use of window systems, there have been few attempts to provide mechanisms for automating the testing of graphical user interfaces, except the rerun of test suites. There are few (if any) published research reports that actually deal with the root problem of software testing : approaches to generating reliable and effective test cases. This thesis addresses the root problem of test design and generation.

The contribution to knowledge is developed in three parts:

- An original investigation of the problems confronting GUI validation is presented through an analysis of graphical user interfaces (Chapter 3), and surveys of existing testing and specification approaches (Chapters 2 and 4). It also examines the practical needs of validating GUIs in the software industry (Chapter 11).

- The main crux of this thesis is the proposed approach to validating GUIs by means of test cases derived from formal specifications. The approach is substantiated by the development of an original specification method, notation, criteria and algorithms for test selection and test input generation.

- Case studies of actual testing experiments are conducted to assess the proposed testing approach. Evaluations are carried out using both error seeding and code coverage measures. The construction of a number of testing tools is explored to investigate automation issues of the proposed approach.

1.5 Criteria for success

- The specification approach and notation should give a precise and comprehensible description of GUI functions from the human tester's perspective.

- The approach should be applicable to a wide range of user interfaces, possibly on different hardware platforms and window systems.

- The specification, once written, should lend itself to the systematic generation of test cases.

- The capability of the specification approach to model GUI functions appropriately would be reflected in the quality of test cases derived from the specification. A low success rate in error detection, function or code coverage, should call for improvements in the specification method and notations.

Chapter 2

A Review of Software Testing

This chapter aims to give an overview of software testing, rather than exhaustively covering all available knowledge and references. It is intended to introduce and evaluate techniques in software testing so that they can be used in later chapters for tackling the problem of GUI validation. It is also important to give clear definitions of the terms used in software testing.

"Software testing is the process of evaluating a program, with or without execution, to verify that it satisfies specified requirements."

from ANSI/IEEE Std 729-1983,
Standard glossary of Software Engineering terminology.

"Testing is the process of executing a program with the intent of finding errors." from [Myers79]

An *error* is a mental mistake by a programmer or designer. It may result in textual problem with the code called a *fault*. A *failure* occurs when a program computes an incorrect output for an input in the domain of the specification.

From [IEEE83] and [Morell87]

Software testing is defined in this thesis as the process of revealing the existence of *errors* in computer programs, by exposing faults or differences in behaviour or code structure from what is expected. Testing is usually carried out by executing the program under test, or by examination and analysis of the program code and design. *Debugging* is different from testing. Debugging is the process of locating and rectifying the textual faults in the program, design or specification, after the existence of errors has been indicated during testing. Research in software testing has largely been practically oriented with few theoretical works published [Goodenough75], [Weyuker80], [Tutorial81]. This thesis does not attempt to argue for or against the view that "program testing can be used to show the presence of bugs, but never to show their absence" [Dijkstra76]. Instead, this thesis subscribes to the pragmatic view that when testing ceases to uncover some known classes of errors, the confidence in the correctness of a program is increased [Morell87], [Hall88].

The terms "verification" and "validation" are sometimes used confusingly. The ANSI /IEEE Std 729-1983 gives clear definitions, and is used in this thesis. *Verification* is the process of evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase [IEEE83]. *Validation* is the process of testing software or its specification at the end of the development effort to ensure that it meets its requirements (that it does what it is supposed to do). [IEEE83]

Confusingly, verification is sometimes used to mean program proving, which is the specific process of proving (mathematically) that an implementation agrees with its specification.

A few other terms are often used in testing practices. *Module testing or Unit testing* is the process of testing the individual subprograms, subroutines, or procedures in a program . A *test case* is a detailed design, consisting of both the required input data for program execution, and a precise description of the correct output of the program for that set of input data. *A test oracle* is the name given to an external mechanism which can be used to check test output for correctness. Test oracles can take on different forms. They can consist of tables, hand calculated values, simulated results, or informal design and requirement descriptions ([Howden78] in [Tutorial81]). An oracle can exist in the form of a written specification or as a person who has the authority to decide if a program is working correctly [Weyuker82].

Test tools are software tools that assist the testing of programs in different ways , such as analysing program structure, generating test data and recording test execution.

2.1 Testing strategies

The software engineering life cycle (see Figure 1.1), shows that some form of testing must be carried out throughout the life time of a software product. This is in accordance with the belief that the later an error is discovered after it was made, the more expensive it is to rectify. Hence it is very costly to fix a design error that is not discovered until the maintenance phase. This partly accounts for the statistics that show software maintenance is the most expensive phase of the life cycle, as revealed in the following table [Roper87b].

Life Cycle Phase	% Cost [Lientz80]	% Error Fix Cost [Glass79]
Requirements/Specification	10	4.6
Design	10	5.5
Implementation	10	6.5
Testing	20	7.0
Maintenance	50	76.4

As soon as program specifications and design papers are available, they should be reviewed by testers. These early reviews can help to detect errors made in the requirements-analysis process. In order to ensure coordinated validation and verification throughout all life cycle phases, it is vital to draw up a test plan at an early stage. A *test plan* is the overall schedule covering all the different stages of testing, from design reviews and module testing, to final regression testing. It may enlist many test cases designed for individual modules and the program as a whole. *Regression testing or revalidation* is the rerun of some existing tests after changes have been made to a program which had previously been test-accepted. This is to determine if the changes have regressed other aspects of the program.

When the first modules are coded and become available from the developers, module or unit testing is carried out. These program units can then be executed according to some previously designed test cases. Test design involves selecting a small subset of all possible inputs to the module under test. This is because an exhaustive-input test is often impossible [Myers79].

The process of test input selection should be based on all available factual information rather than on coincidence, myth or guesswork. There are two main sources of information about a software product. The source code, if it is available to the testers, and the functional specification of the program. They give rise to two main streams of testing approaches, *structural testing* and *functional testing*. Structural testing is also known as *white box testing*. It is a testing strategy by which the testers, concerned with the internal structure of the program, can derive test data according to their understanding of the program's logic. The program code provides a precise, formal and machine readable notation required for the systematic generation of test data. This is the reason why research has concentrated on white box testing [Ince84].

Functional testing is also known as *black box testing*. It is a testing strategy in which the testers are unconcerned about the internal behaviour and structure of the program under test. They perform tests based on their understanding of the intended function of the program. Unlike program source codes, proper functional specifications are often unavailable, incomplete or mainly written in natural language descriptions. Consequently, functional testing has been carried out in informal and unsystematic ways for many years [Howden81].

Another source of information is the expert knowledge of likely causes of errors [Ostrand84]. This gives rise to an approach called *error-based testing* [Morell87]. It is a testing strategy which seeks to demonstrate that certain classes of errors have not been made in the programming process [Weyuker83]. Error classes may be derived from a history of programmers' errors, measures of software complexity, knowledge of error-prone syntactic constructs, or even error guessing [Myers79].

Once a strategy or a combination of strategies is decided, there are a number of established techniques which can be followed to design test cases. Some of these techniques require the execution of the program and some do not. This is why testing techniques can also be classified as either *dynamic* or *static*. Static Analysis is any testing technique that does not involve the execution of the program under test. Dynamic analysis is any testing technique that requires the program to be executed. Generally, a structural testing strategy can be performed with or without executing the program (i.e. either dynamic or static). Although a functional testing strategy can either be static or dynamic, it is often carried out by executing a program to test its functions (i.e. dynamic techniques are used). A brief description of some structural and functional testing techniques are given in the following sections.

2.2 Structural Testing (White Box) Techniques

Techniques of varying degrees of sophistication exist for the analysis of a program's code structure. The simplest approach is visual inspection by human testers. Complexity of techniques increases from code coverage measurements, anomaly detection, through to tools for proving the correctness of programs.

• *Code Inspection and Walkthrough* (Static)
These are "human testing" methods, involving the reading or visual inspection of a program or module by a small group of people, with the help of the program's author. Questions and reasoning interjected by testers, in the presence of the author, are effective in exposing faults and errors. It appears to be a more effective strategy than

code inspection by the author alone. Uses of code inspections by IBM have shown error detection rates of approximately 80% [Myers79].

• *Anomaly analysis* (Static tool)

There are code analysers which will produce flow-graphs for programs and detect anomalies such as unexecutable code (island code), array bounds, variable initialization, unused variables and labels, jumps into and out of loops. Analysers are automated tools and are therefore more efficient than code inspection by human testers. However they are only sensitive to the raw mechanics of code structure. They are incapable of detecting logic errors that are entirely proper constructs within the rules of the programming language.

• *Code coverage criteria* (Dynamic)

These are a range of criteria requiring increasing code coverage of all program statements, branches, conditions, combinations of conditions, and lastly, all program paths. These criteria are well published [Myers79], [White87]. Brief but original examples are used in the following descriptions.

Statement coverage requires the design of test cases to ensure that every statement in the program / module is executed at least once. This can be seen in the example of a statement such as :

IF hours ≥ 25 THEN employed := "Full time" ;

A single test input of "hours=30" will satisfy statement coverage. Whilst it is a useful and necessary criterion, statement coverage is by no means sufficient. Consider a modification to the above IF statement by the addition of an ELSE statement as given below :

IF hours ≥ 25 THEN employed := "Full time" ELSE employed := "Part time";

The test input of "hours=30" will not cover the ELSE statement, which can only be tested with an input of "hours<25".

Branch coverage requires enough test cases to be written so that each direction of branch (or decision) in the program would have a true and false outcome at least once. For the above program statement, there are two branches, THEN and ELSE. The test inputs of "hours=20" and "hours=30" would test both branches, thus satisfying the branch coverage criterion.

Consider a slightly different program statement :

IF (hours $\geq$ 25) AND (salary $\geq$ 2500) THEN employee := "Taxable"
ELSE employee := "Tax-free" ;

There are still two branches, THEN and ELSE. However the statement now consists of two conditions, "hours $\geq$ 25" and "salary $\geq$ 2500". The inputs according to branch coverage, ("hours=20" and "hours=30", with a fixed value of "salary=3000"), would still exercise both of the branches. However, the condition "salary < 2500" remains untested.

Condition coverage requires enough test cases to be written so that each condition would be tested for a true and false outcome at least once. The test inputs "hours=20" with "salary=2000", and "hours=30" with "salary=3000", would now satisfy condition coverage as well as branch coverage.

The order of combination of conditions is important. If the test inputs "hours=20" with "salary=3000", and "hours=30" with "salary=2000" were used instead, condition coverage would still have been satisfied. However the THEN branch would not have been tested.

Multiple condition coverage requires enough test cases to be written in order that all possible combinations of conditions are tested. A multiple condition coverage would always satisfy both branch and condition coverage. Four sets of test inputs would be required to test the above program statement according to multiple condition coverage :

"hours=20" with "salary=2000" ,
"hours=20" with "salary=3000" ,
"hours=30" with "salary=2000" , and
"hours=30" with "salary=3000" .

Finally, *path coverage* is the strongest code coverage testing technique. It simply requires that all possible program paths be executed at least once. This effectively satisfies all the above coverage criteria [Myers79]. However, there are a number of shortcomings noticeable in code coverage strategies [Coward88a].

- Coverage criteria can assist test input design, but do not provide test oracles to judge the correctness of the output.
- Exhaustive path testing cannot guarantee that the program matches its specification.
- If some required functions of the program have been left unimplemented, code coverage cannot detect the missing paths for these functions.

- Combinations of all conditions can easily result in a situation called path explosion, in which the existence of huge numbers of program paths prevents exhaustive coverage.
- Due to the existence of infeasible paths, coverage criteria cannot guarantee that every path is tested. For example if the program statement used in the last example is followed by :

  IF (employee = "Taxable") THEN ... ;
  IF (employee = "Tax-free") THEN ...;

  Since a path through both of the THEN clauses cannot be satisfied with any possible inputs, it is an infeasible path.

• *Domain Testing* (Dynamic)

Domain testing is a modified form of path coverage. It helps to select a finite set of paths for analysis. Ranges of inputs are deduced from the program structure to establish path domains. This technique reveals errors by picking test data on and slightly off the borders of path domains. Again, using the program statement :

IF (hours ≥ 25) AND (salary ≥ 2500) THEN employee := "Taxable"
ELSE employee := "Tax-free" ;

The path domains have two borders, the two lines representing "hours=25" and "salary=2500". Points on and slightly off the borders are :

hours=25 and salary=2500,
hours=26 and salary=2501,
hours=26 and salary=2499,
hours=24 and salary=2501,
hours=24 and salary=2299.

The ideal path test, which requires execution of all possible paths in a program, is almost always impractical. Domain testing aims to overcome this problem by selecting a limited number of test points. However, its application is restricted to linearly domained programs, according to [White87]. Moreover, the presence of iteration loops in programs may increase test points to an unacceptable number [White87].

• *Symbolic execution* (a test tool, Static)

Symbolic execution is a technique, also known as symbolic evaluation, which does not execute a program in the traditional sense. Symbolic values of input data, instead of actual values, are fed, together with the program, into a tool that carries out symbolic execution. The outcome of symbolic execution is a set of expressions based on the symbolic values of the data. These output expressions represent what the program would have produced as output with the given data, based on the tool's analysis of the program. The output expressions can then be compared with the expected result [King76], [Howden77]. The main difficulties with symbolic execution are in program loop control variables and array element access indexes that are dependent on values of input variables. Module calls and infeasible paths are also problematic for symbolic evaluators [Coward88b].

• *Program proving* (a test tool, Static) :
Program proving is a technique that goes one step beyond symbolic execution. It does not simply analyse the program to see what it does; it is able to decide whether the program is correct in trying to achieve what it is supposed to do. Mathematical assertions are added at the beginning and end of all procedure blocks to specify the inputs and oracles. By checking against the assertions, a program prover is able to analyse the code and determine if it achieves its goals correctly [Hantler76]. Program proving as described above, is related to the mathematical proof approach ([Hoare69], [Floyd67]). Whilst assertions are handled by program proving tools, mathematical proofs are largely carried out by hand. More recently, the mathematical proof approach has been developed during the 1980s into an important part of the formal methods in software engineering [Gehani86]. Mathematical proof approaches use self-contained formal specifications written in well defined specification languages, which is different from scattering assertions alongside source codes as in program proving approaches. Mathematical proofs are often conducted on specifications against invariant statements and on specification refinements [Jones90], or against axioms [Liskov75] to validate the correctness of the design even before the implementation stage. Automation of mathematical proofs is generally pursued through theorem prover tools. However, mathematical proofs are not infallible. Errors can exist in the specification, and in the deduction process of proving the conformance of specification and implementation [Hall91a, b].

• *Mutation Analysis* (Dynamic) :
Mutation analysis is a technique mainly performed by test tools. It requires the production of many *mutant programs* , which are almost identical to the original, from the original program under test. Mutants have very slight code variations, making them subtly different from the original program. The idea is to establish a set of high quality test data, by using these mutant programs to test the test data. If a given set of test data always gave a different result in any mutant program, from the result of the test data on the original program, this set of test data is shown to be of the highest quality. The larger the number of live mutants (mutants that do not produce distinguished results), the poorer the ability of that set of test data to reveal errors [Budd78]. This technique lies between structural and error-based testing. The main difficulty of mutation testing is the enormous number of mutant programs involved, even when a small number of operators (e.g. =, <, ≤, >, ≥, and, or) are considered for mutation.

2.3 Functional Testing (Black Box) Techniques :

As discussed earlier, research has concentrated on white box testing. It is because the program code provides a precise, formal and machine readable notation required for the systematic generation of test data [Ince84]. Nevertheless functional testing techniques are important, as higher error detection rates have been reported with functional testing than with structural testing [Howden76], as illustrated in the following table.

Method	Errors discovered	% of total errors
Path testing	12	43
Branch testing	6	21
Functional testing	17	61

This section discusses a number of well known functional testing techniques, which are extensively covered in [Myers79]. Brief but original examples of the use of these techniques are given to show their strengths and weaknesses.

• *Equivalence Partitioning* (Dynamic) :
This is a technique which attempts to partition the input space, so as to select a small subset of input data from the domain of all possible inputs, aiming to select the subset with the highest probability of finding the most errors. The input domain of a program is partitioned into a finite number of equivalence classes, so that one can reasonably assume a test of a representative value of each class is equivalent to a test of any other value. This approach comes from the fact that an exhaustive input test of a program is ideal, but impossible. Equivalence partitioning helps to select a finite set of input data for testing. The main difficulty is that the identification of equivalence classes is largely a heuristic process. The following gives a small part of a program specification, from which a simple example of equivalence partitioning is developed.

"... An integer no_of_hrs can be inputted at this point of program execution, representing the number of hours the employee is employed each week. A full-time working week is 25 hours or more. No employee works for more than 70 hours, or less than 7 hours a week. The program responds by printing one of the three possible messages :
Employment is full time.
Employment is part time.
Invalid input for no_of_hrs. ..."

Following the concept of equivalence partitioning, four equivalent classes are identified.
no_of_hrs $\geq$ 25, but $\leq$ 70 , this is the valid class of "full time workers".
no_of_hrs $\geq$ 7, but $<$ 25 , this is the valid class of "part time workers".
no_of_hrs $<$ 7 , this is a class of invalid inputs.

no_of_hrs > 70 , this is another class of invalid inputs.

A representative value from each of the four equivalence classes is chosen, producing four different sets of test inputs :
no_of_hrs = 40
no_of_hrs = 10
no_of_hrs = 5
no_of_hrs = 90

• *Boundary value analysis* (Dynamic)
This technique can be seen as a special case of equivalence partitioning. Boundary value analysis requires the selection of test data directly on, above and below the boundary of equivalence classes. Thus it generates more test inputs than just one representative value from each equivalence class. This approach can be used in the result space domain as well as the input space, making boundary analysis different from equivalence partitioning. Referring to the same example used for equivalence partitioning, 9 different test inputs are required to cover the three boundary values (7, 25 and 70) for the input of no_or_hrs:
no_of_hrs = 6, 7, 8, 24, 25, 26, 69, 70, 71.

Experience shows that test cases which explore boundary conditions have a higher payoff. The drawback is that a degree of creativity is required in order to derive boundary conditions from specifications [Myers79]. Another weakness, according to [Myers79], is that only boundary conditions of individual input or output variables are analysed, and no consideration is given to exploring the combination of different input or output variables.

• *Cause-Effect graphing*
This technique relies on functional specifications for the identification of causes and effects [Elmendorf73]. A *cause* is a distinct input condition. An *effect* is an output condition or a system transformation, i.e. an effect that an input has on the state of the program. Causes and effects are identified by reading the specification. Each cause and effect is assigned an unique number. The semantic content of the specification is analysed to link up causes and effects into a graph showing transitions from causes to effects. The Boolean operators NOT, AND and OR are used to connect multiple causes. For instance, the following may form part of a specification.

"... If an employee is employed for not less than 25 hours each week, the employee's income will be taxed. If the employee's income exceeds 2500 pounds or if free full board is given, income tax is applicable irrespective to the hours of employment. ..."
Three causes are identified from the above part-specification :

C1: hour $\geq$ 25

C2: fullBoard = true

C3: salary > 2500

One effect is identified :

E1: employee = "Taxable"

A simple cause-effect graph is then developed in Figure 2.1.

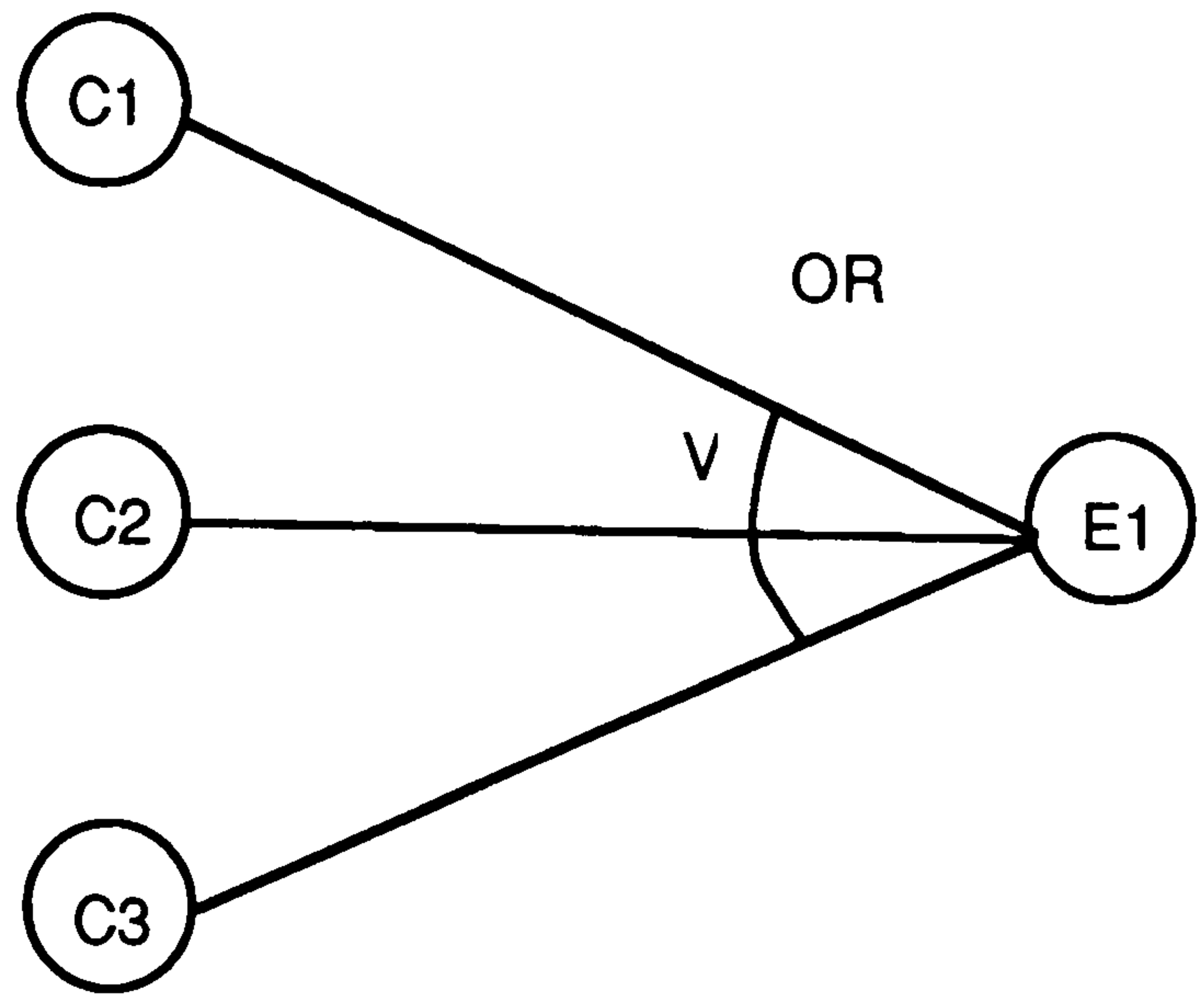


Figure 2.1 A cause-effect graph

Cause-effect graphing overcomes the weakness of boundary value analysis by considering combinations of inputs. However, such complexity is costly and is therefore considered by [Abbott86] as impractical. Cause-effect graphs are often converted into decision tables for the purpose of test generation.

• *Error Guessing* (Dynamic, intuitive) :

The basic idea of error guessing is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list [Myers79]. This is largely an intuitive, ad hoc technique that is often used effectively and subconsciously by experienced programmers and testers. It should not be confused with the more formalized error-based testing approach [Morell87] mentioned earlier.

• *Random Testing* (Dynamic) :

This is a technique used to test a program by selecting at random subsets of all possible input values. Random testing was considered "probably the poorest methodology of all" [Myers79]. An argument against this was published in [Duran84], justifying its effectiveness in terms of the theory of probability. More recent empirical results [Cronin87] indicate that random testing are useful with small programs requiring numerical inputs. [Loo88] reveals that random testing works well on error-prone programs and programs with expected outputs that can be known easily.

2.4 Module, Integration and System Testing

A tester normally chooses a small subset of the techniques discussed above. The decision depends on the nature and properties of the program under test. It is necessary to consider factors such as program size, structure, nature (e.g. real time), severity (e.g. life critical), and the resources available for testing. For example, some very large programs may have problems with symbolic evaluators [Coward88a].

In general, a reasonably rigorous test can be developed by using certain black box orientated test-case-design techniques and then supplementing these test cases with an examination of the program's logic (i.e. using white box techniques). Having successfully tested all the individual modules, the next step in the test plan would be to test the whole program by combining modules together, a process called *integration testing*. Again, a combination of functional and structural testing can be applied. Here, functional tests will be used to examine the overall external functions of the whole program. Structural testing will be used to check the interactions between the component modules (e.g. a 100% subroutine coverage may be used to make sure all subroutines are invoked). Integration testing can be carried out in two alternative ways, incremental or non-incremental.

Incremental integration is to add (or integrate) one module to the program at a time, testing is performed before the integration of the next module. Incremental integration generally results in more thorough testing and earlier detection of interface errors between modules [Myers79]. *Non-incremental integration* is also called "big-bang" integration. In this approach, modules are combined all at once to form an integrated program, before testing is applied.

When the modules are successfully integrated and have gone through integration testing, the complete program (or package) is then subjected to function testing, to see if it performs all its required functions as stated in the product specification.

After integration and function testing, the program is relatively error free (by and large a "working system") and can now be subjected to system testing, such as :

- Installation Testing
- Performance Testing
- Stress Testing

Change or correction of an error in a working program can introduce new errors elsewhere in the program. Therefore, regression testing must be carried out at different stages within bug-fixing cycles of function testing and system testing.

It is a common industrial practice to carry out a *Beta-Test*, before a program (or package) is to be finally test-accepted and ready for release to customers. Beta testing entails the use of the pre-released program in a normal production environment, at a certain selected customer site for a period of time, with proper problem monitoring, problem reporting, debugging and bug-fixing procedures arranged amongst developers, testers and users.

2.5 Summary

Testing is necessary at all stages of the software life cycle. It begins with requirements and design review. It then progresses to module, integration, function, system and acceptance testing, leading to product release. Any subsequent changes during software maintenance are then subject to regression testing, throughout the life time of the software. The testing approach being developed in this thesis is for the testing of user interface functions after integration. These functional tests can constitute part of the acceptance test package, and will most likely form the core of any regression test suites. The proposed new testing approach, named Formal Functional Testing (FFT), is explained in later chapters. Three main points derived from the above survey on software testing were useful in the development of the FFT approach.

• A test oracle is essential;
A program can only be tested properly if the tester knows precisely what the program under test should and should not do. This justifies the requirement for a test oracle in all test cases. Such information, used for deciding if a program is behaving correctly, can generally be derived from the specification of the program.

• Functional testing is often informal and unsystematic;
This occurs because functional specifications are often unavailable, incomplete or mainly written in natural languages that give rise to ambiguity.

• Structural and functional testing are complementary;
Existing testing techniques belonging to the two main strategies (structural and functional) can be combined to create stronger techniques. For instance, equivalence partitioning can be combined with statement coverage.

Chapter 3

Problems confronting the Validation of Graphical User Interfaces

The previous chapter gave a brief review of existing software testing techniques. This chapter aims to consider their application in the validation of graphical user interfaces. Some fundamental questions are useful for a wider understanding of the problem areas.

Q1 - Is a GUI sufficiently different from other types of software to require a separate investigation ?

Q2 - What are the problems of applying existing software validation techniques to GUI ?

Q3 - Are there any theoretical, mathematical concepts or abstract models which could help to reason about GUI software and its validation?

Answers to the above questions will be developed. This chapter begins with an analysis of graphical user interfaces from three different perspectives. From the view of ordinary users, it is a set of display objects (e.g. icons, menus, windows) which provide specific interaction functions. To interface programmers it is a set of window system library routines (e.g. to create windows and display objects). For system programmers, it is an architecture for building application programs on top of window systems and other support software. These three perspectives are discussed further in the following sections.

3.1 Functional perspective

Users perceive a graphical user interface as a means of performing their work through interactions with a set of display objects. Although GUIs are highly interactive and mode-free[1], so far only a few basic types of interaction components are in common use. They are identified below :

- Windows - Text editing windows, terminal emulation windows, etc.
- Icons - For files, file folders, application programs, etc.
- Menus - Pop-up or pull-down menus, and variants such as command buttons, radio buttons, and check boxes.
- Text boxes - Rectangular area where text can be entered.
- Scroll bars - Sliders, dials and other "control panel" component variants.
- Dialogue boxes - Combination of command buttons and text boxes, which may block processing until the dialogue box is cleared.

The most distinctive feature of GUIs is the use of graphical objects to convey meanings and conduct communications between users and their computers. Previously, textual messages, prompts and commands were the main media of user interactions. Lines of text were simply rolled off the top of the screen in the traditional line mode input / output. GUIs use windows and scroll bars to better manage text display. GUIs also entail the organization of the screen layout of windows and other display objects, a process called window management. Most window systems seem to offer a similar set of window management functions [Myers88]. A list of the basic ones are given below :

- Create and destroy display objects (e.g. open and close windows).
- Move display objects around the screen.
- Hide and raise overlapping display objects.
- Resize display objects.
- Iconize windows.

The general functions of user interfaces are portrayed in a number of well known models such as the Seeheim Model [Pfaff85], and SmallTalk's model-view-controller paradigm [Goldberg83]. The Seeheim Model is shown in the following diagram.

[1] This means the user has many choices at every point [Myers89]. See section 3.4.

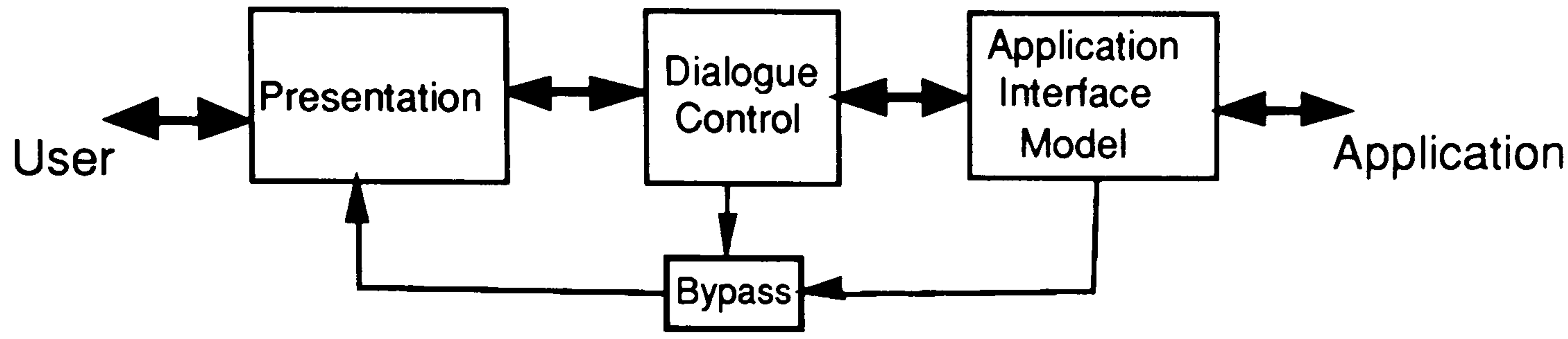


Figure 3.1 The Seeheim Model of User Interface

In the Seeheim Model, the presentation component is responsible for the physical appearance of the user interface, including all device interactions. The dialogue control component manages the dialogue with the user. The application interface model holds the communication between the user interface and the other parts of the application program. The lines and arrows indicate directions of communications. The small box at the bottom is intended for emergency use, to allow messages (e.g. alarms) to be sent to the user rapidly, bypassing normal communication overheads.

3.2 Code-based perspective

A graphical user interface is generally accomplished by a large number of routine calls to an underlying window systems library. The skeleton of a typical OS/2 PM [Petzold89] user interface program is given in Figure 3.2. Another example of an X Windows program can be seen in Figure 3.5.

Before the advent of GUIs, terminal I/Os were performed by the traditional line mode character input / output. I/O interfaces had a very small number of system I/O routines as part of the operating system. The introduction of GUIs has brought additional code complexity. An application program could have a significant increase in code size, when employing a window user interface [Yip91b]. To substantiate this point, consider the "hello world" program using the X Window System [MIT89]. It occupies about 2 to 3 pages of code and comments. In contrast, a typical C program with the conventional character I/O, would only require a couple of lines to print the "hello world" string. The X Windows "hello world" program basically creates a window on the screen and writes the "hello world" string onto the window. Additional lines of code are required to set up the event handling and various attributes for the window. Extra codes are also used to program the desired font, position, size and colour for the character string.

```
/* WELCOME.C -- Skeleton of a typical window user interface program that has an icon,
          it opens a window and undertakes processing according to input events. */
#define ...
#include <os2.h>
int main (void)
{       static ULONG  flFrameFlags =  FCF_TITLEBAR  | FCF_SYSMENU |
                                      FCF_SIZEBORDER ... ;
        static CHAR szText [] = "Welcome" ;
        HAB                 hab;
        HMQ                 hmq;
        HWND                hwndFrame;
        QMSG                qmsg;

        hab = WinInitialize (0) ;
        hmq = WinCreateMsgQueue (hab, 0) ;
        hwndFrame = WinCreateStdWindow   (
                            HWND_DESKTOP,          // Parent window handle
                            WS_VISIBLE,            // Window style
                            &flFrameFlags, ... )   // Pointer to control data

        WinSendMsg ( hwndFrame, WM_SECTION,
                     WinQuerySysPointer ( HWND_DESKTOP, SPTR_APPICON, FALSE),
                     NULL) ;

        while  (WinGetMsg (hab, &qmsg, NULL, 0, 0))
        { switch (msg)
                          {   case     WM_CREATE:
                              [do  initialization]

                              case     WM_PAINT:
                              [paint  the  window]
                              WinDrawText (hps, -1, szText, ...);

                              case     WM_CHAR:
                              [process  keyboard  messages]

                              case     WM_MOUSEMOVE:
                              [process  mouse  movement  messages]

                              case     WM_DESTROY:
                              [clean  up]         }
        }
        WinDestroyWindow (hwndFrame) ;
        WinDestroyMsgQueue (hmq) ;
        WinTerminate (hab) ;
        return 0 ;
}
```

Figure 3.2 Skeleton of a typical window user interface program

A distinctive feature of GUIs is the existence of a main program loop awaiting the next event or user input, as can be seen in the "While (WinGetMsg(...))" statement in Figure 3.2. Another distinctive feature is the existence of call back routines. Call back routines are part of the user interface code, for handling certain pre-declared I/O events associated with interaction objects of the GUI. These call back routines would be given control of processing by the event manager as the I/O events occur. The mode-free nature of GUI user interfaces is generally implemented by asynchronous events and call back routines.

3.3 Architectural perspective

There exist a number of different software levels on which a graphical user interface can be built. They are window systems, toolkits and User Interface Management Systems (UIMS). Figure 3.4 illustrates how they are related to the user interface.

3.3.1 Window systems

A window system consists of a program library that supports the display of objects for user interfaces. It is also the run time system which enables interactions or input/outputs to be performed through display objects.

A well known example is the X Windows System from MIT. (Its structure is illustrated in Figure 3.3.) The X Window System has a library of routines called XLIB, providing more than 200 different routines to be called by window applications.

The X Windows System incorporates a client - server model. Application programs that make calls to XLIB are clients. I/O requests from clients are processed and passed to server programs that carry out these I/O requests on workstations. The library (XLIB) and servers communicate in the X Protocol, over a network if necessary. The library and the server can both, of course, be running in the same workstation. The introduction of the X Protocol between the library and the server is how the X Windows System achieves one of its two main claims, that X is network transparent. The quality of being network transparent is of major significance. Consider a configuration where a heavy cpu-bound application is working out the weather map on a CRAY supercomputer; the user interactions and graphics displays can take place on a number of workstations connected to the supercomputer over a local area network.

In addition to being network-transparent, the X Window System is also claimed to be device-independent. It can be observed in Figure 3.3 that a X Server consists largely of a device-independent part that understands the X Protocol. New devices or workstations can be added to support X simply by creating a new backend of the X Server for that device. Once this is done, all existing applications that use X can be ported to run on this new workstation, without modifications.

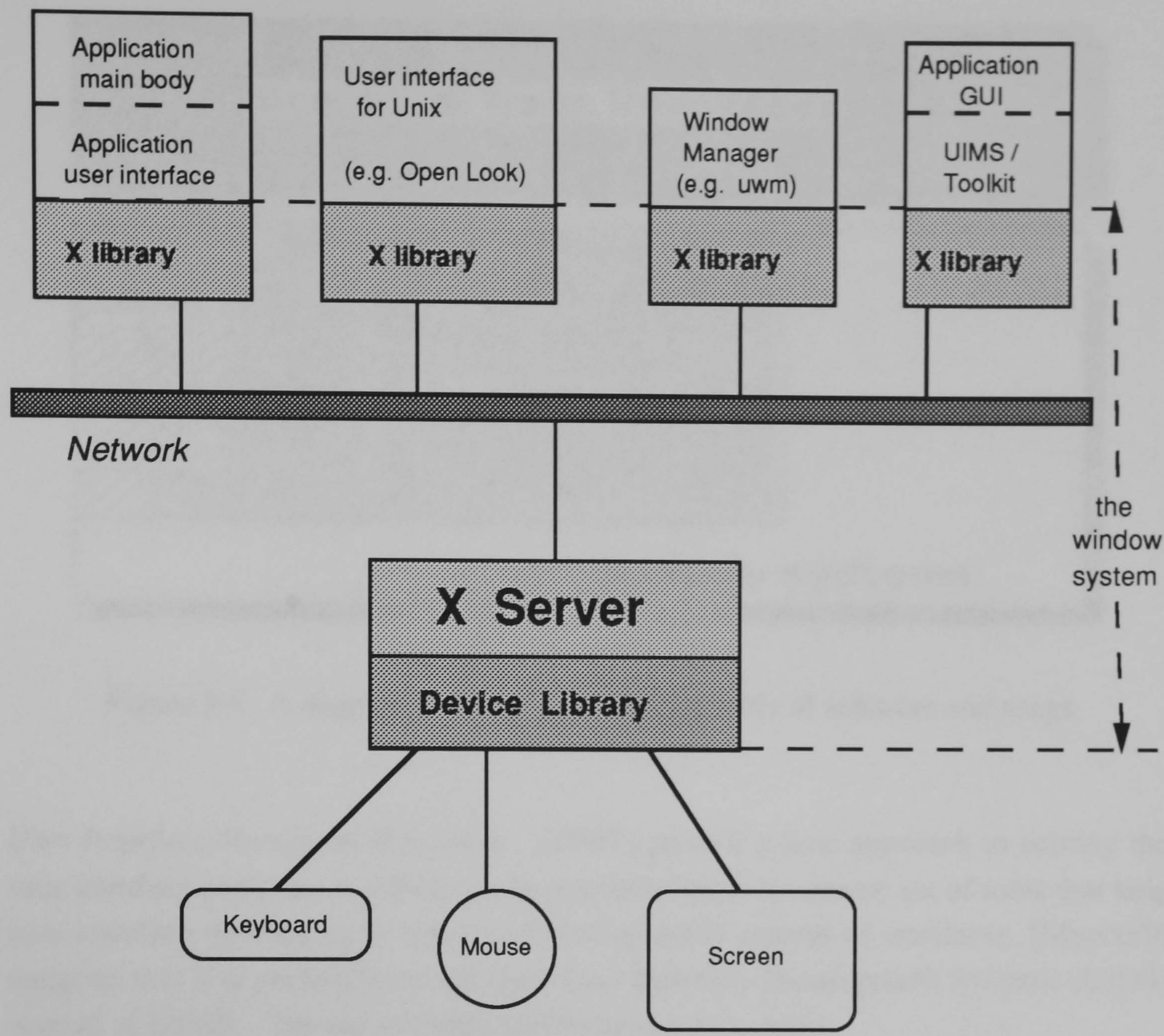


Figure 3.3 The structure of the X Window System

3.3.2 Toolkits and User Interface Management Systems (UIMS)

A window system library can be tedious to use, as it generally provides a programming interface of low level routines. To encourage programmers to use windows, low level routines are built together to form a higher level programming interface generally called a *toolkit* [Myers89], [Hall87]. Toolkits make life easier for programmers, by taking care of small details. They provide a higher level abstraction of display objects called *widgets* [MIT89]. Thus fewer routine calls are required. Toolkits automatically supply default values to some parameters in window library calls. This is why toolkits tend to dictate the "look and feel" of user interfaces. Examples are the X intrinsic toolkit and the Interview toolkit [Linton89], which both exist on top of the X Window Systems. The MacApp framework [Shmucker86] on the Macintosh is another example of a toolkit. Toolkits have become popular through the success of underlying window systems. This can be seen in Figure 3.4 which shows that GUIs can be constructed on top of UIMSs, toolkits or window systems; and UIMSs can themselves be built on top of toolkits or window systems.

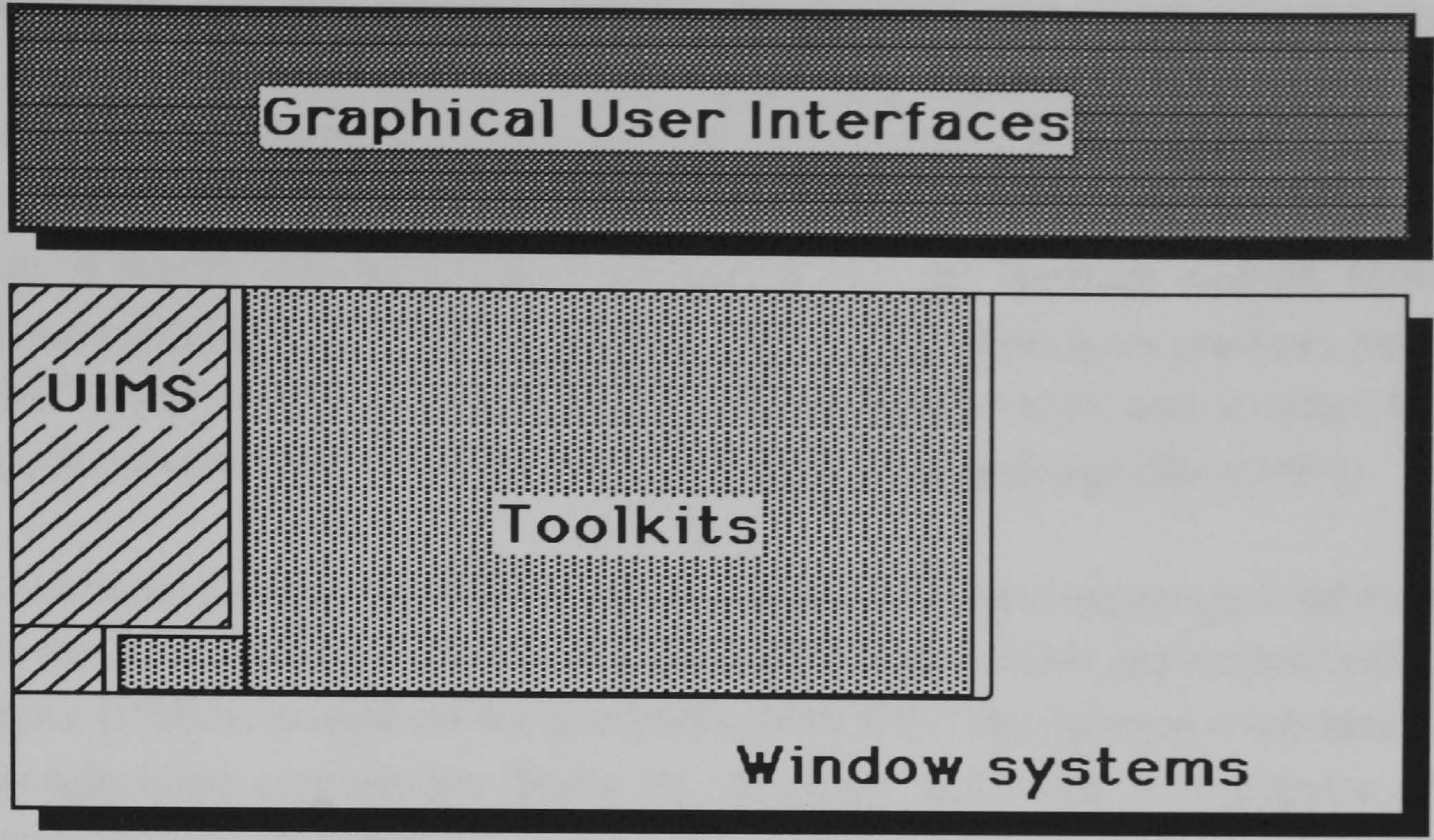


Figure 3.4 A diagram showing the different levels of software and usage

User Interface Management Systems (UIMS) provide a new approach to solving the user interface problem. A UIMS can be perceived as an integrated set of tools that help user interface developers to create and manage many aspects of interfaces. [Myers89] suggests that it is preferable to call them User Interface Development Systems (UIDS) instead of UIMS. The name UIMS will be used in this thesis.

One main function of a UIMS is to support the execution of the user interface at run time. In reality, a UIMS is both a user interface design tool and an underlying window system. For example, the WINDLIB in the University of Alberta UIMS [Green85] is a window-based graphics library package. This explains why a user interface is both closely adhered to and dependent on the UIMS at run time, where the window graphics library of the UIMS can be looked at as part of the run time system.

Some UIMSs support automatic code generation of interfaces, which can then be connected into the main body of the application program ([Gray88], [OSU89]). UIMSs encourage the idea of *dialogue separation* [Cockton86]; that is the dialogue component (i.e. the user interface) should be separated from other components of the application program.

3.3.3 Dialogue Separation

When a user interface is designed as a separate program module, distinct from other components of the application program, dialogue independence is achieved. *Dialogue independence* [Hartson89] means that design decisions which affect only the user interface are isolated from those which affect the other components of the application program. This concept is illustrated in the Seeheim model in Figure 3.1, where the user interface communicates with the application program through an application interface.

Dialogue independence is crucial for easy modification and maintenance of the user interface. For example, if a more meaningful name is invented for an existing menu option, it would only require code changes in the user interface module. However, dialogue separation is difficult to achieve. In a survey of designer practices published in 1987, it was found that 50% of all designers indicated that the user interface had not been considered distinct from the rest of the system during design [Rosson87].

One important feature of a UIMS is that dialogue separation is encouraged and enforced. Often the first step in the process of transforming a monolithic application package to exploit a UIMS is to separate out user interface code (i.e. the dialogue component) from the computation components [Prime88]. Dialogue separation is a popular area of research amongst developers and researchers in user interface management control and communication [Myers89]. An important question to ask at this point is how much processing power should be included in the dialogue component? Where the dialogue component is given processing power, the communication between the dialogue component and the computation component is performed at a higher level, thus the name "macro-communication" [Hartson89]. This is because the dialogue component is able to interpret raw input from the user and communicate interpreted commands and parameters to the computation component. For example, the dialogue component can include the ability to provide the visual feedback locally, simplifying the large communication overhead. Without processing power, the dialogue component has to send all input events to the computation component. This lower level of communication is called "Micro-communication" [Hartson89].

3.4 Testing Graphical User Interfaces

The above analysis has highlighted the need to target tests, since code structure varies at different levels. It was decided that this thesis should concentrate on the testing of application user interfaces. One reason for this is that there are a large number of application programs making use of a relatively small number of window systems, toolkits and UIMS. Vendors of window systems, toolkits and UIMS are better equipped to test their respective products.

As GUIs are highly interactive, the validation of GUIs is difficult to automate. The old automation practice of running a long script in batch mode to exercise programs thoroughly is not applicable to GUIs. The need to generate interactive, position- and timing-dependent test inputs, and to inspect output displays has ruled out batch mode testing. There is also the usual testing need for test case selection and test oracles.

GUIs are interactive and mode free

GUIs are similar to communication programs or other interactive programs, in which an input produces an output (or a change of state). GUIs differ from this class of programs principally in that both the input and output are voluminous and graphical; useful validation can be done by abstracting away some details, and extracting the significant features of the I/O.

Like other highly interactive systems, GUIs are largely mode-free [Myers89]. This means the user has many choices at every point. The partitioning of the screen into different windows and display objects has made it possible for users to change quickly from one mode of interaction to another by moving onto another object or window, thus reducing the restriction of modes. However, this functional requirement of mode-free interactions could easily lead the user interface into a state that has not been foreseen by the designers. This level of complexity can be better handled if interactive functions of user interfaces are clearly written down in a precise and unambiguous manner. It calls for a formal specification of user interfaces.

Visual inspections are essential

A basic I/O function that is vital to GUIs is the movement of the mouse pointer on the screen. Although the tracking of the mouse pointer is mainly achieved by hardware, this basic function is important as most interactions (e.g. selecting a menu option) rely on the accurate mapping between screen positions and the internal (x,y) coordinate representation used in the software. This is one of the main functional differences between GUIs and other interactive programs. Effectively, a GUI has extended the one dimensional input space of command line interfaces to a two dimensional input space, by utilizing the capability of modern display hardware.

The main difficulty in validating this new, position-dependent I/O function is that it requires visual inspection of screen objects. Visual inspection can be time-consuming, tiring and prone to human errors. There are questions concerning whether all locations within the screen map (e.g. 512 x 512 points) should be checked. More importantly, testers need to know the correct shape, size and position of display objects (i.e. presentation attributes) for the purpose of verification.

Window Management Functions

Window managers are usually part of the underlying window system and not part of the user interface. However, in most window management operations, the window manager only makes decisions and draws the window frames. It is up to the application programs to redisplay the window contents upon notifications from the window manager, concerning changes in position, size, overlapping orders and other attributes. Therefore, the testing of window management functions of user interfaces must not be overlooked.

3.5 Structural Testing considerations

The structure of the GUI code varies according to the underlying software: window systems, toolkits or UIMSs. It is vital to determine the right level at which to target tests. It is also important that tests are designed to be reusable.

Static Structural Testing

Most GUI software contains a large number of library calls to the underlying window system. The correctness of window application programs has now become dependent on the parameters and sequences of these routine calls. Information (or rules) about the correct use of parameters and routine sequences are external to the application program. On some occasions this information (or collection of rules) is not always precisely or clearly stated in reference manuals. Since these routines are external to the application packages, it gives rise to difficulties with some structural testing techniques such as code inspection and source analysis. There is also the complication in testing the asynchronous event handling of call back routines. As call back routines are called asynchronously by the window system, they do not fit in the main control flow of the user interface code. The simplest program that uses the X Window System is shown in Figure 3.5 .

```
Xrefresh - Refresh the Screen.

    The following program (xrefresh) is the simplest X application :

%include <X/Xlib.h>
%include <stdio.h>
/*
 * Copyright 1985, MIT
 *
 */

main (argc, argv)
int argc;
char **argv;
{
    Window w;
    if (XOpenDisplay(argc ? argv[1] : " ") == NULL)
            fprintf (stderr, "Could not open Display!0);
    w = XCreateWindow (RootWindow, 0, 0, DisplayWidth(),
              DisplayHeight(), 0, (Pixmap) 0, (Pixmap) 0);
    XMapWindow(w);          /* put it up on the screen */
    XDestroyWindow(w);      /* throw it away */
    XFlush();               /* and make sure the server sees it */
}
```

Figure 3.5 An example application program [MIT89]

This program consists of nothing but routine calls to the window system. Existing code analysers are designed for standard programming language constructs and would not be able to validate these external routine calls. To build a tool that would understand the syntax and semantics of all these routine calls in order to validate them, could require an effort that is comparable to the development of the window system itself. Also, UIMSs and window systems have different program interfaces. A comparison of a small subset of routine names used in three common window systems, namely X [MIT89], OS/2 PM [Petzold89] and the Macintosh Toolbox [Apple85] is shown in Figure 3.6. This exemplifies the fact that variations in routine names in different window systems would hinder the reuse of any general purpose structural analysers, across different platforms. However, these routines from different window systems are seen to provide similar functions.

Dynamic Structural Testing

It is possible to take a dynamic approach (rather than the static code analyser approach) to the structural testing of GUIs. For instance, a statement coverage criterion may be used to ensure code coverage during testing. This may require the tester to validate the behaviour of the user interface at each window library call. This is different from the structural testing of programs that consist largely of arithmetic and logical operations, which tend to give an accumulated result (or output) at the end of execution. User interface programs consist mainly of interactive inputs and outputs which need to be examined during their execution. Since the user interface code consists of many routine calls to the window system, this again requires the detailed understanding of the window system functions.

3.6 Functional Testing Considerations

Structural testing tools and techniques, such as code analysers, are more developed because they are reusable for testing different programs (written in the languages for which the tools are designed). For GUIs, functional testing appears to have the benefit of being generally applicable to different window user interfaces. This is due to the observation [Yip91d] that features and basic interaction components provided by different window systems are very similar even across different hardware platforms.

Ideally, a user interface should provide the same functions, irrespective of the structure of underlying software. A functional specification at the highest level (i.e. at the level of user interactions) encompasses all the required functions of lower level software. For example, when a user interface is ported onto a different window system or hardware, the functional specification of a menu with four options would remain the same, whilst the names and number of routine calls and arguments to set up the menu may change.

OS/2 PM	X Window	Macintosh
#include <os2.h>	#include <X11/Xlib.h>	interface.lib
WinInitialize	Xinit	InitWindows
WinCreateWindow	XCreateWindow	NewWindow
WinCreateStdWindow	XCreateSimpleWindow	-
WinDrawText	XDrawText	DrawText
WinGetMsg	XNextEvent	GetNextEvent
WinQueryPointerPos	XQueryPointer	GetMouse
GpiMove	XDrawPoint	MoveTo
GpiLine	XDrawLine	LineTo
GpiPolyLine	XDrawSegments	-
GpiBox	XDrawRectangle	FrameRect
WinFillRect	XFillRectangle	FillRect
GpiSetColor	XCreateColormap	ColorBit
GpiSetCharSet	XLoadFont	TextFont
GpiSetPattern	XFillStyle	BackPat
GpiErase	XClearArea, XClearWindow	EraseRgn
GpiImage	XPutImage	DrawPicture
WinDrawBitmap	XCreatePixmapFromBitmap	CopyBits
WinShowWindow	XMapWindow	ShowWindow
WinSetActiveWindow	XConfigureWindow	SelectWindow
WinSetFocus	XSetInputFocus	-
WinDestroyWindow	XDestroyWindow	DisposeWindow
WinTerminate	XCloseDisplay	-

Figure 3.6 A Comparison of window library routine names of three window systems

Another reason why research has concentrated on structural testing is that the program code actually provides a precise notation required for the generation of test data [Ince84]. Functional descriptions of programs are often informal and hence unsuitable for the automation of the testing process. However, the advent of formal specifications has now provided a concrete basis for systematic functional testing. Only a very small number of publications, such as [Choquet86], [Hall88], [Roper90], advocate the derivation of test cases from formal specifications. (None of these published works addresses the testing of GUIs.) In this thesis, the name Formal Functional Testing (FFT) is used for this approach.

Display objects and interaction functions
In graphical user interfaces, display objects are given interaction functions. For example, if the mouse pointer is moved inside an icon of a certain program, the clicking of the mouse button at this point would invoke the interaction function to execute that program. To ensure a systematic and thorough testing of GUIs, it is vital that all display objects and interaction functions are identified so that none would escape testing. [Shooman83] has shown that a proper enumeration of program paths is not a trivial problem and is vital to structural (code based) testing.

A specification notation for the enumeration of objects and functions is developed in Chapter 5. The identification and breaking down of a GUI into basic interaction components is a process of functional decomposition [Howden87] for validation purposes. This transforms the testing of the user interface into smaller, more manageable pieces. Interaction objects that are instances of the same basic component are expected to behave in similar (or even identical) manners.

3.7 Tools for GUI testing

One of the early attempts to address the problem of testing interactive systems was the AutoTester project at Wang Laboratories [Leach83]. It pioneered the use of a "record and playback" mechanism to record and replay user inputs for the automation of interactive system testing. Other investigations [Casey82], [Maurer83], [Lewis89R] also proposed the use of Journal Record and Replay (JRR) for testing. However none of the above address the testing of GUIs. There are a small number of commercial JRR products available for recording GUI interactions, such as "Auto Mac" [Microsoft88], "Evaluator" [Elverex89] and “CAPBAK-X” [CAPBAK90] for specific hardware platforms.

3.7.1 Limitations of the JRR approach
A JRR mechanism would only repeat the tests (or interactions) that a human tester had previously carried out by hand. JRR does not help to solve the problem of test case design. Technically there are four problems that are mixed together :

P1 - The design of test cases (i.e. identifying and selecting items to be tested).
P2 - Translating test design into the appropriate format of input sequences.
P3 - The execution of the test cases (e.g. by hand or JRR).
P4 - Evaluation of the results of test case execution (i.e. test oracles).

A JRR tool only provides an answer to the third problem listed above. This thesis proposes a solution to P1 in the use of a formal specification that will identify all items to be tested for test case generation purposes. This approach would also answer part of P4, as test oracles can be obtained from specifications.

3.7.2 Visual verification

In practice, visual inspection by human testers has so far been the approach to P4. Some research [Johnson87], [Islam89] has made attempts to validate GUI screen outputs by comparison with previously recorded bitmaps. This approach is often called visual verification. It has a number of difficulties:

- In deciding suitable check points where snapshots of screens have to be taken.
- Large storage space requirement for bitmap files.
- Screen images are sometimes shifted by a small number of pixels, and transient displays such as time and date can also cause problems during bitmap comparison.
- Minor changes in layout of display objects would invalidate test cases.

This thesis proposes that the actual visual appearance of display objects be included in specifications to form a special kind of state transition diagram called *WinSTD*. A WinSTD would be used by human testers for checking the visual appearance of objects, as well as for identifying interaction objects and functions for testing.

3.7.3 Input synthesis

A solution to P2 is emerging, called input synthesis. *Input synthesis* is an approach which simulates keyboard and mouse inputs, so as to relieve human testers from having to execute tests in generating inputs by hand. The journal file of a JRR mechanism can provide the first step towards input synthesis. New interaction sequences or changes to the recorded sequences can be produced by providing editing facilities on the content of the journal file [Johnson87]. Another step forward would be to generate the contents of the journal file by other means than recording, such as derivation of test cases from specifications. Release 4 of the X Window System [MIT89] contains an "Input Synthesis Extension Proposal" to allow the client program to generate user input actions without the user. It will also allow the client program to control server actions in handling user inputs. This proposal gives a programming interface to simulate user inputs. However, there are synchronization problems with input synthesis that is no easy task to resolve [Islam89], [Jamison90], [Coutu90].

3.8 Review and Decision

In this problem analysis, graphical user interfaces have been examined from various perspectives. GUIs are found to be different from other software. Functionally, a GUI pioneers the graphical communication between humans and computers. It has transformed the traditional line mode character I/O into a new two dimensional, position dependent, mode free, graphical and textual I/O. Structurally, GUIs are distinctive in having event-wait loops and call-back routines. The software levels of window systems, toolkits and UIMS have presented interface designers and testers with difficult choices. This thesis will concentrate on the testing of user interfaces. A functional testing approach is preferred to a structural testing approach. The following summarizes the reasons for the decision;

- Routine calls to window library and call-back routines can give rise to difficulties with some structural testing techniques, as information external to the application packages is required.

- Variations in programming interfaces and routine names of window systems, toolkits and UIMS hinder the development of a general, reusable structural testing approach.

- Program code used to be the only source of the precise notation required for test data generation. The advent of formal specifications has provided an alternative.

- A functional specification at the highest level (i.e. at the level of user interactions) would encompass functions of lower level software (i.e. window system, toolkit or UIMS).

- Functional tests can be reusable, since features and basic interaction components provided by different window systems are very similar even across different hardware platforms.

- Functional testing has the advantage over structural testing in that test oracles can be derived from specifications.

- Higher error detection rates have been reported under functional testing than under structural testing [Howden76] (see Section 2.3).

Chapter 4

Survey of Specification Methods for User Interfaces

The previous chapter established that a functional testing approach for graphical user interfaces is the goal of this research. A functional testing approach is dependent on specifications, preferably formal ones, for derivation of test inputs and oracles. This chapter surveys existing specification methods, in order to evaluate whether any of them can be used for test case generation. It presents a broad overview of a number of published user interface specifications in this section. Discussion is then focused on three main representative approaches in the following sections.

There are a number of published works on the application of formal specification methods to user interfaces, graphics, or interactive programs. Confusion often arises from the languages and interfaces associated with interactive systems . A working group at the Seillac II workshop addressed this issue [Mallgren82] :

"In the interactive world, we distinguish two interfaces to the computer. The first between the user or operator and the computer is called the User Interface. The second between the programmer of the system and the computer is called the Program Interface. Each interface needs a Specification Language. In addition, the User Interface provides a means to communicate with the computer by using the Dialogue Language. The Dialogue Language is handled by its counterpart on the programmer side: the Programming Language."

In this definition, the research work of [Mallgren82] and [Duce86] is on specifications for program interfaces. [Mallgren82] presents a formal specification of interactive graphics programming languages. [Duce86] discusses the formal specification of GKS[1] output primitives. More recently [Purvis90] reports an investigation, and poses the question : "Is the specification of GKS feasible using LOTOS[2] ?". This investigation explores the details of specifying interactions such as mouse operations, in terms of concurrent processes and communication channels. Whilst it is an elegant approach to modelling mouse interaction, it is on a level of abstraction that encompasses too much detail for the purpose of test data generation.

There are a number of published works on dialogue specification. [Hekmatpour88] addresses the prototyping of user interfaces from formal specifications. [Harrison90] gives a collection of papers on the formal specification and analysis of user interfaces, from design, human factors and technical perspectives. [Arthur87] examines menu-based systems. The formal specification of text editing is discussed in [Sufrin82] and [Chi85].

[Parnas69] first suggested the use of state transition diagrams (STD) to describe interactions between the user and the computer. The use of STD, BNF-like grammar and event languages are detailed in [Jacob86] and [Green86]. [Alexander86] proposes the use of CSP [Hoare85] embedded in a functional programming language called "me-too", for the prototyping of user interfaces. This can be seen as a form of an event language approach. [Marshall86] discussed the use of VDM [Jones90] together with a form of STD called State Charts [Harel88] for the formal description of user interfaces. Marshall's work aims to show that user interactions can be specified in a formal language like VDM. It gives a highly mathematical account and achieves some formal proofs. However, it does not go beyond the simplest academic study of a logon user interface. Neither [Alexander86] nor [Marshall86] deal with the graphical, or testing aspects of user interfaces.

[Abowd90] and [Harrison91] describe a "constructive approach" together with a general survey of existing formal methods for HCI. This approach uses the "Agent Model", which is less abstract and more constructive than other specification approaches. It is intended to aid the design and implementation of interactive systems, but not for testing.

[1] GKS stands for the Graphics Kernel System, it is the international standard for 2D graphics.

[2] LOTOS has recently been approved by the International Standards Organization (ISO) for the description of Open Systems Interconnection.

4.1 The use of State Transition Diagram

For some time research effort has concentrated on specification of the dialogue control component. (See the Seeheim model in Figure 3.1 .) Jacob, in 1983, investigated two formal specification methods: transition diagrams and a grammar similar to BNF. [Jacob83] concludes that specifications based on state transition diagrams are preferable, as they can show an explicit time sequence more clearly than grammar specifications. An Example of Jacob's specification for a scroll bar interaction is shown in Figure 4.1. It illustrates that it is possible to produce formal specifications for interactive user interfaces.

Referring to Figure 4.1, the input tokens are listed first. The iENTER token represents the input action of the mouse pointer entering the scroll bar area. This is the initiating point of the scroll bar interaction. All scroll bar interactions must begin with the iENTER input. The simplest example of an interaction would be an iENTER followed by an iEXIT, without the mouse button being pressed. This can be seen in Figure 4.1. The outcome is no display changes in either the scroll bar or the text associated with the scroll bar.

The next example of a user input is that the iENTER token is followed by the iMOUSEDN input token. The expected outcome is an output token called oSHOWBAR, which displays the scroll bar at the pointer location where the mouse button is pressed down. When the mouse button is released, the associated text will be scrolled.

Jacob's specification approach appears to be viable for test case generation. The interactions described above can be written down to form test sequences listed as follows :

1 iENTER <followed by> iEXIT
expected outcome : none

2 iENTER <followed by> iMOUSEDN
expected outcome : oSHOWBAR

3 iENTER <followed by> iMOUSEDN <followed by> iMOVE
expected outcome : oSHOWBAR

4 iENTER <followed by> iMOUSEDN <followed by> iMOVE <followed by> iMOUSEUP
expected outcome : oSHOWBAR <followed by> oSCROLLTEXT

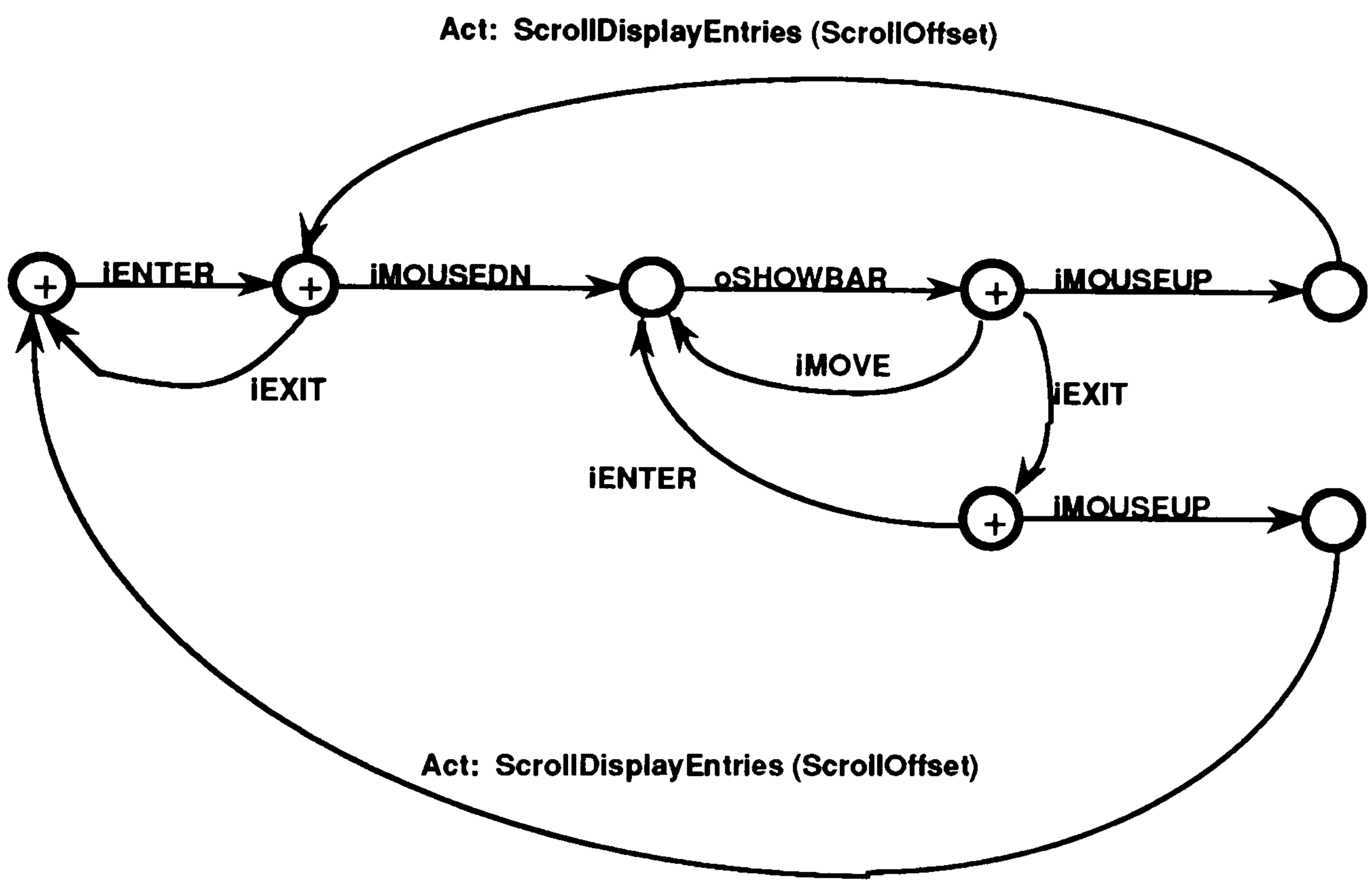
INTERACTION_OBJECT ScrollBar is

...

TOKENS:

iENTER	{ --Mouse enters scroll bar area-- }
iEXIT	{ --Mouse leaves scroll bar area-- }
iMOUSEDN	{ --Mouse button pressed down, -- sets ScrollOffset := scaled X coord of mouse-- }
iMOUSEUP	{ --Mouse button released-- }
iMOVE	{ --Any mouse motion within boundaries of scroll bar area, --return scaled X coordinate of mouse in ScrollOffset-- }
oSHOWBAR	{ --Fills or erase bar up to location corresponding to ScrollOffset--}

SYNTAX:



end INTERACTION_OBJECT;

Figure 4.1 Specification of a scroll bar according to Jacob's method [Jacob86]

Test case generation from Jacob's STD, in terms of states, inputs and outputs, is similar to the functional testing technique called cause-effect graphing (see Chapter 2). They both attempt to describe input events (or causes) and the corresponding state transition and outputs (or effects).

The cause-effect graphing approach was considered by [Abbott86] as impractical. The STD approach was described [Myers89] as a maze of wires easily running outside the boundary of the drawing paper when used for complex systems. In view of the increasing emphasis on modular programming [Wirth71a] and dialogue separation, the augmented STD approach would be practical when used for manageable pieces of individual interaction components. It was for these reasons that Jacob later modified his model [Jacob86].

The main limitation of Jacob's STD approach is the lack of description of the physical visual appearance of interface objects for validation purposes. So far as the functions of a scroll bar are concerned, the specification does not show, for example, how the scroll bar looks and where its control regions are. It also becomes apparent that a state diagram (of nodes, arcs and labels) cannot convey all the necessary information for a specification. In Figure 4.1 textual declaration of input and output tokens are prefixed to the state diagram, where comments are used to convey the semantic meaning (i.e. operations) associated with these tokens. It is vital that semantics should also be specified formally (e.g. with pre- and post- conditions) for specifications to be usable as test oracles. For instance, it is important to state the number of lines which are to be scrolled. This can be expressed in terms of an integer variable being assigned a value proportional to the movement of the scroll bar slider. This movement should be measured from the first location of the mouse pointer, where the mouse button is pressed, to the second location where the button is released. The distance moved should then be divided by the full length of the scroll bar, and multiplied by the total number of lines of text to give the number of lines to be scrolled. A picture of the scroll bar undoubtedly assists the above description, and the use of pictures is advocated in this thesis as an essential part of a GUI specification. Figure 4.2 shows the visual appearance of a scroll bar, as part of an editing window.

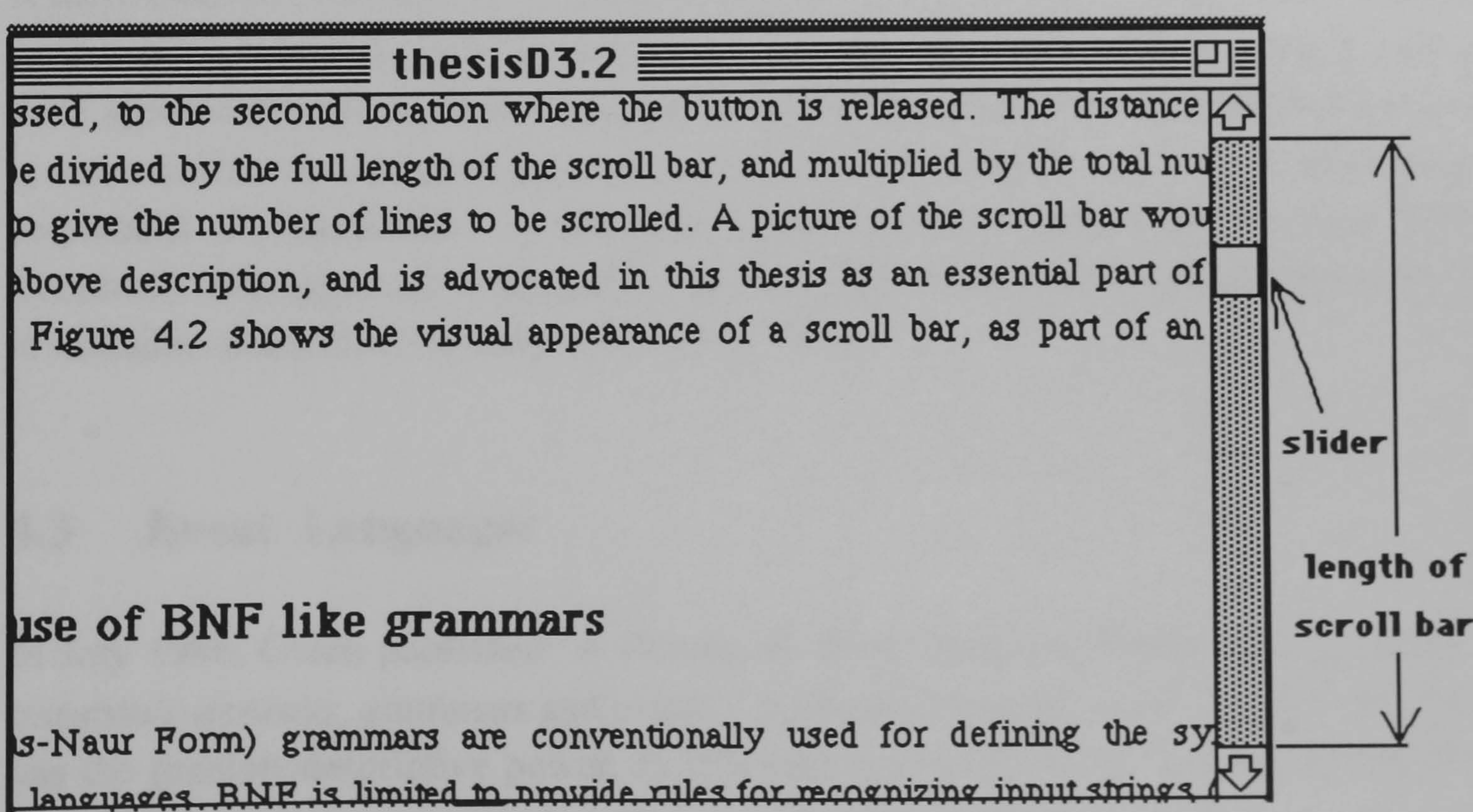


Figure 4.2 Visual appearance of a scroll bar as part of an editing window

4.2 The use of BNF-like grammars

BNF (Backus-Naur Form) grammars are conventionally used for defining the syntax of programming languages. BNF is limited to the provision of rules for recognizing input strings (i.e. the syntax). It is not able to indicate the executable actions (or semantics) for the recognized input. Often a BNF grammar is augmented with notations to specify actions to be taken when a phrase or sentence of the input language is recognized, as used in compiler-compilers such as YACC [Johnson81]. This kind of extension has also made BNF usable for specifying user interfaces, where a dialogue consists of both user inputs and actions (or outputs) from the computer.

A simple specification example of a logon dialogue [Alexander86] is given to illustrate the use of BNF .

```
<logon>       ::= LOGON (1) <user-id>
<user-id>     ::= <bad-user> | <good-user>
<bad-user>    ::= %USER (2)
<good-user>   ::= %USER (3)

          where (1) output:    "user name?"
                (2) condition: not REGISTERED-USER (%USER)
                    output:    "invalid user name, try again"
                (3) condition: REGISTERED-USER (%USER)
```

A more extensive example of BNF specifications for a non trivial interface can be found in [Jacob86], where it is also shown that an STD specification can be generated from a BNF specification of the same interface. As a BNF specification is machine readable, it is more usable for test case generation, than an STD. However, an STD can display sequences of interactions to software engineers, more perspicuously than BNF [Jacob83]. This argument is echoed in [Marshall86], which uses a special form of STD to visualize interactions formally specified in VDM.

4.3 Event Languages

In July 1986, Green published "A Survey of Three Dialogue Models", investigating transition networks, grammars and events. [Green86] concludes that the events model has the greatest descriptive power, as it is suitable for handling both sequential and asynchronous dialogue control. The use of callback routines is the most popular method for handling asynchronous input events [Green85, 86].

```
Eventhandler login Is

 Token
   Keyboardstring s;

 Var
   int state = 0;
   string user_id, password;

 Event Init
   { print "login:"; }

 Event s : string
   { if (state == 0)
       { user_id = s;
         state = 1;
         print "password:";
       }
       else
       { password = s;
         state = 0;
         process_login (user_id, password);
       };
   }

End login;
```

The specification of the Event Handler for the login sequence shown here, is as used in the University of Alberta UIMS. The event language used for the specification looks similar to a C program. According to Green, the expressive power of event languages is greater than that of transition networks or BNF-like grammars.

An event specification like this one can be understood and checked by interface designers with no more difficulties than BNF-like grammars. This method of specification gives precise and complete information for testing the user interface specified. It also allows the possibility of getting tokens from the event language compiler for automating test data generation.

Figure 4.4 Event handler for the login sequence in the Alberta UIMS [Green85]

From the view of testing, an executable event language is a weakness. One of the desirable properties of a formal specification language is to be independent of any specific implementation, so that an implementation can be validated by checking it against the specification. More importantly, Green's event language does not cater for the graphical aspects of user interfaces.

4.4 Requirements of a user interface specification

This chapter has surveyed specification methods used in a number of user interface systems, leading to the conclusion that no one single method is satisfactory in providing all the necessary information for test case generation. A user interface specification suitable for testing and software engineering purposes should include detailed and precise information covering three areas :

- **Presentation** (attributes of display objects)
  It is important for human testers to know the visual appearance of objects for verification.

- **Syntax** (rules governing interaction sequences or dialogue)
  It is helpful for testers to be able to see clearly the control flow of interactions.

- **Semantics** (specification of operations or functions)
  It is vital that functions are specified in a precise and unambiguous notation.

Perhaps one of the main problems facing testers is that such an ideal specification does not normally exist. A formal specification approach aimed at satisfying all of the above requirements is developed in the following chapter.

Chapter 5

A contribution to the Specification of Graphical User interfaces

"A specification is formal if it is written entirely in a language with an explicitly and precisely defined syntax and semantics. Examples of suitable formal languages are first order predicate calculus and a programming language for which the semantics has been defined ... However, a program should not be its own specification, because this eliminates the redundancy need to make verification meaningful. An independent description of desired behaviour is always required."

From [Liskov79] in [Gehani86]

The previous chapter gave a survey of existing specification methods for user interfaces, leading to the observation that a new formal specification approach is needed to aid the testing of graphical user interfaces. This chapter presents an original contribution to GUI specification in terms of a special form of state diagram called WinSTD, in conjunction with a language called WinSpec. The example of a logon user interface is used. The application of this specification approach in the generation of test cases (for the logon user interface) is given in Chapter 7.

This specification approach was developed to describe the required inputs and expected outputs of the functions of a graphical user interface. A set of notations with precise and unambiguous meaning is clearly an advantage over a natural language description of a function. The most obvious feature of WinSpec is the use of primitives to describe visual output. In this chapter, section 5.1 and 5.2 introduce WinSTD and WinSpec. Section 5.3 describes the notations and constructs of WinSpec. Section 5.4 attempts to introduce WinSpec in a more formal manner. It explains how the language is based on set theory and predicate logic, and shows that WinSpec has well defined syntax and semantics, and thus is formal. Considering the classification of formal specification approaches as discussed in [Liskov79], the WinSpec specification language is nearest to the category of "Input / Output Specifications" within the class of "Procedural Abstractions" [Liskov79] .

5.1 WinSTD

For graphical user interfaces, as mentioned earlier, there is one additional requirement in the specification of presentation attributes of display objects. As revealed by the recent interest in visual languages [Harel90], [Shu89], visual information, like the appearance of a menu or an icon need not be specified in yet another textual language (e.g. {ATTRIBUTES; ...; label_text:"OK"; width:50; height:20; x:160; y:75; METHODS: ...} as used in the Serpent UIMS [CMU89]).

A WinSTD is a special State Transition Diagram which shows the visual appearance of display objects linked together by arcs that represent the interaction functions. Effectively, the display objects or components are the nodes (embracing states) in the user interface specification, and the interaction functions (arcs) indicate state transitions. In a WinSTD, every display object (and components), as well as functions (arcs), are enumerated with a unique name. Although similar, a WinSTD is strictly speaking not a finite state diagram. A WinSTD does not expose all the possible states and transitions. Consider an object OBJx which is an editable text box. Strictly speaking, OBJx is in different (visual) states when there are different texts inside the text box. For example :

State1 : text(OBJx) = "abcde"

State2 : text(OBJx) = "xyz"

are two different states. It would be impossible for a WinSTD to capture all states.

A WinSTD illustrates the main control flow of a GUI and thus aids the comprehensibility of the corresponding WinSpec. A WinSTD also provides the initial placement (i.e. xy coordinates) of display objects. It assists the enumeration and identification of all display objects for testing purposes. Considering the complexity and flexibility of window user interfaces in real practice, it is confusing (if not impossible) to include all feasible combinations of interactions in one or more diagrams [Myers89].

However, at least one state for each object must be included in the WinSTD. The specification of window management functions (such as moving a window by dragging) are not illustrated in a WinSTD as they are mostly provided by the underlying window system and not part of the application user interface. The use of WinSTDs is illustrated with the example of a logon user interface as shown in Figure 5.1. In this WinSTD for the Logon interface, all display objects are shown. This includes an icon for the Logon interface, a dialogue box for entering username and password, a dialogue box to inform about logon failure, and a terminal window for successful logons. The interaction functions associated with each of the objects are identified, such as :

F1 - Invoke the Logon interface
F2 - Keyboard inputs in the username text box
F3 - Keyboard inputs in the password text box
F4 - Mouse input to select the "OK" command button
F5 - Display of "Logon Failure" dialogue box
F6 - Mouse input to select the "Reset" command button
F7 - Display of a terminal window
F8 - Mouse input to close a terminal window

Interaction sequences can be expressed in this notation, e.g. :
F1 o F2 o F3
(Function F1 then F2 then F3)

The notation used in this WinSTD example is further explained in the next section, which introduces the WinSpec notations. An exploratory version of a tool called the WinSTD editor, for the construction of WinSTDs, is described in Chapter 10.

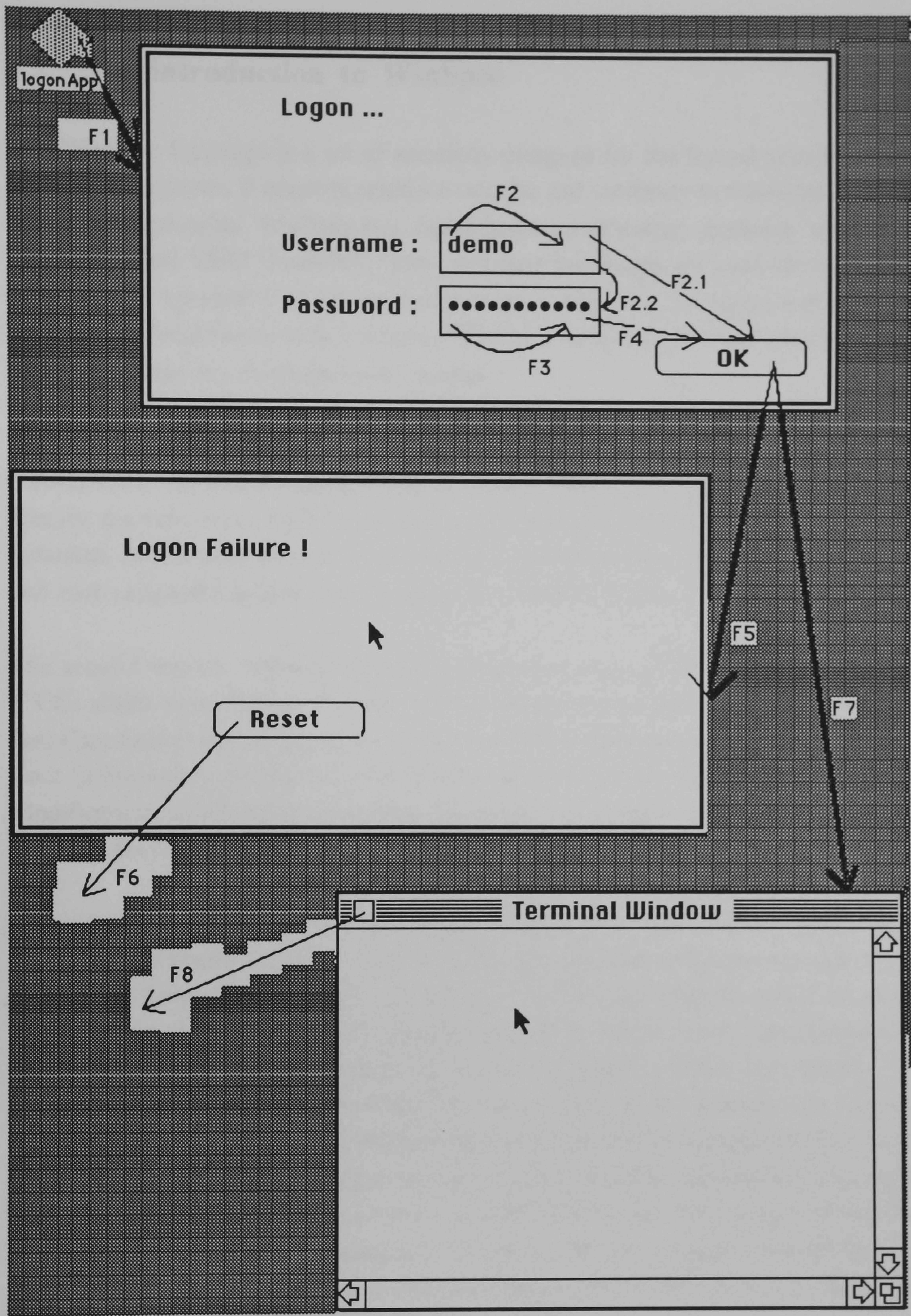


Figure 5.1 A WinSTD for the logon user interface

5.2 An introduction to WinSpec

The WinSpec language is a set of notations designed for the formal specification of interaction functions. It employs predicate calculus and set theory to minimize ambiguity and misinterpretation. WinSpec is a model-based specification approach, similar to Z [Spivey89] and VDM [Jones90]. States and state predicates are used for the implicit specification [Jones90] of interaction functions, to allow the behaviour of a user interface implementation to be checked. WinSpec has special constructs for abstracting GUI interactions in a comprehensible manner.

The version of WinSpec presented in this thesis is the outcome of reiterations of modification and improvement. The first version [Yip91a] was an original attempt to specify the functions of GUIs by stating pre- and post-conditions in a small set of notations. In this early version, display objects and interaction functions were identified and each assigned a unique numeric name (e.g. OBJ01, F123).

The second version emerged during the implementation of the Test Case Generator (TCG) which turns the specification of a GUI into required inputs to test the GUI. The Test Case Generator includes a parser [Schreiner86] constructed using lex [Leek81] and yacc [Johnson81] for lexical and syntactical analysis of the WinSpec notations. Modifications were introduced to make the notations acceptable to the syntax parser. For example, the "$\wedge$" symbol is changed to "and".

As a consequence of the refereeing process of [Yip91a] and other reviews, further changes were made. The final version of the specification notation uses visual state predicates. Some symbols are replaced by more comprehensible notations (e.g."-OBJxx" is changed to "is_not_visible(OBJxx)"). Furthermore, enumerations of objects and functions are replaced by more meaningful generic type names. (For example cBtn_'Reset' instead of OBJ211, where cBtn_ is the generic type name for "**c**ommand **B**utto**n**s".) The specification approach can now be completely described in terms of logic, sets and mappings (see Section 5.6). More development is required to update the parser (built on lex and yacc) and the TCG for this new version of WinSpec for test case generation. This work will be left as a future extension beyond this PhD thesis. The main idea of Formal Functional Testing (FFT) for GUIs can largely be explored without a full implementation of these tools.

5.3 Basic theories

Before introducing the WinSpec specification language, it is necessary to give a brief definition of the notations to be used.

5.3.1 Set Notations

A set is an unordered collection of distinct objects [Jones90]; set values are marked by braces. For example, $\{a,b\}$ is a set, and $\{a,b\} = \{b,a\}$.

In this example, the set has two distinct elements, namely a and b. The number of elements of the set, denoted as $|\{a,b\}|$, is 2. There is no concept of the number of occurrence of an element in a set. Elements are either present ($\in$) or absent ($\notin$). Thus:
$a \in \{a,b\}$, $c \notin \{a,b\}$.

Consider three sets A, B and C. Where, $A=\{a,b\}$, $B=\{a,b\}$, and $C=\{a,b,c\}$.
A is said to be a *proper subset* of C, denoted as $A \subset C$. (Obviously, $B \subset C$ as well.)
Furthermore, the denotation $A \subseteq B$ can be used to indicate that A is either a subset of B, or A is equal to B. (In this case, it is true that $B \subseteq A$ too, as A=B.)

The set denoted by $\{\}$ is the empty set. Apart from simple enumeration of their elements, sets can also be defined by "set comprehension". This is to define a set which contains all elements satisfying some property. For example:
$\{i \in Z \mid 1 \leq i \leq 3\} = \{1,2,3\}$, where Z is the set of integers.

Another example, of the set of even numbers, can be denoted as:
$\{e \mid e=2*n \bullet n \in N_1\}$ where $N_1 = \{1,2,3, ...\}$.
Here N_1 is the set of natural numbers starting from 1 (i.e. excluding 0).

The set of prime numbers can be denoted as :
$\{p \in N_1 \mid p \bmod n \neq 0 \ . \ \forall n \in N_1 - \{1,p\}\}$
Here "mod" is the modulus operator. "p mod n" gives the remainder in dividing p by n. ($\forall$ is the universal quantifier of predicate logic, see section 5.3.3.) "$p \bmod n \neq 0$" states the property of p, that p is indivisible by any natural numbers, other than 1 and p itself ($\forall n \in N_1 - \{1,p\}$). The notation for "set difference" as in $N_1 - \{1,p\}$, states that n is any element from the set N_1, except the two elements 1 and p.

5.3.2 Propositional logic

A proposition is a statement of some alleged fact which must be either true or false [Woodcock88]. For example, the statement : "1 is an even number", is false. However, the proposition : "2 is an even number", is true.

A number of operators are available in propositional logic for the construction of compound statements. The logical operators commonly used are :

- Logical and (denoted as $\wedge$)
- Logical or (denoted as $\vee$)
- Negation (denoted as $\neg$)
- Implication (denoted as =>)
- Equivalence (denoted as $\Leftrightarrow$)

For instance, consider the two propositions :
P1: "1 is an even number", and
P2: "2 is an even number".
The "logical and" of these two propositions is denoted as P1 $\wedge$ P2. The definition of logical operators, and their associated truth tables, are widely available in the literature ([Woodcock88], [Jones90], [Alagar89]) . They are not discussed further here.

5.3.3 Predicate logic

Predicate logic is similar to propositional logic. Propositions, such as P1: "1 is an even number" , and P2: "2 is an even number", are restrictive. Predicates allow flexibility in the choice of the objects in the proposition. For example, is_even(x) , is a predicate. Free variables or place holders [Woodcock88] are used in predicates, x in this case, which can be filled in by the names of suitable objects to create propositions. Thus, the proposition is_even(1) is formed by substituting 1 for x. If x=2 is used, it gives the proposition is_even(2).

A predicate itself is not truth valued, it expresses a property or relation using variables. Predicates can give rise to propositions in two ways. First, as we have already seen, by substantiating variables with names of objects. The second way is the use of a technique called quantification. Quantification introduces two symbols, $\exists$ the existential quantifier, and $\forall$ the universal quantifier.

For a unary predicate, P(x), with free variable x, both

$\exists x \bullet P(x)$ and $\forall x \bullet P(x)$

are propositions. The variable x is now said to be bound by the quantification, and can no longer be instantiated.

$\exists x \bullet P(x)$ asserts that there is at least one value, in the domain of interest, for which the predicate P(x) is true. For example, $\exists x \in N_1 \bullet$ is_even(x), states that there exist at least one element, in the set of natural numbers N_1, which is an even number.

$\forall x \bullet P(x)$ is interpreted as the proposition "x can be substantiated by the name of any objects, in the domain of interest, and the resulting proposition will be true." For instance, $\forall x \in N_1 \bullet$ is_positive(x), propose that it is true that any member from the set N_1 is a positive number.

In general, the logical operators used in propositional logic :
$\vee, \wedge, =>, \Leftrightarrow$ and $\neg$
are also used in predicate logic.

5.3.4 Cartesian products

Let A and B be two sets. The *Cartesian product set* A x B (read as "A cross B" or "A times B") is defined to be : $A \times B = \{<a,b> \mid a \in A, b \in B\}$.

The elements <a,b> of A x B are called ordered pairs. For <a,b> and <c,d> $\in$ A x B, <a,b> = <c,d> if and only if a=c, and b=d. In general, $A \times B \neq B \times A$.

This concept can be generalized to n sets :
$A_1 \times A_2 \times \ldots \times A_n = \{<a_1, a_2, \ldots, a_n> \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n\}$,
and $<a_1, a_2, \ldots a_n>$ is called an ordered n-tuple.
For ordered pairs of the form (a,b) with $a \in A$, $b \in B$; |A| denotes the number of members of the set A, and |B| the number of members of B. It is easy to see [Brady77] that : $|A \times B| = |A| * |B|$, where * is the arithmetic multiplication operator.

5.3.5 Relations

A *relation* is a subset of a product set. An n-ary relation is a subset of a product set of n sets. If n=2, the relation is called a *binary relation*. If R is a subset of A x B for some sets A and B, R is called a relation from A to B, and is denoted as $R : A \leftrightarrow B$.

Whenever $<a,b> \in R$, it can be denoted as a R b.

The set $C = \{a \in A \mid \text{for some } b \in B, <a,b> \in R\}$ is called the *domain* of R, and the set $D = \{b \in B \mid \text{for some } a \in A, <a,b> \in R\}$ is called the *range* of R. Consequently, $C \subseteq A$ and $D \subseteq B$. For a binary relation $R : A \leftrightarrow A$,

R is reflexive if	$a R a \;\; \forall \; a \in A$.
R is irreflexive if	$a \not{R} a \;\; \forall \; a \in A$.
R is symmetric if	$a R b \Rightarrow b R a \;\; \forall \; a,b \in A$
R is antisymmetric if	$(a R b \text{ and } b R a) \Rightarrow a=b \;\; \forall \; a,b \in A$
R is transitive if	$(a R b \text{ and } b R c) \Rightarrow a R c \;\; \forall \; a,b,c \in A$

5.3.6 Functions

A *total function* from a set A to a set B is an association or pairing of a member of B with each element of A. Several members of A may be paired with the same member of B, but no single member of A may be paired with more than one member of B. If f is such a total function, we write

$f : A \rightarrow B$,

and if $b \in B$, $f(a)$ is used to denote the unique member of B paired with a. A is called the domain of f, and $f(A) \subseteq B$ is called the range. The range of f is the set :

$f(A) = \{b \in B \mid b = f(a), a \in A\}$.

A total function is usually called a *function* or a *mapping*. When there is no necessity for all the members of A to be mapped to members of B (i.e. the function is defined on only a subset of A), f is called a *partial function*, and denoted as $f : A \nrightarrow B$.

A function $f : A \rightarrow B$ is called *onto* or *surjective* if $f(A)=B$; that is, for every element $b \in B$, there is at least one element $a \in A$ with $f(a)=b$.

For example, the function $f : \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x)=x3$ is onto; where $\mathbb{R}$ is the set of real numbers.

A function $f : A \rightarrow B$ is called *one to one* or *injective* if $a_1, a_2 \in A, f(a_1)=f(a_2)$ implies $a_1=a_2$; that is, every element in the range of f is the image of exactly one element from the domain.

5.3.7 Finite State Machine (FSM)

A finite-state machine (FSM) is an abstract model of control mechanism found within any deterministic input/output device (e.g. a digital computer). A FSM has only a finite number of internal states that have the capacity to remember certain information and behave in an expected manner for valid input data. The machine as a whole will recognize some input and produce an expected output. There are four equivalent methods of describing a FSM :

(1) as a labelled digraph

(2) as a matrix

(3) as a regular expression

(4) as an algebraic system

A formal definition of a finite-state machine [Alagar89] with semantics is
$M=<S, S_0, A, B, f, g>$ where

$S = \{S_0, S_1, S_2, ..., S_m\}$ is a finite set of states, and S_0 is the unique initial state.

$A = \{a_1, a_2, ..., a_n\}$ is a finite set of input symbols.

$B = \{b_1, b_2, ..., b_p\}$ is a finite set of output symbols.

f is a transition map $f : S \times A \rightarrow S$

where $f(S_i, a_j) = S_k$ means that if a_j is the input symbol encountered by the machine in state S_i, the machine transits to state S_k.

g defines the output, $g : S \times A \rightarrow B$

where $g(S_i, a_j) = b_k$ means that the machine produces the output b_k if the input symbol a_j is encountered at its state S_i.

5.4 WinSpec notation

In addition to symbols normally used in predicate calculus, a number of special notations have been introduced in the WinSpec language developed to abstract vital I/O details of user interactions.

5.4.1 Notations for display objects

The particular GUI being specified would have a finite set of display objects, denoted as gui_objects. The syntax for representing elements of the set gui_objects is :
obj_type "_" identifier ["#" instance_no] } , where :

• [#instance_no] indicates that the appearance of instance_no is optional; it is only necessary when there is more than one occurrence of objects of the same type and identifier. When used, instance_no is a natural number, and is preceded by a "#". (See examples on the next page.)

• _identifier is of the form _string or _'string'. When bound by quotation marks, string is the actual name (or label) that appears on the screen. For example, cBtn_'OK' indicates that the command button can be seen by the user as labelled 'OK'. When quotation marks are not used, string represents an internal name by which the display object is known to the user interface. This is the case where an object is displayed on the screen without a label.

• obj_type indicates the kind of display object being denoted. An example can be any member of the set of objects given below;
{wind , menu , mOpt, icon, diaB , cBtn , chkB , texB , cloB , sizB, zomB, tBar, sBar, hBar, vBar } , where

wind	indicates that the display object is a window ,
menu	indicates that the display object is a menu,
mOpt	indicates that the display object is a menu Option,
icon	indicates that the display object is an icon,
diaB	indicates that the display object is a dialogue Box,
cBtn	indicates that the display object is a command Button,
chkB	indicates that the display object is a check Box,
texB	indicates that the display object is a text Box,
cloB	indicates that the display object is a close Box for closing a window,
sizB	indicates that the display object is a size Box for resizing a window,
zomB	indicates that the display object is a zoom Box for zooming a window,
tBar	indicates that the object is a title Bar displaying the title of a window,
sBar	indicates that the object is a scroll Bar for scrolling text within a window,
hBar	indicates that the object is a horizontal scroll Bar,
vBar	indicates that the object is a vertical scroll Bar.

Some examples of elements of the gui_objects set are given below;

diaB_openFile	A dialogue box that is identified as the "openFile dialogue box". The identifier _openFile is not in quotation marks because that is not the actual wording in the title field of the dialogue box, or the dialogue box does not have a title field.
wind_'ReadMe'	The identifier _'ReadMe' represents the actual characters in the title of the window, since quotation marks are used round ReadMe.
wind_editor#2	This denotes the 2nd instance of the "editor window". The suffix #instance_no is used only when there is more than one instance of the same object.

The set state_primitives is the finite set that contains all the state primitives for describing visual states of display objects. For example, the following may be the state_primitives set for a certain user interface :

state_primitives = { is_visible , is_not_visible , is_hiLit , is_not_hiLit , has_kb_focus, is_modal, is_disabled, is_at_front, is_next_behind, is_inside, rect, text }

Elements of the state_primitives set (e.g. is_hiLit), are useful for describing elements of the gui_objects set (e.g. wind_folder#3), in statements such as is_hiLit(wind_folder#3). This gives rise to visual state predicates. A formal definition of visual state predicates is given in section 5.6.

5.4.2 Notations for visual state predicates (primitives describing display objects) :

is_visible(wind_x)	A predicate stating that wind_x is visible on screen (see WinSTD for visual appearance).
is_not_visible(wind_x)	A predicate stating that wind_x is not visible.
is_hiLit(cBtn_'OK')	A predicate stating that the border of command button 'OK' is highlighted (thicken).
is_not_hiLit(cBtn_'OK')	A predicate stating that the command button 'OK' is not highlighted.

is_at_front(wind_x) — A predicate stating that window wind_x is the front window.

is_next_behind(wind_y, wind_x) — A predicate stating the stacking order of two windows, wind_y is behind wind_x .

is_inside(mp?, icon_x) — A predicate stating that the hot spot of the mouse pointer lies within the borders of icon_x. (This primitive is useful for specifying interactions such as dragging a file icon onto the trash-can.)

has_kb_focus(texB_x) — A predicate stating that texB_x has keyboard focus (i.e. ready for keyboard inputs).

is_disabled(mOpt_'Save') — A predicate stating that menu option 'Save' is disabled (or dimmed).

is_enabled(mOpt_'Save') — A predicate stating that menu option 'Save' is not disabled.

is_modal(diaB_warn) — A predicate stating that dialogue box diaB_warn is a modal dialogue (i.e. blocks all inputs until dialogue is cleared).

rect(wind_'Editor') = (y1, x1, y2, x2) — A predicate stating that the window titled 'Editor' is bounded by the rectangle of coordinates (x1,y1) and (x2,y2) representing the positions of the top-left and bottom-right corners.

Loc(mp?) = (x,y) — A predicate stating that the location of the mouse pointer is at (x,y).

text(texB_'Username') = "demo" — A predicate stating that the text (character string) in text box titled 'Username' is "demo".

The semantics of visual state predicates are given informally in the above list. The development of formal semantics, if necessary, will have to be based on the formal specification of display objects (such as points, lines, regions and pictures). There are a number of published research work :

- Formal specification of a straight line [Marshall85].
- Formal specification of GKS output primitives [Duce86].

- Formal Specification of bitmap[4] images [Fiume89].
- Formal specification of a "Look Manager" [Narayana90].

The work in this area can be traced further back to [Mallgren82], who pioneered the formal specification of the graphics data type. [Mallgren82] gave definitions for four general graphics concepts, two of which (i.e. region and picture) are useful here. A region corresponds to an area in two dimensions, defined as a set of points in some universe U, typically the real plane for a two-dimensional area. A picture is modelled by a partial function P, whose domain (the points contained in the picture) is a subset of the set of points in the universe, and the range represents grey scales or colours. Some of the predicate operations discussed by [Mallgren82] are :

• coincident	picture X picture $\rightarrow$ boolean
• contains	picture X picture $\rightarrow$ boolean
• disjoint	picture X picture $\rightarrow$ boolean
• visible	picture X region $\rightarrow$ boolean
• bounded	picture X region $\rightarrow$ boolean

These will be useful for constructing formal semantics of constructs in WinSpec. The level of details about, for example, points and lines does not facilitate the testing of interaction functions. These details may be useful to research areas such as validating screen images by bitmap comparison (sometimes called visual verification), which is outside the scope of this thesis.

[4] A bitmap is a collection of picture elements (pixels). Computer graphics are often divided into two categories : raster graphics and vector graphics. They refer to the graphics output devices and the way that application programs draw graphics objects on these devices. Raster output devices (e.g. video displays, dot-matrix printers, laser printers) display images that are made up of dots called pixels (picture elements). Vector output devices, such as plotters, display images made up of lines and filled areas. Graphics programming interfaces of most window systems (e.g. Macintosh QuickDraw, OS/2 GPI) are basically vector graphics systems. They provide vector drawing commands in terms of lines and filled areas. However, these drawing commands are translated by the device driver into the appropriate format for the particular device, vectors or pixels.

5.4.3 Notations for GUI inputs :

kb? denotes keyboard inputs.

kb? can be viewed as a character string variable.

For example kb?= "abcdef" is a logical statement (a predicate) stating that the content of the keyboard input buffer is the string "abcdef". There are 4 special cases :

- kb?=<cr> is a carriage-return input.
- kb?=<tab> is an input of the tab key.
- kb?=<del> is an input of the delete key.
- kb?=<cmd-?> is the input of a single key together with the command key (often used instead of menu options, see chapter 8 for details).

mb? denotes mouse button inputs.

For example mb?=<click> is a predicate stating that there is a click at the mouse button input device. Four different kinds of mouse button inputs are usually distinguished in user interfaces.

- mb?=<down> means mouse button is pressed down.
- mb?=<up> means mouse button is released.
- mb?=<click> means mouse button is pressed down and then released quickly. A <click> is a <up> followed by a <down> within a certain pre-defined time limit.
- mb?=<dClick> means mouse button is given a double click input. A <dClick> is a <click> followed by another <click> within a certain pre-defined time limit.

mp? denotes mouse pointer inputs.

For examples :

is_inside(mp?, icon_x) is a predicate stating that (the hot spot part of) the mouse pointer is inside icon_x.

Loc(mp?) = (x,y) is a predicate stating that the location of the hot spot (i.e. the arrow tip) of the mouse pointer is at (x,y).

5.4.4 Notations for messages

In addition to the visible output in display objects, the GUI communicates with the main body of the application by sending (and receiving) messages. GUI validation is to check these application messages together with display objects (shown in WinSTDs). An example of a message predicate is :

app_msg_sent = "abc...123"

This is a predicate that will become true when a message "abc...123" has been sent. Another example is :

app_msg_sent = text(texB_user)
stating that the text in texB_user has been sent as a message to the application. Another example :

app_msg_recv = "logon failure"
is a predicate stating that a message of "logon failure" has been received from the application.

5.4.5 Notations for logical operators

or logical or

and logical and

Tand Temporal logical and. It is a "non-commutative and" for showing a time sequence, requiring that the predicate on its left must be satisfied before the predicate on its right. For example :

is_inside(mp?, cmdB_'OK') Tand mb?=<click>
will become true when the mouse pointer is first located inside cmdB_'OK' "and then" there is a mouse button click input.

"predicate-A Tand predicate-B" has the semantic :
First predicate-A becomes true, then predicate-B becomes true whilst no other input predicates have become true in the meantime.

5.4.6 Notations for comments

! Comments can be inserted after "!" .
For example :
text(texB_1) = "user1" ! Text box texB_1 now holds string "user1".

5.5 Specification of interaction functions

In WinSpec, an interaction function represents a basic step of user interaction with the user interface. Each interaction function of a GUI is given a unique name. The specification of an interaction function is the basic building block of a WinSpec specification for a GUI. The following is an example of the specification of an interaction function F1. (A complete specification, including declaration of all display objects and states, is given in section 5.7.)

<u>Specification for function F1 :</u>

From_state :	Start ! "Start" is the initial state
F_state_predicate :	is_visible(icon_logon)
Inputs :	is_inside(mp?, icon_logon) Tand mb?=<dClick>
To_state :	PostF1 ! It transits to state "PostF1".
T_state_predicate :	is_visible(diaB_logon) and has_kb_focus(texB_user) ! Logon dialogue box appears, and username entry field has input ! focus, <dClick> is the same as <doubleClick>
Output_msg :	none

The specification of an interaction function consists of 7 different parts. The first line gives the name of the interaction function being specified. The "From_state" clause specifies the state of the user interface before the execution of the function. The "F_state_predicate" describes the vital properties of the state (i.e. the "From_state") of the GUI. This predicate, on the visual state of display object(s), must be true, for the execution of the function to take place. The "Inputs" clause specifies the required inputs to invoke this interaction function. The "To_state" clause specifies the state of the user interface following the execution of the function. Finally, the "T_state_predicate" specifies the vital properties of the "To_state" of the GUI, after the execution of the function. The "T_state_predicate" becomes true as a consequence of the function executed. No output messages are specified in F1, and comments can be added following "!".

The above is an example of part of a WinSpec specification, which can be represented as part of a State Transition Diagram (STD).

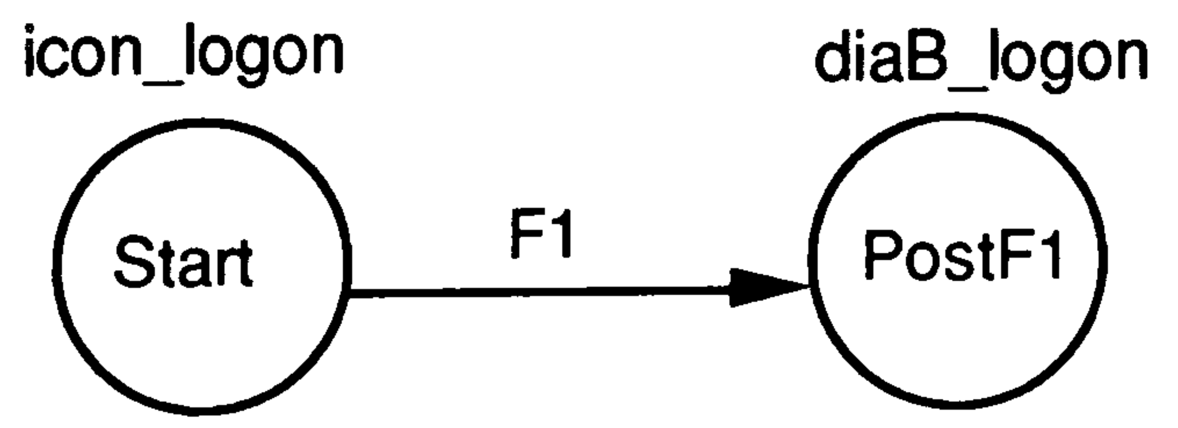


Figure 5.2 Part of a STD showing function F1

The associated WinSTD has already been given in Figure 5.1 (section 5.1), covering F1 and other functions of the Logon interface. The complete WinSpec specification for the Logon interface is presented in section 5.7, following section 5.6 which attempts to define WinSpec formally.

5.6 A Formal Definition of WinSpec

In essence, the specification approach is to model user interactions as transitions in a state diagram. Following the definition of a finite state machine, as given in section 5.3.7, a WinSpec specification W is formally defined as :

W = $<S, S_0, A, B, f, O, P, G, h>$ where

$S = \{S_0, S_1, S_2, \ldots, S_m\}$ is a finite set of states, and S_0 is the unique initial state.
$A = \{a_1, a_2, \ldots, a_n\}$ is a finite set of inputs.
$B = \{b_1, b_2, \ldots, b_p\}$ is a finite set of outputs.

f is a transition map (i.e. a function as described in section 5.3)

$f: S \times A \rightarrow S \times B$

where $f(S_i, a_j) = (S_k, b_l)$ means that if a_j is the input encountered by the machine in state S_i, the machine generates the output b_l and transits to state S_k.

$O = \{O_1, O_2, \ldots, O_q\}$ is a finite set of display objects.
$P = \{P_1, P_2, \ldots, P_r\}$ is a finite set of visual primitives to describe display objects.

G is a relation, $G : S \leftrightarrow P \times O$
where $S_t \; G <P_u,O_v>$ means that if the machine is in state S_t, object O_v is described by the primitive P_u. (Since G is a relation, it is also possible that $S_t \; G <P_u,O_w>$, where $w \neq v$. See section 5.3 for the definition of a relation.)

Finally, h is a function, $h : P \times O \rightarrow$ Boolean

where $h(P_u, O_v) =$ True or $h(P_u, O_v) =$ False, but not both.
For $h(P_u, O_v) =$ True, it is also written as $P_u(O_v) =$ True, or simply $P_u(O_v)$.
The form $P_u(O_v)$ is called a visual state predicate in this thesis. In conjunction with the definition of relation G above, the visual state predicate $P_u(O_v)$ is used to describe the state S_t.

"O", the set of display objects, is necessarily a finite set for the GUI being specified (as described in section 5.4 earlier). "P" is the set of visual state primitives for display objects (described in 5.4.2). When state primitives are used in a WinSpec to describe display objects, visual state predicates (in the form of $P_u(O_v)$, e.g. is_visible(icon_'Logon')) are formed. "S" is the finite set of states of the GUI captured in the specification. These states are represented as nodes in state diagrams. Each state can be described by visual state predicates. "A" is the set of inputs specified for the GUI, including keyboard, mouse and message inputs. "B" is the set of outputs, which is used in a WinSpec to specify messages that the GUI can send to the underlying application. The visual output of display objects is captured in the states and visual state predicates, and not included in the output set B.

5.7 An example of specification : the logon user interface

The logon user interface interacts with the user by displaying a dialogue box prompting the input of a username and password. The visual appearance of display objects and control flow of interaction functions can be seen in the WinSTD in Figure 5.1. Text boxes are provided for the username and password entry. If the user inputs a mouse click at the "OK" command button or types the carriage-return key, the existing contents of the username and password text boxes will be checked for authorization. If the username and password are valid, a terminal window will be displayed for user access. Otherwise a "Logon failure" dialogue box will appear to inform the user, which must be cleared by selecting the "reset" command button. The WinSpec specification for the logon user interface is given in the following listing.

```
WinSpec_begin logon

logon_objects = {icon_logon, diaB_logon, cBtn_'OK', texB_user, texB_pass,
                 diaB_'Logon Failure', cBtn_'Reset', wind_term, cloB_term}
logon_states = {Start, postF1, postF2.2, postF4, postF5, postF7}
state_primitives = {is_visible, is_not_visible, is_inside, has_kb_focus, text, is_hiLit}


Specification for function F1 :
From_state :          Start                            ! "Start" is the initial state
F_state_predicate :   is_visible(icon_logon)
Inputs :              is_inside(mp?, icon_logon)  Tand  mb?=<dClick>
To_state :            PostF1                           ! It transits to state "PostF1".
T_state_predicate :   is_visible(diaB_logon)  and   has_kb_focus(texB_user)
                      ! Logon dialogue box appears, and username entry field has
                      ! input focus, <dClick> is the same as <doubleClick>
Output_msg :          none

Specification for function F2 :
From_state :          PostF1
F_state predicate :   is_visible(diaB_logon)  and   has_kb_focus(texB_user)
Inputs :              kb? ∉ {<cr>, <tab>, {} }    ! Keyboard input, except
To_state :            PostF1                      ! <cr>, <tab> or empty i/p.
T_state_predicate :   text(texB_user)=kb?         ! Content of text box reflects key i/p.
Output_msg :          none

Specification for function F2.1 :
From_state :          PostF1
F_state_predicate :   is_visible(diaB_logon)  and   has_kb_focus(texB_user)
Inputs :              kb?= <cr>                   ! A <cr> keyboard input.
To_state :            PostF4
T_state_predicate :   is_hiLit(cBtn_'OK')
                      ! The OK command button (cBtn_'OK') is highlighted, and
                      ! a message is sent.
Output_msg            app_msg_sent= ("user=", text(texB_user), "pass=",
                            text(texB_pass) )
```

Specification for function F2.2 :

From_state :	PostF1	
F_state_predicate :	is_visible(diaB_logon) and has_kb_focus(texB_user)	
Inputs :	kb?=<tab>	! A <tab> keyboard input.
To_state :	PostF2.2	
T_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Output_msg :	none	

Specification for function F2.3 :

From_state :	PostF1	
F_state_predicate :	is_visible(diaB_logon) and has_kb_focus(texB_user)	
Inputs :	is_inside(mp?, diaB_pass) Tand mb?=<click>	
To_state :	PostF2.2	
T_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Output_msg :	none	

Specification for function F3:

From_state :	PostF2.2	
F_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Inputs :	kb? ∉ {<cr>, <tab>, {} }	! Keyboard input, except
To_state :	PostF2.2	! <cr>, <tab> or empty i/p.
T_state_predicate :	text(texB_pass)=kb? and is_not_visible(text(texB_pass)) ! Password entry has no echo	
Output_msg :	none	

Specification for function F3.1 :

From_state :	PostF2.2	
F_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Inputs :	kb?= <cr>	
To_state :	PostF4	
T_state_predicate :	is_hiLit(cBtn_'OK') ! The OK command button is highlighted, and then logon dialogue ! box (diaB_logon) disappears, and a message is sent.	
Output_msg	app_msg_sent= ("user=", text(texB_user), "pass=", text(texB_pass))	

Specification for function F3.2 :

From_state :	PostF2.2	
F_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Inputs :	kb?=<tab>	
To_state :	PostF1	
T_state_predicate :	has_kb_focus(texB_user)	! Input focus back to texB_user
Output_msg :	none	

Specification for function F3.3 :

From_state :	PostF2.2	
F_state_predicate :	has_kb_focus(texB_pass)	! Password field now has input focus
Inputs :	is_inside(mp?, texB_user) Tand mb?=<click>	
To_state :	PostF1	
T_state_predicate :	has_kb_focus(texB_user)	! Input focus back to texB_user
Output_msg :	none	

Specification for function F4 :

From_state : PostF1
F_state_predicate : is_visible(diaB_logon)
Inputs : is_inside(mp?, cBtn_'OK') Tand mb?=<click>
To_state : PostF4
T_state_predicate : is_hiLit(cBtn_'OK')
! The OK command button is highlighted, and a message is sent.
Output_msg : app_msg_sent= (“user=”, text(texB_user), “pass=”, text(texB_pass))

Specification for function F5 :

From_state : PostF4
F_state_predicate : is_hiLit(cBtn_'OK')
Inputs : app_msg_recv= "Logon failure"
To_state : PostF5
T_state_predicate : is_not_visible(diaB_logon) and
is_visible(diaB_'Logon Failure') and is_hiLit(cBtn_'Reset')
! Logon failure dialogue box appears and reset command button
! is highlighted.
Output_msg : none

Specification for function F6 :

From_state : PostF5
F_state_predicate : is_visible(diaB_'Logon Failure') and is_hiLit(cBtn_'Reset')
Inputs : kb?=<cr> or
(is_inside(mp?, cBtn_'Reset') Tand mb?=<click>)
To_state : Start
T_state_predicate : is_not_visible(diaB_'Logon Failure')
! a <cr> input or a click on the reset button (cBtn_'Reset')
! clears logon failure dialogue box (diaB_'Logon Failure')
Output_msg : none

Specification for function F7 :

From_state : PostF4
F_state_predicate : is_hiLit(cBtn_'OK')
Inputs : app_msg_recv= "Logon ok"
To_state : PostF7
T_state_predicate : is_not_visible(diaB_logon) and
is_visible(wind_term) and has_kb_focus(wind_term)
! Logon success, terminal window (wind_term) appears and
! has input focus.
Output_msg : none

Specification for function F8 :

From_state : PostF7
F_state_predicate : is_visible(wind_term) and has_kb_focus(wind_term)
Inputs : is_inside(mp?, cloB_term) Tand mb?=<click>
! A click on the close box makes the terminal window to disappear.
To_state : Start
T_state_predicate : is_not_visible(wind_term)
Output_msg : none

WinSpec_end

The following STD gives a graphical representation of the above WinSpec specification for Logon. There are two main differences between this STD and the WinSTD given in Figure 5.1 earlier. The first difference is that a WinSTD uses display objects instead of circles to represent nodes. Secondly, a WinSTD does not show all the functions specified in the corresponding WinSpec, for reasons of space and clarity.

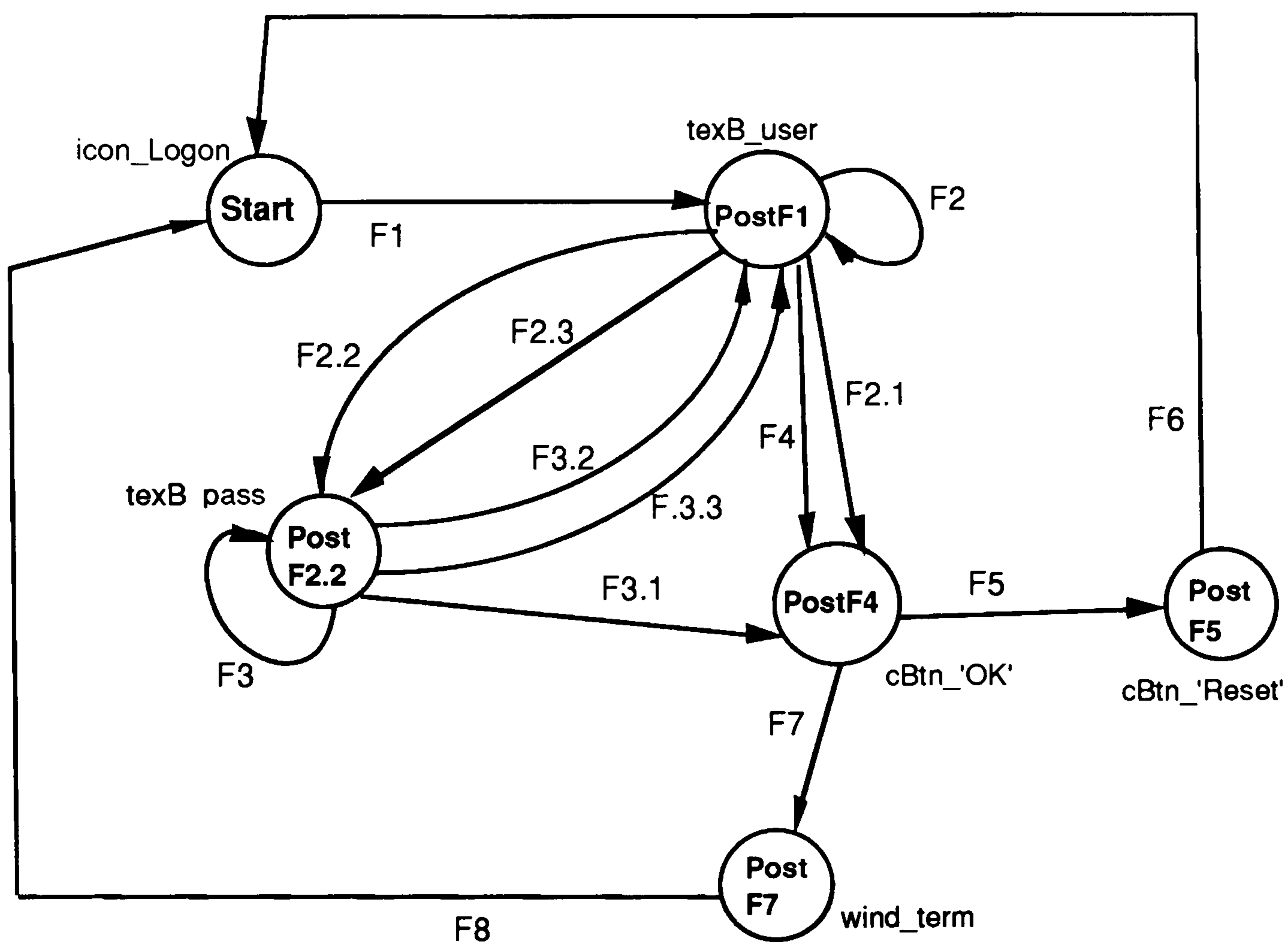


Figure 5.3 An STD for the Logon interface

5.8 Review, assumptions and summary

The above specification precisely and unambiguously states the functions of the logon user interface for the purposes of test input generation and output checking. More statements can be added to the state predicates, if desired, to expose other properties of the user interface. This illustrates the power of abstraction in a specification approach, by revealing important features and hiding unnecessary details. For instance, in the specification of the function F2 for entering text to the username field (texB_user), no definite text length restrictions are stated. This is to keep the specification abstract and not to dictate implementation issues unnecessarily. Moreover, F2 does not explicitly indicate any text editing capabilities that texB_user might have. This exemplifies the use of functional decomposition, where a simple text entry function within a text box can be expanded into detailed text editing functions, similar to those of a text editor specified in chapter 8. One reason for not presenting such functional decomposition in the specification of Logon is that these editing functions are, strictly speaking, part of the underlying window system library, not the Logon interface. There are a number of "common sense" conditions that can be, but are not, included in the state predicates of functions in order to keep the specification concise and readable. The following are some examples.

• When the system is still busy handling the previous input, a different shape for the mouse pointer (e.g. of the shape of a watch or clock, denoted as icon_wait) is displayed, instead of the arrow shape normally used for the mouse pointer. This signals users to wait, and inputs are blocked (queued or discarded). This is a way of serialising inputs in order to regulate "type ahead" and "mouse ahead". It is necessary because the second input might be directed at some objects that are part of the output responding to the first input, which is still not ready. There is an assumption in all the above F_state_predicates that the system is ready to receive input, which can be explicitly specified to expose this feature of a real time system: is_not_visible(icon_wait).

• There is a hidden, unspecified means of determining that a new input (or message) has just been received. In a real time system, an input predicate (e.g. kb?= "abc...123") always refers to the new input that has just arrived. This can be explicitly expressed, if necessary, by additional predicates (e.g. kb?={ }) to indicate that input buffers are cleared after the last input.

• Another feature which is hidden in the above specification is that of concurrency. There might be other user interfaces running in a time-sharing manner with the logon interface in a single processor workstation environment. (Multi-processor and parallel systems are outside the scope of this thesis.) It is not possible to specify effects of other user interfaces that may become concurrent with the logon interface. For example, it is not possible to specify the changes made, by another user interface, to the content of the global clipboard. (The global clipboard allows text transfer amongst concurrent user interfaces.)

• Another hidden assumption in WinSpec is that predicates in the T_state_predicate will only state display object changes as consequences of a specified function. Other display objects are assumed to remain unchanged by the execution of this function. Any unspecified visual changes on the screen (e.g. time clock) are considered irrelevant. This assumption helps the tester to focus on the testing of one GUI, whilst there may be other GUIs running concurrently.

• Although the mouse input device (pointer and button) is used exclusively in specifications in this thesis, it does not imply other devices cannot be used in graphical user interfaces. There are a number of existing devices, such as thumb wheels, crosshair, joysticks, tablet styluses, physical buttons and dials. There are also mouse devices that support more than one mouse button. Despite the physical differences in these devices, similar logical inputs can be achieved. The two main classes of logical inputs, locator (a screen location pointer) and button (entry of a single bit information), can be achieved by a number of devices [Hopgood86]. WinSpec specifications are designed for logical inputs and would be suitable for use with devices other than mouse pointers and buttons.

In this chapter, an original specification approach for graphical user interfaces has been presented, in terms of a state diagram called WinSTD and a language called WinSpec. A WinSpec specification of a user interface is formed by defining interaction functions with precise statements about the required inputs and expected outputs. An example specification of a logon interface has been given. The specification method developed will be used as the basis for test case generation, as revealed in the following chapters.

Chapter 6

Graph theory, postman problem and test sequences

In Chapter 5, specification for the Logon interface was developed in terms of WinSpec notations and state transition diagrams (STDs). In order to derive test sequences from these specifications, it is necessary to study some graph theoretic algorithms applicable to STDs. In this chapter, some algorithms and results of graph theory are used without extensive formal derivation or justifications, since the study of graph theory is secondary to the main theme of testing user interfaces in this thesis. Furthermore, graph theory terms are only defined here if they are useful to the application of test sequence generation. Graph theoretic terms used in this chapter are not included in the glossary in appendix A.

The first paper on graphs was written by the Swiss mathematician Leonhard Euler (1707-1783) and was published in 1736 by the Academy of Science in St.Petersburg. Euler's study of graphs was motivated by the so-called Konisberg bridge problem [Minieka78].

6.1 Definition of terms used in graph theory

Formally, a *graph* G=(V,E) consists of a finite set V of *vertices* (nodes or points) and a set E of *edges* (or arcs) joining pairs of vertices. If $V=\{v_1, v_2, ..., v_n\}$ and $E=\{e_1, e_2, ..., e_m\}$, then each e_k is an unordered pair (v_i, v_j). The vertices (v_i, v_j) are the end vertices of e_k and are *adjacent* to each other. We also say that e_k is *incident* with v_i and v_j. The total number of edges incident to a vertex is called the degree of vertex v_i and is denoted by $d(v_i)$. If all vertices in graph G have even degrees (i.e. $d(v_i)$ is even for all v_i), then graph G is said to be *even* .

Let v_0 and v_m be the two vertices of a graph G. The sequence $(v_0, v_1, ..., v_m)$ is called a *walk* joining v_0 and v_m if each v_j is adjacent to v_{j-1}, $1 \leq j \leq m$. Two vertices v_i and v_j in G are said to be *reachable* from each other if there is a walk joining v_i and v_j in G.

When the set E consists of ordered pairs $<v_i, v_j>$ of vertices, then $E \subseteq V \times V$, and G is a *digraph* (a directed graph). For vertices of a digraph, reachability is directional. In a digraph, the number of arcs directed into vertex v_i is called the inner degree of vertex v_i and is denoted by $d^{in}(v_i)$. The number of arcs directed away from vertex v_i is called the outer degree of vertex v_i and is denoted by $d^{out}(v_i)$. If $d^{in}(v_i) = d^{out}(v_i)$ for all vertices v_i in graph G, then G is called a *symmetric* graph. A graph $G_1=(V_1, E_1)$ is a *subgraph* of a graph G=(V, E) if $V_1 \subseteq V$ and $E_1 \subseteq E$.

6.2 The Euler tour problem

In 1736, Euler developed the concept of an Euler tour, in connection with the Konisberg bridge problem.The city of Konisberg (later called Kaliningrad) in East Prussia was built at the junction of two rivers and two islands (see Figure 6.1). In all, there were seven bridges connecting the islands to each other and to the rest of the city along the river banks.

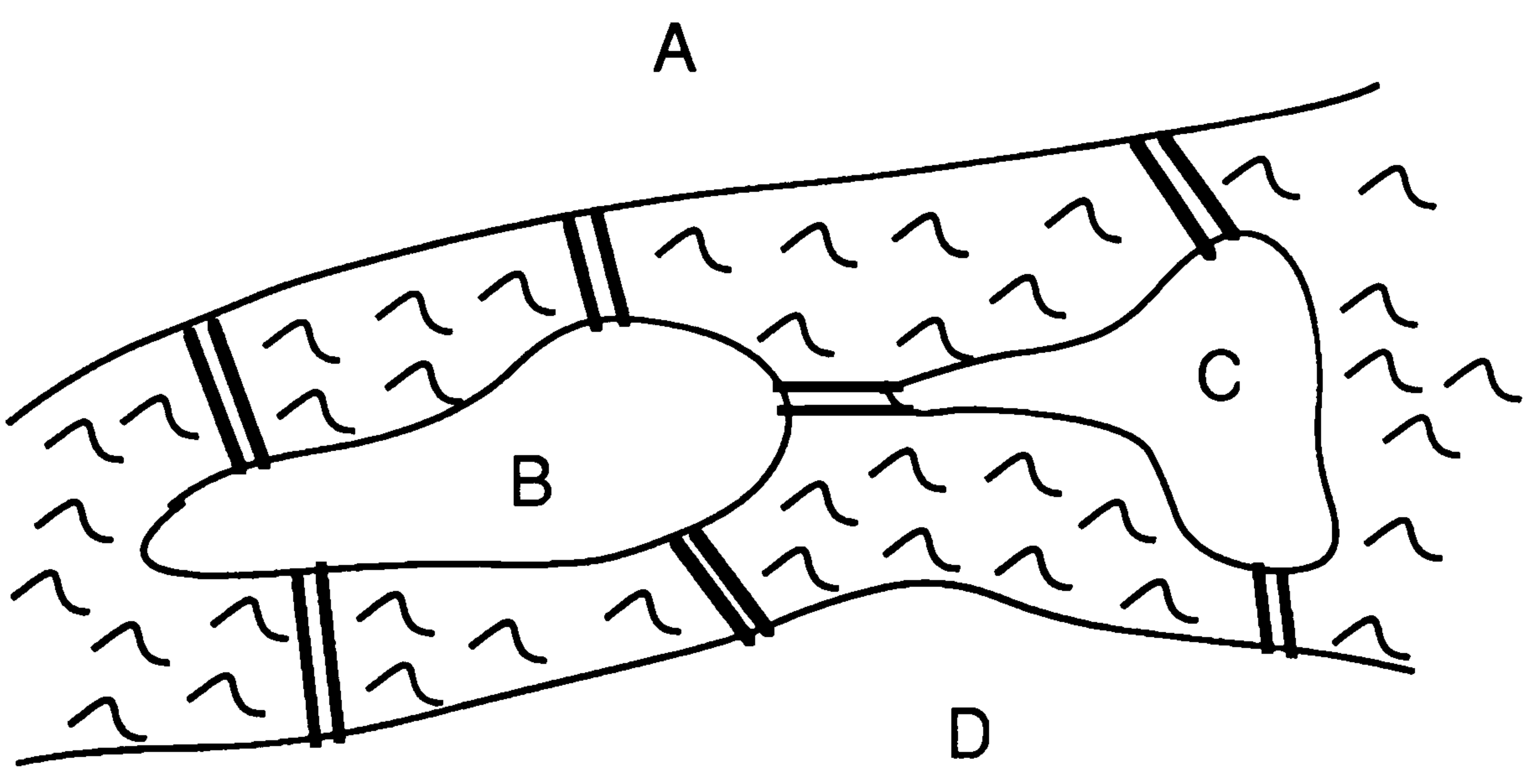


Figure 6.1 The Konisberg bridge problem

The Konisberg bridge problem was : could a Konigsberger start from his home and cross each bridge exactly once and return home? (Such a walk was later called an Euler tour in graph theory.) Euler proved that the answer was no. Avoiding the detailed mathematical proofs, a simple explanation is given here. The first step is to redraw the picture in Figure 6.1 as a graph in Figure 6.2.

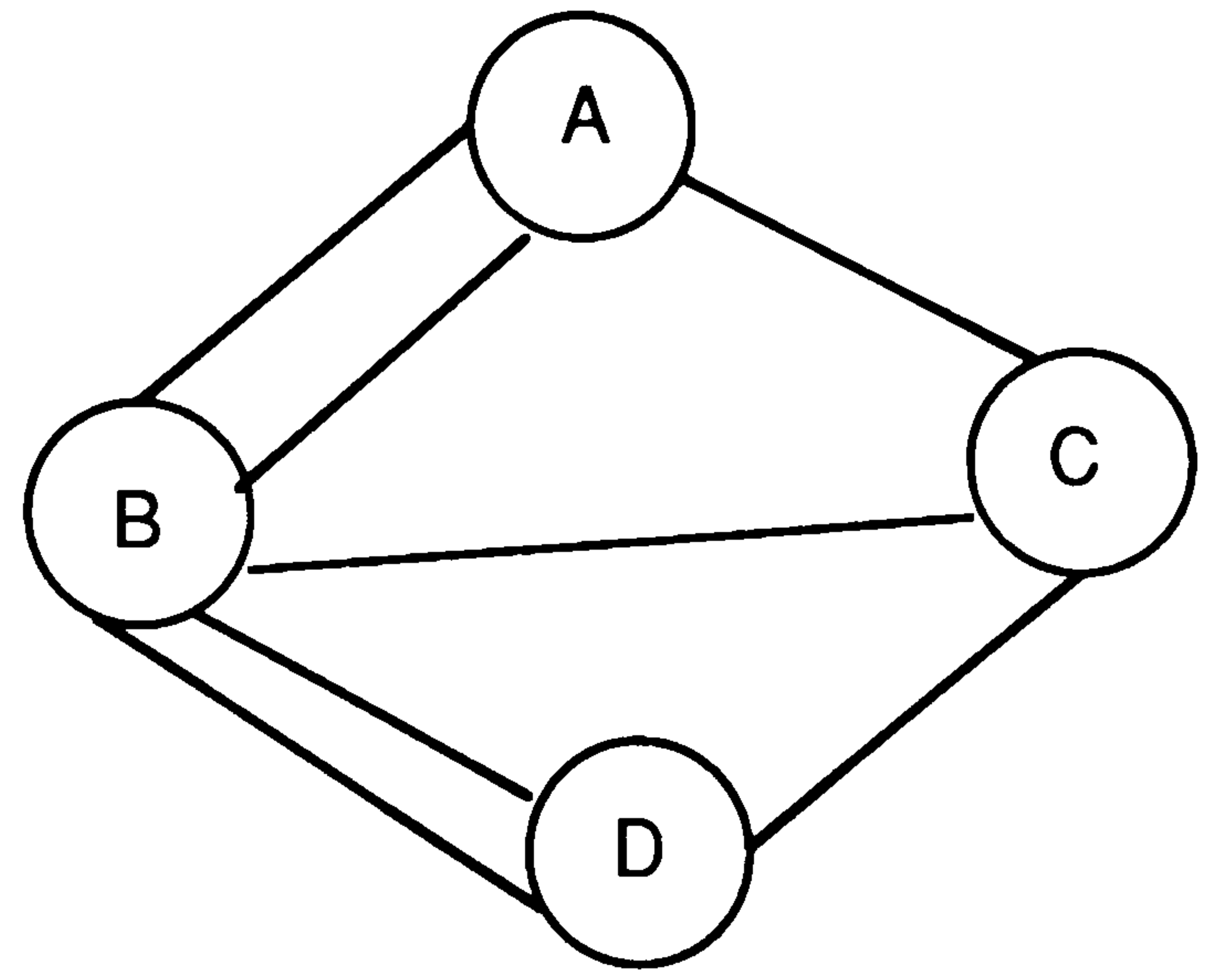


Figure 6.2 The Konisberg bridge problem represented as a graph

The Konisberg bridge problem is represented by a graph consisting of 4 vertices (or nodes) and 7 edges (or arcs). It is evident that a Konisberger who arrives at a node via one bridge must leave that node by a different bridge, in order that each bridge is crossed exactly (and not more than) once. This means that there must be an even number of edges (i.e. bridges) connected to each of the nodes. In graph theoretic terms, the graph must contain an even degree at all vertices. This requirement is not satisfied by the graph in Figure 6.2, as the vertices have an odd number of edges connected to them. Therefore, the answer to the Konisberg bridge problem is no.

A walk through a graph in which each edge is traversed exactly once is called an Euler tour. An Euler tour for a graph G exists only if G is an even undirected graph, or if G is a symmetric directed graph [Minieka78]. There are standard techniques for deriving the list of edges of an Euler tour from an even undirected or a directed symmetric graph.

6.3 The Postman tour

M.K.Kuan first developed the postman problem in 1962. Consider a graph G=(V,E), in which each edge represents a street in the postman's route and each vertex represents a junction between two or more streets. The postman problem is that of finding the shortest route by which the postman can traverse each edge at least once and return to his starting vertex. The first publication of this problem appeared in a Chinese journal [Kuan62] and is often referred to as the Rural Chinese Postman Tour (RCPT).

A postman tour allows repeated traverses of some edges, if necessary, so that the shortest route which covers all edges can be taken, thereby differing from an Euler tour.

In any postman route, the number of times that the postman enters a vertex equals the number of times that the postman leaves that vertex. Recall that an undirected graph G is even, if all vertices in graph G have an even degree (i.e. even number of edges connected to all vertices). For an undirected even graph G, an optimal solution to the postman problem is an Euler tour. The postman does not have to repeat visits to any edges.

For a directed graph, the number of arcs entering a vertex V_i is denoted as $d^{in}(V_i)$. The number of arcs leaving vertex V_i is denoted as $d^{out}(V_i)$. If G is symmetric (that is $d^{in}(V_i) = d^{out}(V_i)$ for all vertices v_i), then it is possible for the postman to perform his route without repeating any arcs. Thus, an Euler tour provides an optimal solution to a postman problem if that problem can be represented by a directed symmetric graph.

If the graph G representing the postman problem is not even or symmetric, it is first transformed into an even or symmetric graph G* by the addition of a minimum number of edges (which are replicas of some existing edges in G).

If V_i has more arcs leaving than entering it (that is $d^{in}(V_i) < d^{out}(V_i)$), the postman must repeat some of the arcs entering into V_i. In other words, the transformation from G to G* is effectively the process of finding the edges to be repeated, resulting in the minimum total route length for the postman.

Let $f(V_i,V_j)$ denote the number of times that the postman repeats arc (V_i,V_j) and $c(V_i,V_j)$ be the cost for traversing the arc. The postman wants to select non-negative values for $f(V_i,V_j)$ so as to minimize :

$$\sum c(V_i,V_j)\ f(V_i,V_j) \quad , \quad \text{so that for all vertices } V_i$$

$$d^{in}(V_i) + \sum_j f(V_j,V_i) = d^{out}(V_i) + \sum_j f(V_i,V_j)$$

According to [Minieka78], this minimization is a minimum cost flow problem. Because of space and time limits, a general introduction to the minimum cost flow problem is not described in this thesis. Instead, a worked example using the STD of the Logon interface is given in the following section. In essence, it is to balance the flow (or degrees) at vertices, at minimum cost. Vertices with more incoming arcs than outgoing arcs are called sources. Vertices with more outgoing arcs than incoming ones are called sinks. A supersource and a supersink are appended to the graph, connecting the supersource to all sources, and connecting the supersink to all sinks. The problem is solved by deriving a minimum repetition of edges (i.e. optimal value of $f(V_i,V_j)$) that satisfies all source and sink requirements. Consequently, the number of incoming and outgoing arcs are balanced at all vertices .

6.4 Test sequences for the Logon user interface

For the purpose of testing graphical user interfaces, the graphs involved are directed graphs, as the arcs of STDs are directed. Since the interaction functions of a GUI are represented in the arcs of the STD, a test coverage of the functions of a GUI requires coverage of all the arcs, and is a postman problem. The following outlines how a solution to the postman problem is used to select the optimal test sequences for GUIs.

The STD for the Logon user interface (Figure 5.3) is reproduced in Figure 6.3, with simplified names of vertices. In Figure 6.3, the names A, B, ..., E and F are used for the vertices. Since the graph G in Figure 6.3 is not symmetric, the minimum cost flow approach is used to transform G into a symmetric graph. The basic idea is to equalize the $d^{in}(v_i)$ and $d^{out}(v_i)$ for all vertices, by the minimum repetition of existing edges.

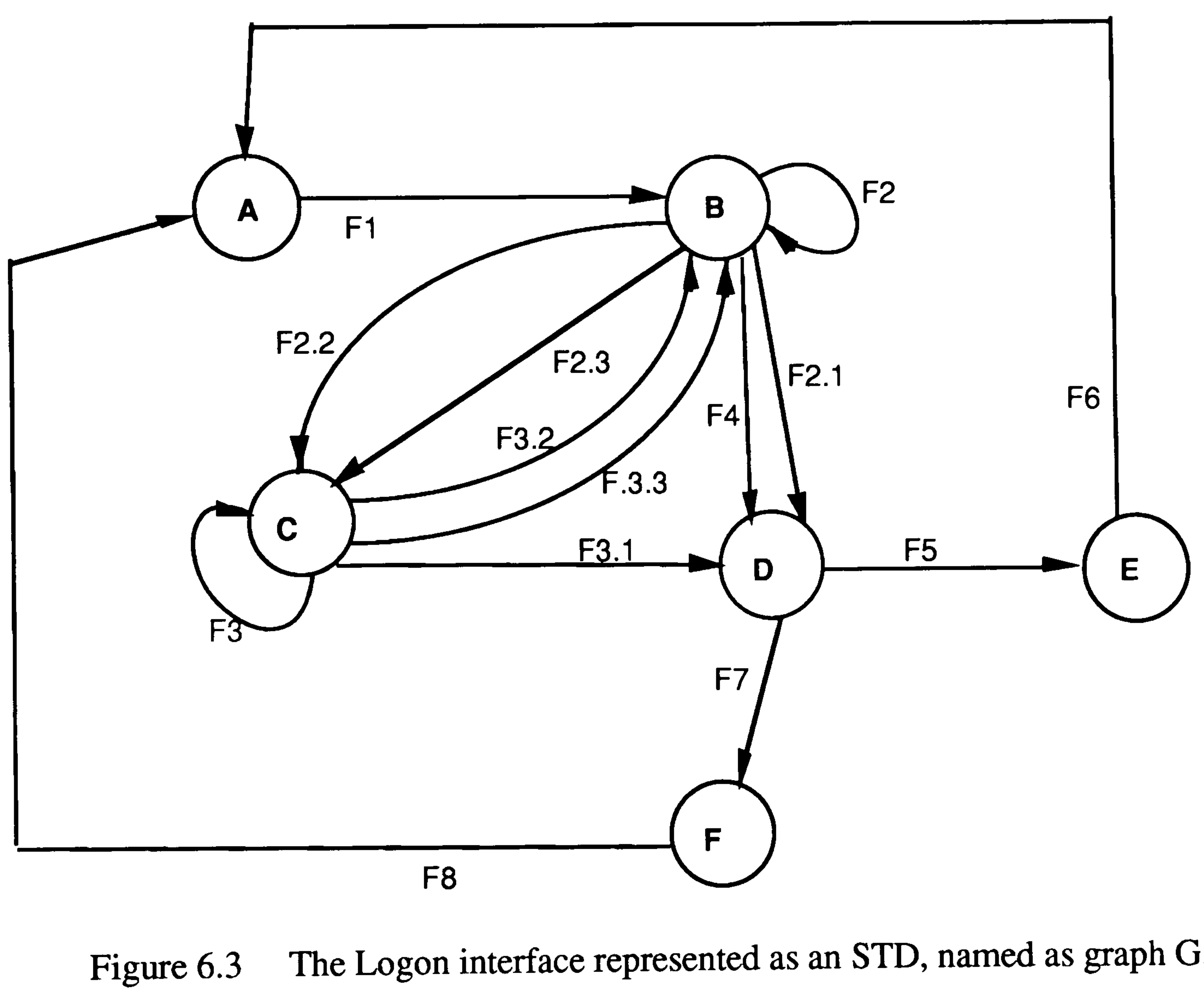


Figure 6.3 The Logon interface represented as an STD, named as graph G.

From Figure 6.3, the degrees of incoming and outgoing edges are calculated for all the vertices :

$d^{in}(A)=2$, $d^{out}(A)=1$; vertex A is a source with 2-1=1 unit supply.

$d^{in}(B)=4$, $d^{out}(B)=5$; vertex B is a sink with 5-4=1 unit demand.

$d^{in}(C)=3$, $d^{out}(C)=4$; vertex C is a sink with 4-3=1 unit demand.

$d^{in}(D)=3$, $d^{out}(D)=2$; vertex D is a source with 3-2=1 unit supply.

$d^{in}(E)=1$, $d^{out}(E)=1$; vertex E is intermediate vertex.

$d^{in}(F)=1$, $d^{out}(F)=1$; vertex F is intermediate vertex.

Having identified the sources and sinks of the graph G in Figure 6.3, it is necessary to decide which edges are to be replicated to satisfy the flow requirement. The following steps are taken :

- Create a supersource S and join S to source vertices A and D by arcs (S,A) and (S,D), each of 1 unit. (See Figure 6.4.)
- Create a supersink T and join T to sink vertices B and C by arcs (B,T) and (C,T), each of 1 unit.

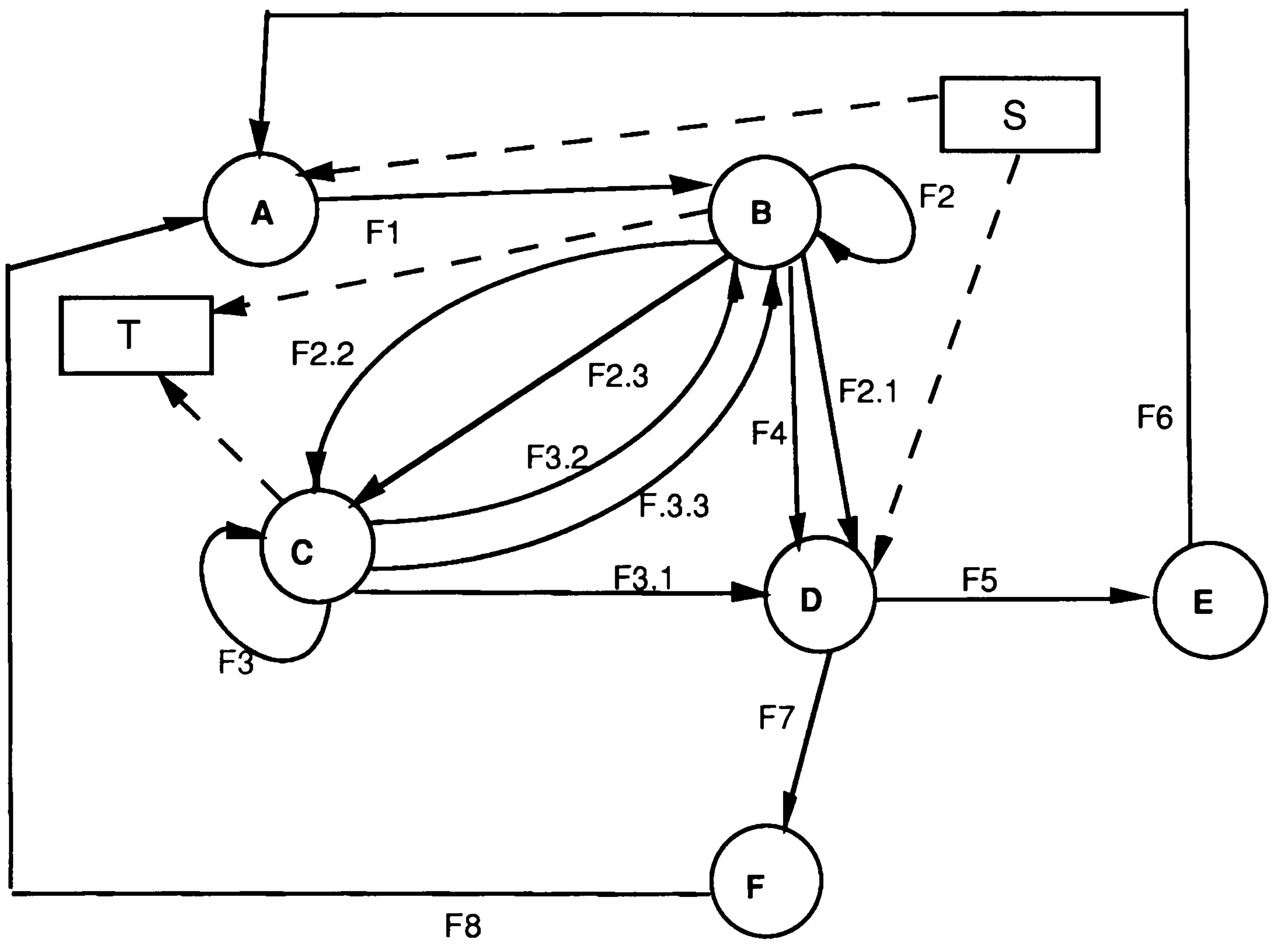


Figure 6.4 An illustration of the minimum cost flow approach

In order to balance the flow requirements, at most two flow units can be sent from S to T. One unit must leave S by way of vertex A, and another unit must leave S by way of vertex D. One unit must arrive at T by way of vertex B, and another unit must arrive at T by way of vertex C. The paths taken by these flow units as they travel from S to T correspond to the arcs which the postman must repeat.

From close inspection of Figure 6.4, it is obvious that one flow unit occurs along the path (S,A), (A, B) and (B, T). This is clearly the shortest path from S to T, via vertices A and B. This means the postman has to repeat the edge (A,B,F1) once.

Another flow unit is along the path (S, D), (D, E), (E, A), (A, B), (B, C) and (C, T). It is obvious, by inspection, that this is the shortest path from S to T, via vertices D and C. (It can be seen in Figure 6.4 that there is an alternative path from S to T, by way of vertices in the order S, D, F, A, B, C and T. This is of an equal length with the path chosen above. Mathematical proofs for these shortest paths are not given in this thesis, in order not to diverge too much from the main theme of testing.)

The second flow unit requires the postman to repeat each of the arcs joining vertices (D, E), (E, A), (A, B) and (B, C) once. There are two different edges which connect vertex B to vertex C. A replica of the edge (B,C,F2.2) is used here. (Alternatively, the edge (B,C,F2.3) can be used instead. This makes no difference to the testing process, as the unit cost for testing an edge (i.e. an interaction function) is assumed to be the same for all edges.)

In combining the requirements of the two flow units, the edges to be added are :

2 replicas of the edge (A,B,F1)
1 replica of (B,C,F2.2)
1 replica of (D,E,F5) and
1 replica of (E,A,F6).

These additional edges are augmented to G, shown as dotted lines, to form the symmetric graph G* as shown in Figure 6.5.

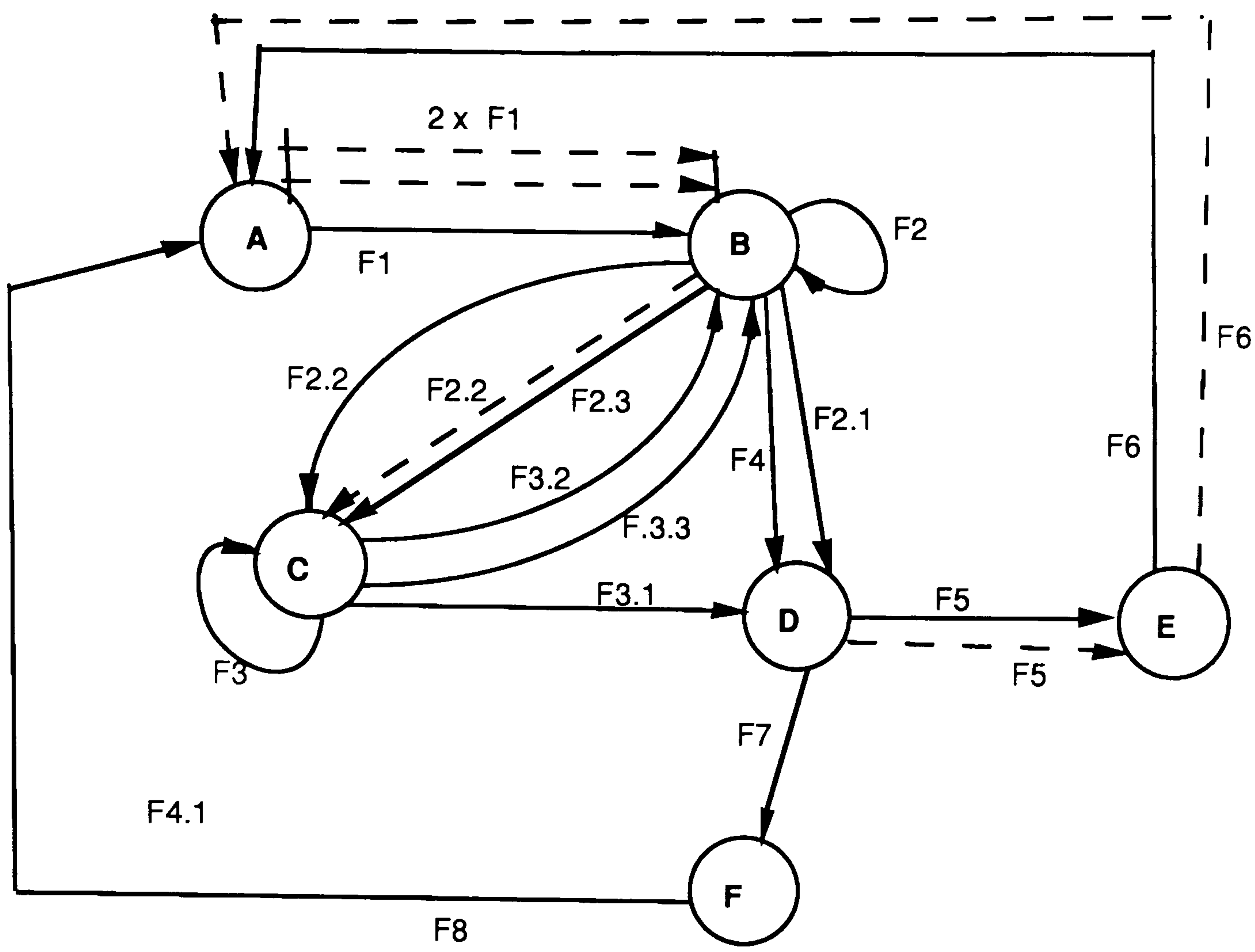


Figure 6.5 G* , a symmetric graph.

The graph G in Figure 6.3 has now been transformed into the graph G* in Figure 6.5. The dotted edges in Figure 6.5 represent replicas of existing edges, making G* symmetric. Since G* is a symmetric graph, it is possible to have an Euler tour of G*. An Euler tour of graph G* will provide an optimal postman tour for graph G.

A technique called splicing, useful for developing an Euler tour for an even undirected or a symmetric directed graph, is described as follows. Beginning at the starting vertex A, traverse the edges along their directions without reusing any edge until one returns to vertex A. This traces out a circuit Ct1. Next, starting at any unused edge, trace out another circuit, Ct2, using only unused edges. Repeat this procedure until all edges have been used. Lastly, splice together all the circuits into one large circuit CT. Circuit CT contains each edge exactly once and constitutes an optimal solution to the postman problem. This technique is demonstrated in the following paragraphs.

Applying the splicing technique, a circuit is recognized by following unused edges in Figure 6.5, starting from A and returning to A :
Ct1 : (A, B, F1), (B, C, F2.2), (C, D, F3.1), (D, F, F7), (F, A, F8)

Repeat the process of forming circuits, starting from any unused edge, using unused edges only. (A circuit is a walk which starts from a vertex and returns to the same vertex, without traversing any edges more than once.) The following circuits can be seen as walks starting and finishing at vertex B :

Ct2 : (B, B, F2)
Ct3 : replica of (B, C, F2.2), (C, B, F3.2)
Ct4 : (B, C, F2.3), (C, B, F3.3)

Again referring to Figure 6.5, the following circuit starts and finishes at vertex C :
Ct5 : (C, C, F3)

Remember that the augmented edges (dotted lines) are edges in their own right and have to be covered as well. So far, the remaining (i.e. unused) edges in G* are :
2 replica copies of (A,B,F1), (B,D,F2.1), (B,D,F4), (D,E,F5), replica of (D,E,F5), (E,A,F6) and a replica of (E,A,F6). These edges form two circuits. One circuit is :

Ct6 : replica of (A,B,F1), (B, D, F4), (D, E, F5), (E, A, F6).

The other circuit consists of :
Ct7 : replica of (A,B,F1), (B, D, F2.1), replica of (D,E,F5) and replica of (E,A,F6).

Now it can be verified that all edges in Figure 6.5 are covered exactly once in the circuits Ct1 to Ct7. Note that each of these circuits, when treated individually, is an Euler tour. The technique of splicing is used to join these circuits together to form a larger Euler tour.

Splicing Ct2 to Ct5 as detours during the tour of Ct1, results in a larger Euler tour :

(A, B, F1),	... First part of Ct1
(B, B, F2),	... Ct2, a detour from Ct1 at vertex B
(B, C, F2.2), (C, B, F3.2),	... Ct3, a detour from Ct1 at vertex B
(B, C, F2.3), (C, B, F3.3),	... Ct4, a detour from Ct1 at vertex B
(B, C, F2.2),	... part of Ct1
(C, C, F3),	... Ct5, a detour from Ct1 at vertex C
(C, D, F3.1), (D, F, F7), (F, A, F8)	... Last part of Ct1

In the Logon interface STD, the labels on the arcs are the names of interaction functions to be tested. The larger Euler tour developed above is represented in the following sequence, using the names of edges (i.e. interaction functions) only. This gives the test sequence :
TS1 : F1 o F2 o F2.2 o F3.2 o F2.3 o F3.3 o F2.2 o F3 o F3.1 o F7 o F8.

In Chapter 5, it was shown that the execution of the Logon interface begins with the interaction function F1. The two circuits Ct6 and Ct7, as listed above, both start with the edge F1. They are therefore, treated as individual test sequences. This gives the two test sequences :

TS2 : F1 o F4 o F5 o F6, and

TS3 : F1 o F2.1 o F5 o F6.

Used together, the three test sequences, TS1, TS2 and TS3 provide optimal coverage of all the edges in G*. These test sequences are used in Chapter 7, which examines the testing of the Logon interface.

6.5 Other work on state machines and testing

If the test criterion is to cover all nodes rather than all edges, a Hamilton circuit of the STD is required. (A closed walk in a graph that includes all the vertices of the graph just once is a *Hamilton circuit.*) If it is necessary to traverse some vertices more than once (i.e. a Hamilton circuit does not exist), a solution to the travelling salesman problem will provide optimal node coverage [Minieka78]. There is one main difference between the travelling salesman problem and the postman problem. The postman needs to cover all roads (i.e. edges in a graph), whereas the salesman only wants to call at client locations (i.e. nodes in a graph). There are standard solution techniques for producing path sequences for Hamilton circuits and the travelling salesman problem. Examples can be found in pages 617-621 of [Alagar88], and pages 277-283 of [Minieka78]. These standard solution techniques are lengthy; they are applicable, but not specific to the testing of GUIs. Since an Euler tour that traverses all edges of a graph would, in any case, cover all nodes as well, the study of a Hamilton circuit is not pursed in this thesis.

E.P.Hsieh, in [Hsieh71], developed the concept of UIOs (Unique Input/Outputs). This is useful for testing FSMs where the states are not physically visible (or distinctive) to the tester. The UIO test sequences are designed to visualize the state transitions, by observing the distinctive (or unique) output as a consequence of the test input applied to the particular state, during the process of testing.

[Aho88] presents an approach to protocol testing, based on RCPT and UIOs. The graph of an FSM (i.e. the specification) is augmented with additional edges representing the UIOs. An optimal test coverage is, according to [Aho88], a RCPT problem of covering all edges at least once. The RCPT problem is solved by adding replica edges, chosen on a minimum cost flow basis, to produce a symmetric graph. Since the graph is symmetric, an Euler tour exists and gives the required test sequence.

Chapter 7

Testing the Logon user interface

"The subject of test case design is considered to be the crux of software testing. ... There are no known tools that can automatically design test cases ..." from [Myers79]

As revealed earlier in Chapter 2 and 3, the design of test cases is probably the most technically demanding task in the testing process. The scarcity of tools to generate test data is echoed in a number of seminars [Wolverhampton90] and workshops [Durham91]. A comprehensive report on "Computer Aided Software Testing" [Graham90], lists only one tool [PEI90] as being commercially available in Europe that attempts to generate test cases automatically. This tool appears to be primitive and does not cater for user interfaces.

The reasons for the lack of automated test case generation are twofold. As discussed in Chapter 2, functional testing has not been formalized or automated, as functions are largely specified in natural languages. On the other hand, structural testing strategies can assist the selection of test inputs, but do not provide test oracles to check outputs. Considering the phases of the software engineering life cycle, the proper source for deriving test oracles is the specification. The specification is the global reference point for communications amongst designers, programmers, testers and users. For the derivation of expected results in test cases, a formal specification is preferable .

7.1 Survey of testing approaches using formal specifications

Despite the potential for widespread use of formal specification languages, little has been published about deriving test data from such specification [Morell88], [Jones85], [Richardson89]. There was one earlier exploration on the relationship between predicate calculus specifications and path testing [Gourlay81]. A small number of published works, proposing the generation of test cases from formal specifications, exist. An investigation to identify test domains through the partitioning of input and output (sets and states) is presented in [Hall88]. It uses Z to specify a part of the temporary storage (queues) system under IBM CICS. Another report [North90] uses the triangle program as a basis of comparison to contrast the feasibilities of automatic test data generation with specifications made in a number of formal specification languages, including VDM, Miranda and Prolog. The triangle program is concerned with whether inputs consisting of three numbers (or integers) would form the three sides of a triangle. Techniques of equivalence partitioning (into valid and invalid inputs), boundary value analysis and error guessing are used for test data generation. These ideas are not directly applicable to the testing of GUIs, which does not often deal with a range of numbers (or integers) as inputs. Instead, GUIs deal with inputs according to different ranges of positions on the screen. The borders of display objects naturally provide the partitions of inputs according to screen positions.

A tool, presented in [Roper90], attempts to generate test cases by identifying functions and conditions from a test specification for commercial data processing programs. The specification language resembles that of a programming language, thus blemishing the desirable property of a specification as an implementation independent oracle. Moreover, the specification language is so restrictive that it can only be used for very simple problems [Roper90].

Apart from model based specifications, there exists the work of Choquet, Bouge and Gaudel in generating test data sets from algebraic data type specifications implemented in extended versions of Prolog [Choquet86]. However this approach is unsuitable for capturing the input / output features of graphical user interfaces.

There is also the DAISTS (Data-Abstraction Implementation, Specification, and Testing System), which combines a data abstraction implementation language with algebraic axioms in specifications [Gannon81]. In this system, the user writes specification axioms, the implementation, and the test data. The system furnishes the testing process with the test driver and the evaluation of correctness. The axioms are used as test oracles. Verification is carried out between the axioms and the implementation. Structural testing is applied to both the axioms and the implementation to evaluate the test data. The DAISTS system was applied to a practical example of a record-oriented text editor with good results [McMullin83]. However, both the specification and testing of the user interface were omitted in this case study, by assuming that input / output were carried out by side effect.

The evaluation of test cases generated from formal specifications for interactive graphical user interfaces is the main experimental work of this thesis. The direction of test case generation from formal specifications was first inspired by [Hall88], and is coined Formal Functional Testing (FFT) in this thesis. The behaviour of GUIs necessitated the development of a new specification approach and notations, as argued in preceding chapters. The concepts of test domains appear to be very different as functions of GUIs are largely associated with display objects, rather than ranges of numerical values in conventional test domains, as illustrated in [Hall88] or [Choquet86].

7.2 Formal Functional Testing of the Logon interface

[Liskov86] has advocated that specifications should be precise, unambiguous and should be reasonably easy to understand. A WinSpec gives the precise information for test case derivation. Comprehension of the user interface specification is improved when the control flow is presented in a WinSTD. The most natural way to visualise the flow of interaction, is by linking display objects in a state diagram as shown in a WinSTD. The human tester can see clearly the expected visual appearance of objects, together with their respective interaction functions which are to be tested. A WinSTD is useful for detecting any missing objects or functions in the design or implementation. Apart from their use in testing, pictures of display objects (e.g. icons, menus) often have to be made available in documents like user manuals. A WinSTD could also be useful for users to receive early training and for evaluation of the interface. With a WinSTD, human testers would be able to cope with minor changes in layouts of display objects, which may invalidate a whole suite of test cases previously recorded with a playback mechanism. A WinSTD can be easily constructed as most window systems can produce screen dumps on paper. The tester has to add the arcs joining objects, and then identify and enumerate all objects and functions for testing. Alternatively, there are tools (e.g. [OSU89]) which aim to assist the user interface designer to produce design drawings of display objects and to define interaction sequences.

In the first instance, testing experiments were conducted with the Logon user interface, to derive test cases according to four different criteria :

TC1 - 100% coverage of objects
TC2 - 100% coverage of messages
TC3 - 100% coverage of functions
TC4 - 100% coverage of interaction sequences (all possible combinations of functions)

For functional testing, as surveyed in Chapter 2 (section 2.3), the conventional test coverage are conducted in terms of ranges of inputs and outputs. Equivalence partitioning and boundary value analysis are such examples. For graphical user interfaces, inputs are directed towards specific display objects. Outputs, from the GUI software, are the display of objects and the dispatch of messages to the underlying application program.

"Object coverage" (TC1) is used in the testing process to mean :

- Inputs are given to the GUI so that each display object should appear at least once during the interactions performed in the test sequence.
- These display objects are inspected visually to ensure the correct visual appearance.
- At least one interaction function is carried out with each of these display objects to check that these objects are alive (responding to the input correctly).

Since display objects are outputs of GUIs, TC1 can be related to the output partitioning as used in conventional test coverage.

"Message coverage" (TC2) requires the testing of those functions that have app_msg_sent or app_msg_recv specified in their WinSpec specifications. In WinSpec, an app_msg_recv is regarded as an input to the interface, and an app_msg_sent as an output. Thus, TC2 can be viewed as input and output partitioning, when related to conventional test coverage.

"Function coverage" (TC3) requires the testing of all the interaction functions for the GUI, as specified in WinSpec. Clearly, TC3 embraces both TC1 and TC2, as all display objects and messages are captured in the specification of functions. TC3 is stronger than both TC1 and TC2 combined, because many display objects have more than one associated interaction function. As ranges of acceptable user inputs are clearly specified for each function, function coverage effectively provides the equivalence partitioning used in conventional test coverage.

"Interaction sequence coverage" (TC4) requires the testing of all combinations of interaction functions. This is not always possible or practical, due to the large number of possible combinations of interaction sequences. This is similar to the path explosion situation mentioned in section 2.2, when examining the path coverage criterion used in structural testing.

It soon becomes apparent that test cases for object coverage and function coverage are relatively similar. This is because the interactions necessary to test all objects are likely to cover a large number of functions as well. Message coverage is found to be the weakest criterion, as many interaction functions are achieved entirely within the GUI without any communications with the corresponding application. The number of application messages is dependent on the model of dialogue separation used, such as "Macro-communication" and "Micro-Communication", as discussed in Chapter 3. It is found that a 100% functional coverage is the strongest of the first three criteria listed above (i.e. exposes more errors). There are cases where a 100% coverage of objects and messages might not guarantee a 100% coverage of functions. For instance, two objects being tested (text boxes texB_user and texB_pass) seem to pass the object coverage as they have the right visual appearance in accepting and echoing keyboard inputs. The error is only uncovered when keyboard inputs kb?=<tab> and kb?=<cr> are identified as having special functions according to the specification. This example is taken from error E7 listed in Section 7.4.

Another perspective of the coverage criteria can be gained by referring to the theoretical foundation of FSMs on which the specification is based. The 100% object coverage is related to the traverse of all the nodes (or vertices) in the STD for the user interface. A 100% function coverage requires the traversal of all edges at least once. A 100% coverage of interaction sequences requires the testing of all possible paths in the STD.

Chapter 6 has given the detailed development of the procedure for the derivation of test sequences for TC3 (i.e. a 100% function coverage). This is based on the graph theory of Euler tour and a solution of the postman tour. This will also satisfy TC1, as a coverage of all edges will inevitably cover all the nodes in the STD as well.

There is no attempt to generate all alpha-numeric keys for input testing, because the ability to accept keyboard inputs belongs to the underlying window system and device driver. What is being tested is the GUI's ability to pass keyboard inputs correctly to other parts of the application program. It does not aim to test the other part of the application that actually undertakes the authorization check of the username and password against the authorization database. There is one element of information necessary for test generation but not available in the specification, and that is pairs of valid and invalid "username - password". Since the list of valid usernames and passwords varies from system to system, it is assumed that the tester has knowledge of such information from other sources.

In order to evaluate the test selection criteria, code changes to seed ten errors were introduced. The 100% functional coverage approach actually generated test cases that uncovered 9 out of the 10 errors injected in the logon GUI. The 100% object coverage found less than 8 of the 10 errors. This approach of generating functional tests from a formal specification appears to be successful for the logon user interface. Though academically interesting and not too time consuming to follow through, the Logon GUI is a very small program. Case studies of other more complex GUIs are given in later chapters.

7.3 Listing of Test Cases

A total of 14 functions have been specified for the Logon interface, in section 5.7.
F_Logon = {F1, F2, F2.1, F2.2, F2.3, F3, F3.1, F3.2, F3.3, F4, F5, F6, F7, F8}

Three test sequences, as listed in the following sections, are found to be necessary to cover all 14 functions. These test sequences are developed in Chapter 6, using the graph theoretic algorithms of the Euler tour and the postman tour. As mentioned earlier, the required inputs are taken from the "Inputs" clauses, and the expected outputs (oracles) from the T_state_predicates. Functions are exercised one after another in a sequence, as the To_ and From_states permit. The test selection criterion is a 100% coverage of all functions (at least once). It can be seen that once functions are specified in a WinSpec, test cases of interaction sequences can be represented in very concise and unambiguous notations, such as : F1 o F2 o F2.2 o F3.2 o F2.3 o F3.3 o F2.2 o F3 o F3.1 o F7 o F8.

Test case (A) test sequence TS1:
F1 o F2 o F2.2 o F3.2 o F2.3 o F3.3 o F2.2 o F3 o F3.1 o F7 o F8

Required input	Function	Expected O/P	Comments
is_inside (mp?, icon_'Logon') Tand mb?=<dClick>	F1	is_visible(diaB_'Logon') and has_kb_focus(texB_user)	! Invoke Logon icon, ! dialogue box appears, ! username text box ! has input focus.
kb?= "god"<del> "oduser"	F2	text(texB_user)=kb?	! Input to username ! text box.
kb?=<tab>	F2.2	has_kb_focus(texB_pass)	! Password text box ! has input focus.
kb?=<tab>	F3.2	has_kb_focus(texB_user)	! Return key focus to ! username text box .
is_inside (mp?, texB_pass) Tand mb?=<click>	F2.3	has_kb_focus(texB_pass)	! Password text box ! has input focus.

Required input	Function	Expected O/P	Comments
is_inside (mp?, texB_user) Tand mb?=<click>	F3.3	has_kb_focus(texB_user)	! Return key focus to ! username field.
kb?=<tab>	F2.2	has_kb_focus(texB_pass)	! Password text box ! has input focus.
kb?="goodpasw" <del>"sword"	F3	is_not_visible (text(texB_pass)=kb?)	! Text in password ! field is not echoed.
kb?=<cr>	F3.1	is_hiLit(cBtn_'OK') app_msg_sent = ("user=",text(texB_user), "pass=",text(texB_pass))	! Ok button is highlighted ! Message of ! username and ! password sent.
app_msg_recv = "Logon ok"	F7	is_not_visible(diaB_'Logon') is_visible(wind_term) is_hiLit(wind_term)	! diaB disappears, ! terminal window ! appears as active.
is_inside (mp?, cloB_term) Tand mb?=<click>	F8	is_not_visible(wind_term)	! Click at close box of ! terminal window, ! window disappears .

Test case (B) test sequence TS2: F1 o F4 o F5 o F6

Required input	Function	Expected O/P	Comments
is_inside (mp?, icon_'Logon') Tand mb?=<dClick>	F1	is_visible(diaB_'Logon') and has_kb_focus(texB_user)	! Invoke Logon icon, ! dialogue box appears, ! username text box ! has input focus.
is_inside (mp?, cBtn_'OK') Tand mb?=<click>	F4	is_hiLit(cBtn_'OK') Tand app_msg_sent = ("user=",text(texB_user), "pass=",text(texB_pass))	! Ok button is highlighted ! Message of ! username and ! password sent.
app_msg_recv = "Logon failure"	F5	is_not_visible(diaB_'Logon') is_visible(diaB_'Logon Failure') is_hiLit(diaB_'Logon Failure')	! diaB disappears, ! Logon failure dialogue ! box appears as active.
is_inside (mp?, cBtn_'Reset') Tand mb?=<click>	F6	is_not_visible (diaB_'Logon Failure')	! Click at reset command ! causes Logon Failure ! dialogue box to ! disappear.

Test case (C) test sequence TS3: F1o F2.1 o F5 o F6

Required input	Function	Expected O/P	Comments
is_inside (mp?, icon_'Logon') Tand mb?=<dClick>	F1	is_visible(diaB_'Logon') and has_kb_focus(texB_user)	! Invoke Logon icon, ! dialogue box appears, ! username text box ! has input focus.
kb?=<cr>	F2.1	is_hiLit(cBtn_'OK') Tand app_msg_sent=("user=",text(texB_user), "pass=", text(texB_pass))	! Ok button is highlighted ! Message of ! username and ! password sent.
app_msg_recv = "Logon failure"	F5	is_not_visible(diaB_'Logon') is_visible(diaB_'Logon Failure') is_hiLit(diaB_'Logon Failure')	! diaB disappears, ! Logon failure dialogue ! box appears as active.
is_inside (mp?, cBtn_'Reset') Tand mb?=<click>	F6	is_not_visible (diaB_'Logon Failure')	! Click at reset command ! button causes Logon ! Failure dialogue box ! to disappear.

In order to check that all interaction functions for the Logon interface have indeed been covered by these three test sequences, an analysis is made as follows. The set of all functions in the Logon interface is denoted as F_Logon. From the WinSTD and WinSpec developed for the Logon interface in chapter 5, it is established that :
F_Logon = {F1, F2, F2.1, F2.2, F2.3, F3, F3.1, F3.2, F3.3, F4, F5, F6, F7, F8}

From the three test cases listed above, it can be seen that :
Functions covered by sequence TS1, F_TS1 = {F1, F2, F2.2, F3.2, F2.3, F3.3, F3, F3.1, F7, F8}
Functions covered by sequence TS2, F_TS2 = {F1, F4, F5, F6}
Functions covered by sequence TS3, F_TS3 = {F1, F2.1, F5, F6}

The set of functions covered by the three test sequences is the union of the three sets :

$F\_TS1 \cup F\_TS2 \cup F\_TS3$

= {F1, F2, F2.1, F2.2, F2.3, F3, F3.1, F3.2, F3.3, F4, F5, F6, F7, F8}

= F_Logon

This confirms that all interaction functions of the Logon interface have been tested.

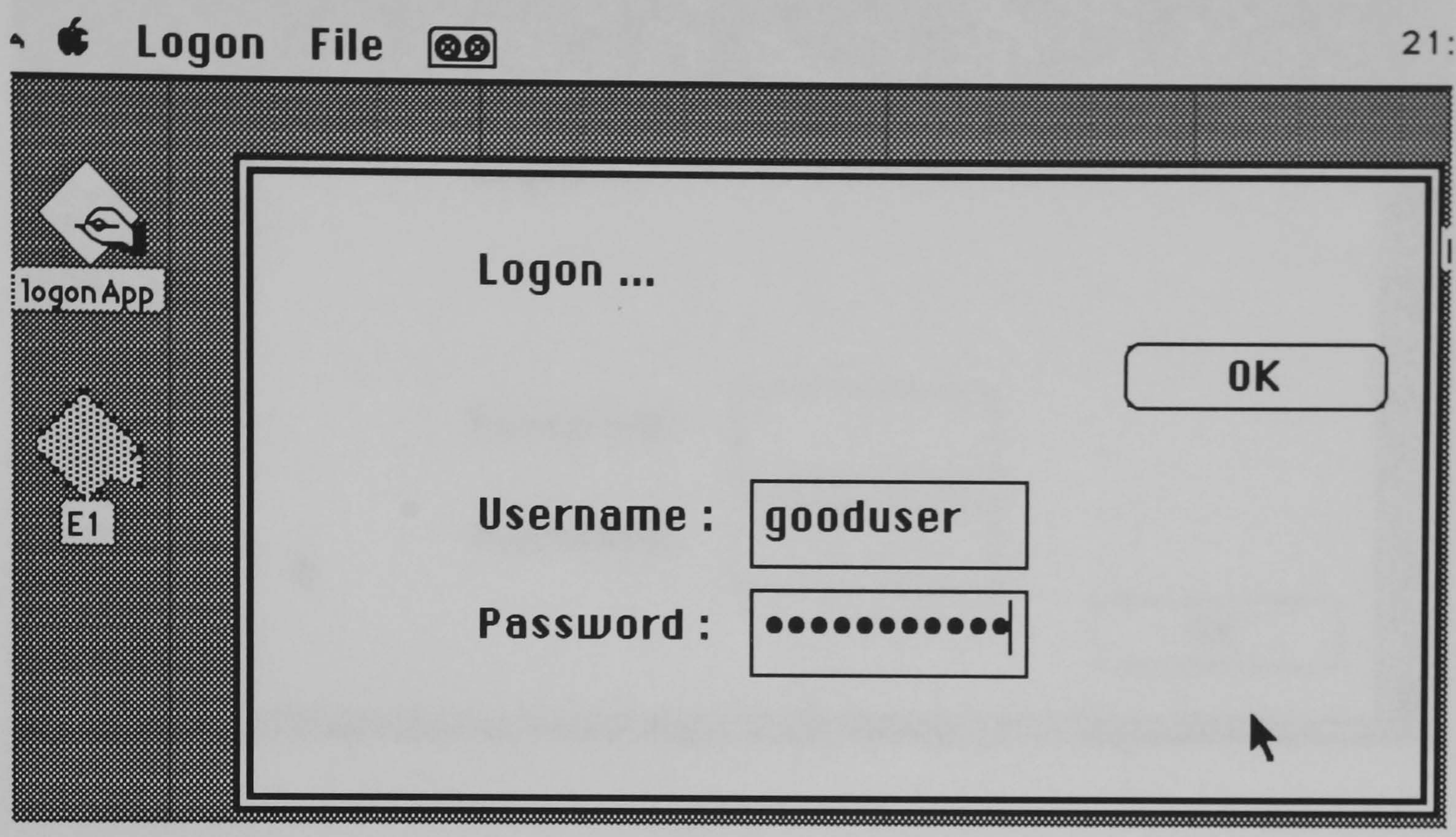
7.4 Results of testing

The following table presents the results of this testing experiment with the logon user interface. Nine out of the ten errors seeded were detected by the functional coverage criterion that requires each interaction function be invoked at least once. The undetected error was that of a very small shift in the screen position of the username textBox. Even in this case the textBox functioned normally, as the defect was purely visual. The reason why only 10 errors were injected is because the logon user interface is too small to warrant any more meaningful errors. Screen dumps of some visible symptoms caused by injected errors are given in section 7.5. An analysis of common error types in GUIs is presented in Chapter 12.

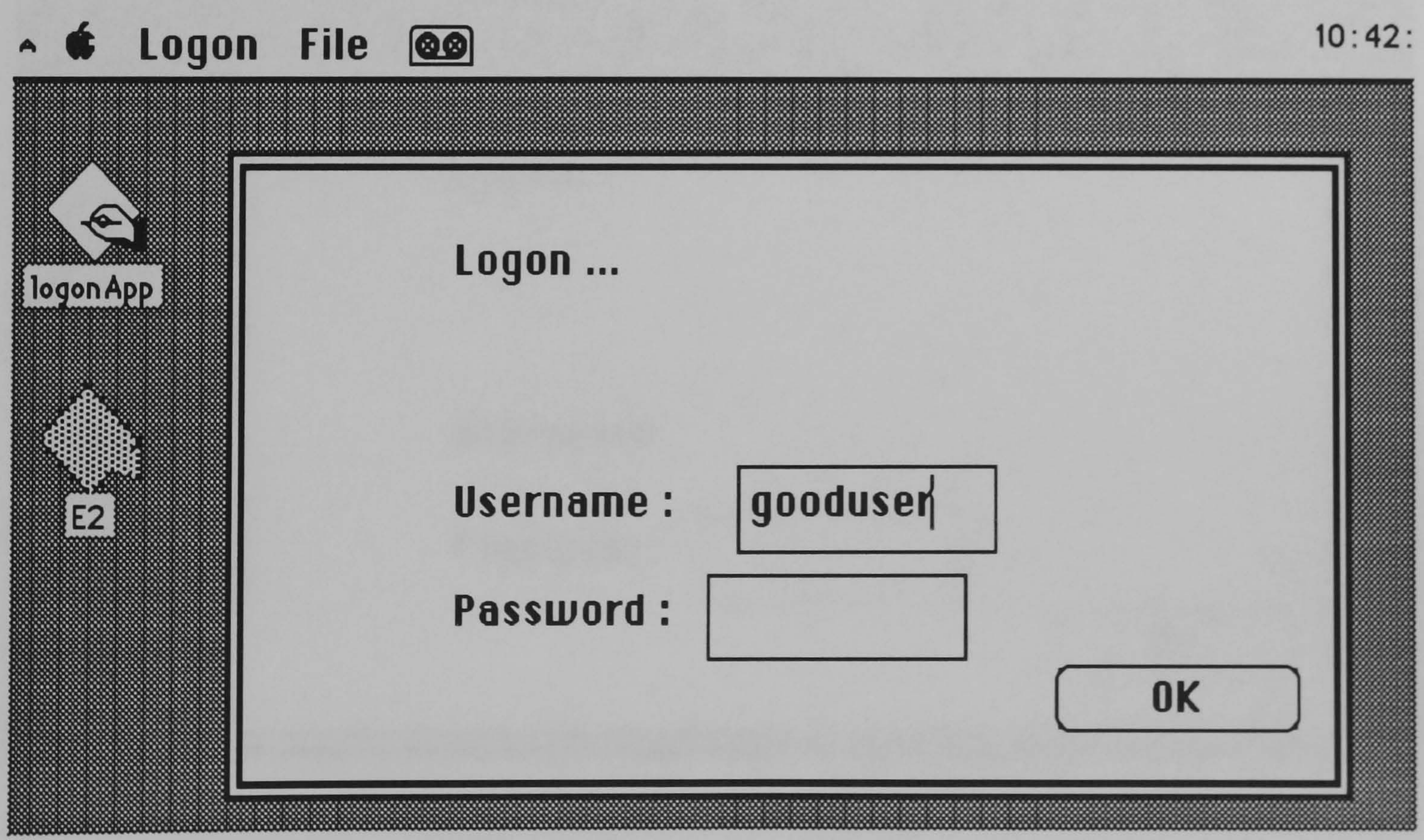
Error	Brief description	Object coverage (Visual inspection of all objects)	Function coverage (Test sequences from WinSpec)
E1	"Ok" button at wrong location	detected	detected
E2	Text box (texB_user) misplaced	detected	undetected (see section 12.1)
E3	labels "username" and "password" swapped	detected	detected (logon failure)
E4	texB_user missing	detected	detected
E5	"Reset" command button has no function	detected	detected
E6	"Ok" command button has no function	detected	detected
E7	<tab> & <cr> has no function at texB_user and texB_pass	undetected	detected
E8	wind_term is displayed at logon failure	detected (not always)	detected
E9	diaB_'Logon Failure' is displayed at logon success	detected (not always)	detected
E10	text(texB_user) and text(texB_pass) swapped in app_msg_sent	undetected	detected

7.5 Screen prints of some visible symptoms

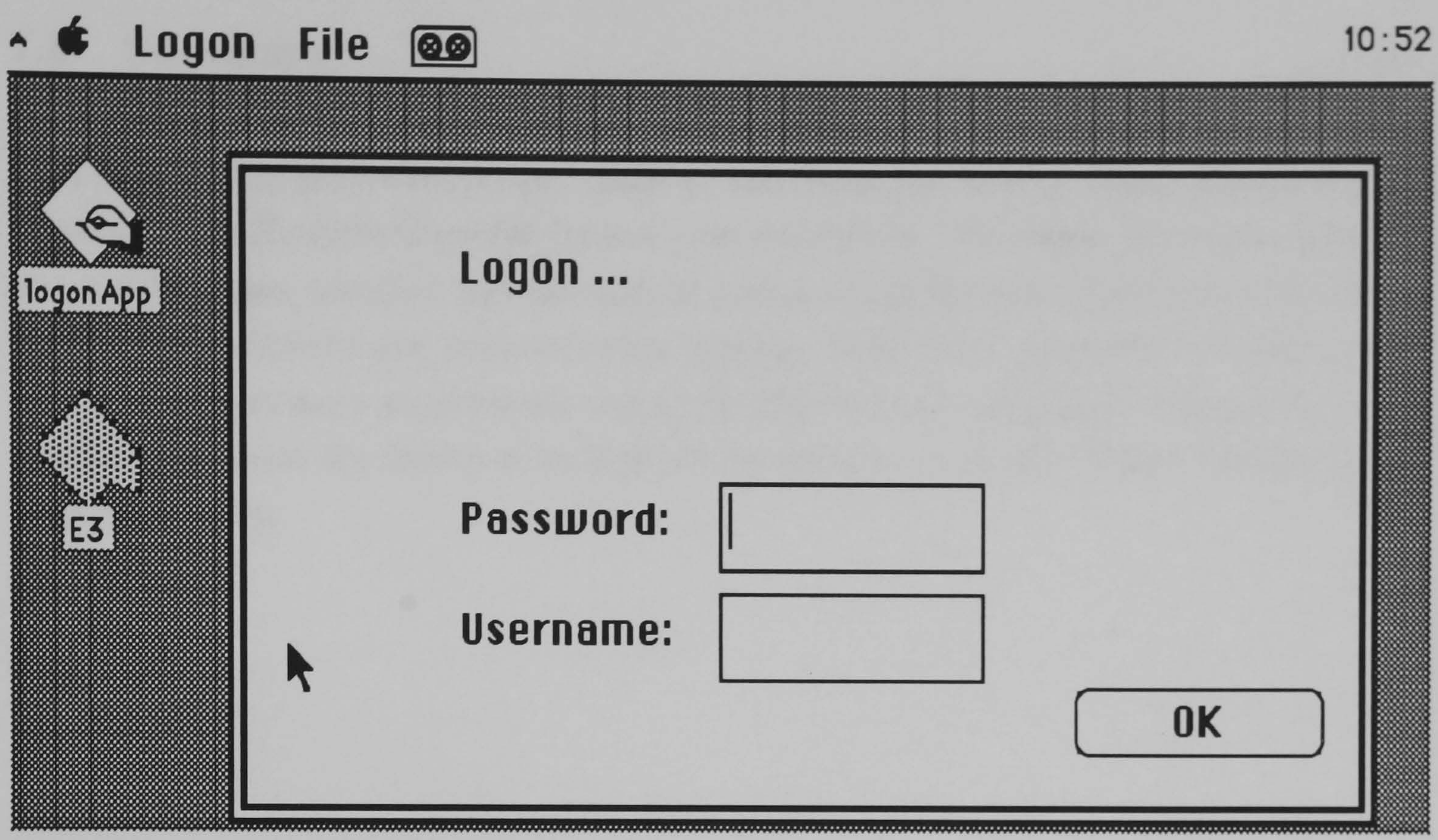
Error E1 : Command Button "OK" (cBtn_'OK') dislocated



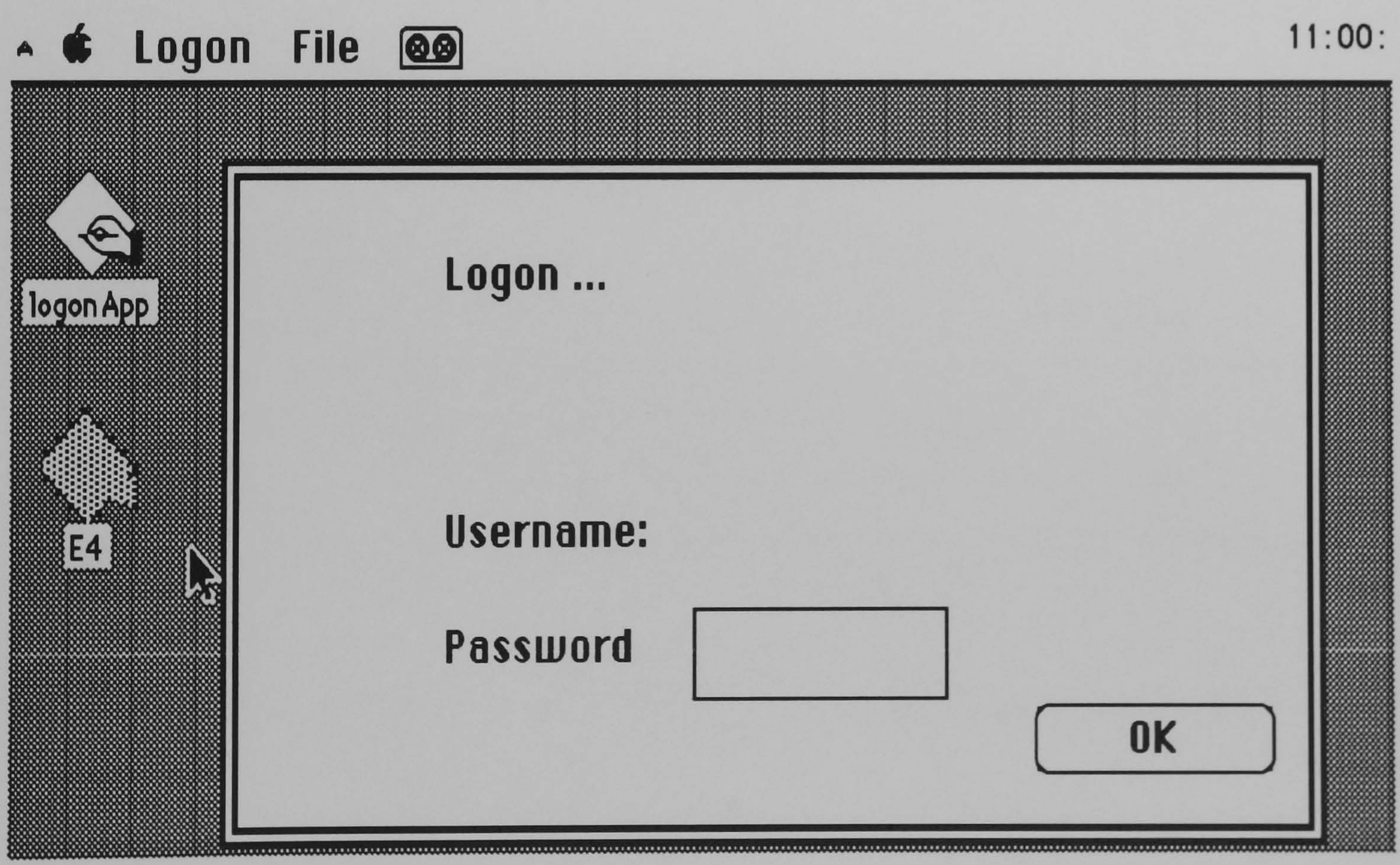
Error E2 : Text box (texB_user) misplaced



Error E3 : Labels “Username” and “password” swapped



Error E4 : Username entry field (texB_user) missing



7.6 Summary

This chapter has presented a case study of user interface testing. It has shown that the WinSpec specification is useful for test case derivation. The results are encouraging as the specification notation has formalized and assisted the reasoning and selection of interaction functions and sequences for testing. Only three interaction sequences are found to be necessary to cover the testing of all functions. Moreover, the quality of the selected test cases are shown to be high, in detecting 9 out of the 10 errors seeded in the Logon interface.

Chapter 8

Specifications for ThinkEdit

Window-based text editing is often an important part of graphical user interfaces. In previous chapters, the specification model and notations were applied to a simple GUI. An investigation of the same approach to text editing is presented here, following the usual steps of scientific development : observation, modelling and experiments. The investigation begins with the observation of the behaviour (functions) of existing window-based text editors. The modelling process is an attempt to describe the editing functions with the WinSpec notations, extending the notations as necessary. Once these editing functions are expressed in WinSpec, test cases will be derived from the specification. The success of such a functional testing approach is then judged by its ability to find errors in editors during the testing experiments.

8.1 Natural language description of ThinkEdit functions

The editor being used in this case study is called ThinkEdit, which runs in the Macintosh environment. It is part of an object-oriented programming tutorial set included in THINK Pascal, a commercially available software package [Symantec90]. The source code of this editing GUI occupies a total of about 40 pages, approximately two to three thousand lines of Pascal. There is no user manual for this editor. The following is an informal specification (or description) of its functions. It is mainly drawn from comments in the Pascal source code, and modified to improve coherence and to include the user's perspective. This informal specification can be seen as ambiguous, imprecise and incomplete, exemplifying the weaknesses of an informal natural language specification. A number of vital functions are missing in this specification, probably due to the assumption that readers are familiar with mouse and menu driven editors. The informal specification begins by describing the functions of the two main menus ("File" and "Edit") of ThinkEdit, together with the various menu options within these menus. The "Apple" menu is unrelated to ThinkEdit functions and too specific to the Macintosh environment to warrant a general discussion.

8.1.1 The "File" menu

The "File" menu has menu options for the "New" , "Open" , "Close" , "Save" , "Save As..." , "Quit" commands, described as following :

- The "New" command opens an empty window with an "UntitledX" name (where X is an integer starting from 1, to a maximum of 5).
- The "Open" command prompts the user to enter the name of a file to edit with the Open File dialog box. If the name is the same as a file that is already open, the user is prompted to open the file with an "UntitledX" name. If ReadFile is successful, the new window is opened after any existing windows are updated.

- The "Save" command simply writes the file to the disk and resets various flags.
- The "Save As" command puts up the Save File dialog box to prompt the user to enter a filename in which to save the editing. For "Save As", the window title and associated "Windows" menu item are changed.
- When the user attempts to "Quit", or "Close" a file; the "textDirty" flag is tested. If the text is dirty (i.e. content changed since last "Save"), the user is prompted with a dialog box to Save, Discard, or Cancel the save operation. In the exceptional case of quitting after running out of memory, the only options are Save and Discard.

- Upon invocation of the "Close" option on the "File" menu, or when the close box on the front window is clicked. Actions are taken according to three possibilities :
  (a) The window is a user interface window, in which case it is hidden.
  (b) The window is the Clipboard, in which case it is restored to normal size if it is zoomed, and hidden.
  (c) The window is an Edit window, in which case an attempt is made to save the associated file, and if successful, before the window is closed it is first restored to normal size if it is zoomed. (Note that a natural language description allows undefined terms, e.g. "zoomed" , to be used ambiguously.)

• When the "Quit" command is invoked, or if the program runs out of memory, an attempt is made to save all open files, and if successful, the global variable "Done" is set to TRUE.

8.1.2 The "Edit" menu

The menu option provides the edit commands : "Cut", "Copy", "Paste", "Select All"; commands that affect the scrap and text selection (highlighted, selected through mouse interactions).

• The "Cut" command deletes the selection and places its content onto the scrap.

• The "Copy" command copies the selection onto the scrap without deleting it.

• The "Paste" command first tests if the text length would exceed 32767 before doing the paste. This is the maximum text length that the editor can deal with.

8.1.3 Mouse pointer and button inputs

The editor employs the mouse as one of its main input devices, with functions as follows :

• Mouse-downs in a scroll bar : all parts of the control other than the thumb are handled in a uniform manner. (This exemplifies how unclear a natural language specification can be.)

• Mouse-downs in the up-line, down-line, up-page, or down-page regions of the scroll bar : it first determines which scroll bar the mouse was clicked in so that the proper page size is used. The value of the associated control is then updated and the text is scrolled if necessary. The actions are described in the Control Manager section of "Inside Macintosh".

• AutoScroll is as described in the TextEdit Programmer's Guide in Inside Macintosh. After the mouse was first clicked in the text window, for as long as the user holds down the mouse button, the text in the window is scrolled up or down repeatedly, depending upon whether the mouse is above or below the text.

• SignalZoom signals that the current front Edit window is zoomed by inverting the grow icon. It must validate the grow icon after inverting it so that it will not be drawn over when updating the window.

• HandleGrow grows the current front window when the user drags on the grow icon, or zooms the window by double-clicking on the title bar of the window. The two parameters "hSize" and "vSize" are the new height and width of the window with respect to the top-left corner of the window. The window, scroll bars, and text in view are first resized. It then validates any remaining uncovered portion of the window's text if the text was not scrolled. This makes for a much cleaner update.

• ZoomWindow zooms the current front window in or out when the user double-clicks on the title bar. When zooming out, the original size of the window is stored and the window is resized to fill the entire screen. When zooming in, the window is restored to its original size even if it was resized while it was zoomed. The "show" parameter is used to hide and unzoom a window, typically the Clipboard, when the window is closed.

• Mouse clicks in the title bar of a given window : if the user double clicks and the window is not the front window, it is brought to the front. If it is the front window, it is zoomed. Otherwise, the window is dragged to a new location.
• Mouse-downs in the text or scroll bars of the current front window are dealt with accordingly. Note that mouse-downs in the text of non-Edit windows are ignored.

• If the mouse was clicked in the content or controls of an Editor created window that is not the current front window, the window is first activated as per Macintosh User Interface guide-lines.

8.1.4 Keyboard Inputs

The keyboard is the other main input device of the editor. It first tests if the key is a menu key equivalent. If so, control is dispatched to the menu handler. Otherwise, it must test if an Edit window is the current front window. If so, it tests if :
• the key is a printing character, a <Cr>, or a <BS>, and if
• there is enough room in the text buffer to insert the character. If so, the character is inserted and various flags are updated.

8.1.5 Housekeeping

Various housekeeping functions exist (upon Activate/Deactivate events) to set flags and menu items as appropriate:
• Initially as the editor is launched, to create the Clipboard window, and prompt the user to open a file. The main event loop is then executed repeatedly until the user quits. Note that the Desktop scrap is updated when exiting the program.
• It changes the cursor to an I-Beam if it is over the text of an active Edit window. The insertion point in the text is also flashed on and off here. Otherwise, the cursor is changed to the Arrow. The previous status of the cursor has to be kept (by means of a global variable "InWindow").

Apart from the main editing and display functions that will be specified and discussed in detail in the following sections, other common features of editors are :
• justification of the text display (flush left, centred or flush right).
• word wrapping.
• search (or find) and replace (or change).
• cursor movement by arrow keys.
• select word, line, all.
• right-delete, that is delete forwards as different from the normal delete key that deletes the last character (i.e. delete backwards).
• delete word, line.
• undelete.

These functions do not exist in ThinkEdit, and are not included in the following discussion on specification.

8.2 Specification approaches for editors

There are a few published papers on the formal specification of text editors, such as [Sufrin82], [Took90]. Some of these specifications are highly abstract and mathematical. They are intended for mathematical proofs rather than for derivation of test data. These specifications have concentrated on abstract models of the internal data types. Often, little explicit description is given to display functions, which provide visual feedback of interactions to users and testers. This is partly due to varying levels of abstraction, and also to the fact that some of these specifications were produced before the advent of GUIs.

The main difference in the specification approach presented in this thesis is in its explicit statements on inputs (required as test data) and outputs (for visual checks). It tells the tester what inputs to generate and what outputs to expect, as stated in the "Inputs" and "T_state_predicate" clauses. It also assists the understanding and testing of interaction sequences, by allowing functions of matching To_ and From_states to be considered for execution in sequences.

WinSpec notations are adequate for specifying inputs to a text editing GUI. The three basic input sources remain as : keyboard, mouse pointer and mouse button. But WinSpec constructs for describing visual outputs were found to be inadequate and had to be extended. Following the specification approach developed in Chapter 5, the first step was to identify all display objects and draw up the WinSTDs for ThinkEdit, as given in the following section. The second step is to specify interaction functions in WinSpec notations, which is presented in section 8.4 .

8.2.1 A WinSTD for ThinkEdit

In order to construct the WinSTDs for ThinkEdit, all display objects must be identified. The following WinSTD shows only the high level objects and functions, as detailed specification of interaction functions are being developed in the following sections.

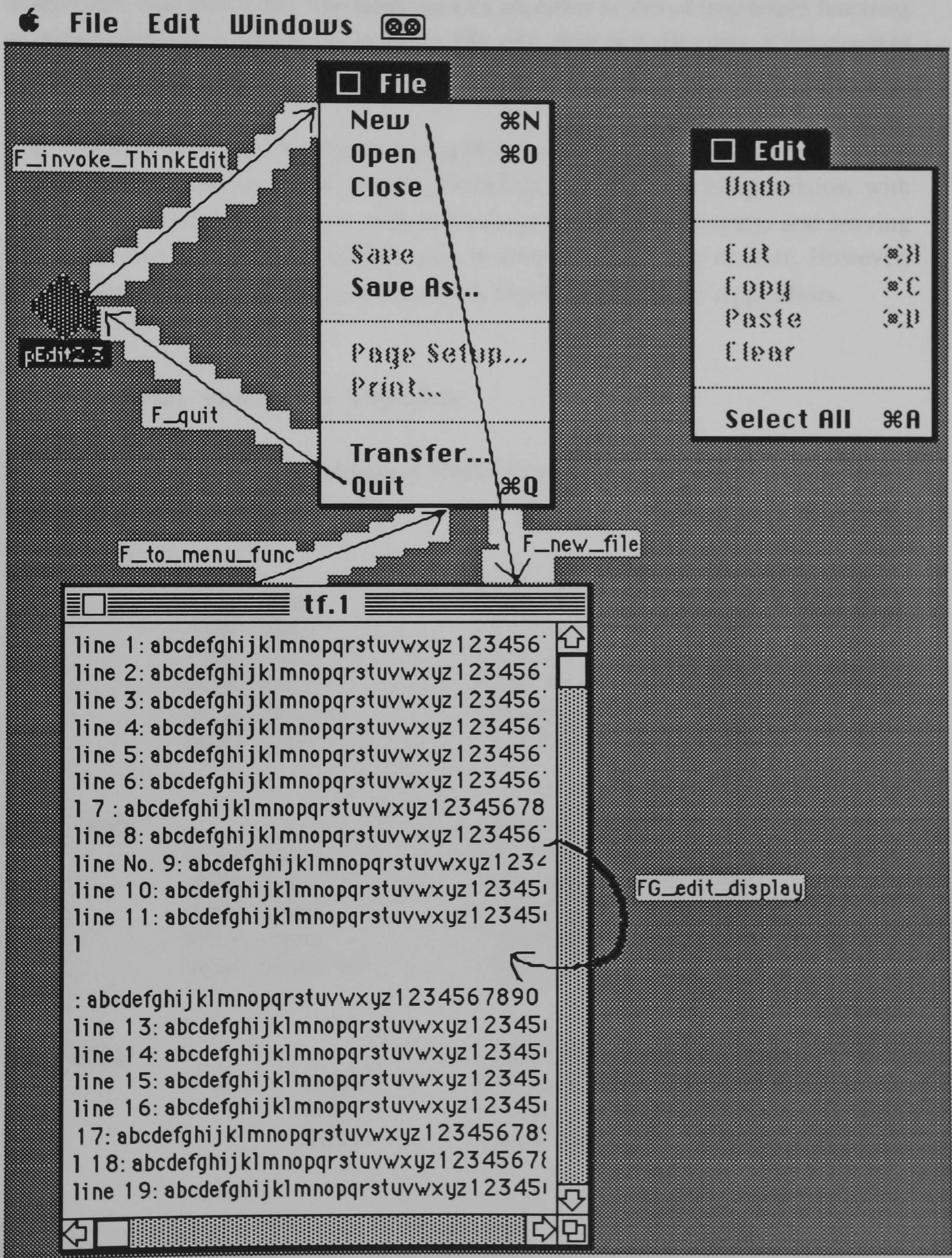


Figure 8.1 A WinSTD for ThinkEdit

The above WinSTD shows the visual appearance of the "File" and "Edit" menu, together with an editing window, which are the top level display objects in ThinkEdit. The lower level display objects are identified and listed in section 8.2.2.

The WinSTD visualizes the control flow in ThinkEdit, where arcs and arrows are used to represent state transitions. The labels on arcs are either names of interaction functions or names of function groups. For instance, FG_edit_display is the name of the group of edit display functions.

The WinSTD (in Figure 8.1) gives a simplified overview of interaction in invoking ThinkEdit with the function F_invoke_ThinkEdit, creating an editing window with F_new_file, performing editing with function group FG_edit_display, and leaving ThinkEdit with F_quit. The simplification is necessary for space reasons. However, detailed state diagrams are given in Figure 8.3, Figure 9.1 and in the Appendices.

8.2.2 Display objects for ThinkEdit

In the following table, display objects in ThinkEdit are identified; the hierarchy is shown by indentation (e.g. viewRect is part of wind_editor) :

menu_'File'			! The "File" menu
	mOpt_'New'		! The "New" menu option in "File" menu
	mOpt_'Open'		! The "Open" menu option
	mOpt_'Close'		! The "Close" menu option
	mOpt_'Save'		! The "Save" menu option
	mOpt_'SaveAs'		! The "SaveAs" menu option
	mOpt_'Quit'		! The "Quit" menu option[5]
menu_'Edit'			! The "Edit" menu
	mOpt_'Cut'		! The "Cut" option in the "Edit" menu
	mOpt_'Copy'		! The "Copy" menu option
	mOpt_'Paste'		! The "Paste" menu option
	mOpt_'Clear'		! The "Clear" menu option
	mOpt_'Select All'		! The "Select All" menu option
wind_editor			! The editing window, its title is not "editor"
	tBar	strips	! It has strips on its title bar,
		cloB	! It has a close box, and
		title	! a title of the name of the file being edited.

[5] The Page Setup, Print, Transfer and Undo options are either unsupported or irrelevant.

```
destRect                         ! The editor window has a destination
              destRect(1,1)      ! rectangle of (destLen X destWidth) chars.
              ...                ! i.e. total no of lines = destLen, with
              destRect(x, y)     ! no. of chars along each line = destWidth.
              ...                ! Char y in line x is denoted destRect(x, y).
              destRect(destLen, destWidth)

viewRect                         ! The editor window has a view rectangle
              viewRect(offset+1, 1)
              ...                ! of length viewLen and width viewWidth,
              viewRect(offset+viewLen, viewWidth)
                                 ! at an offset from the destRect.

vBar                             ! The window has a vertical scroll bar
              sBar_upArrow       ! The up arrow shape on the scroll bar
              sBar_pgUpRect      ! The page up region on the scroll bar
              sBar_SlideBox      ! The slider box on the scroll bar
              sBar_pgDnRect      ! The page down region on the scroll bar
              sBar_dnArrow       ! The down arrow shape on the scroll bar

hBar                             ! The window has a horizontal scroll bar
              sBar_ltArrow       ! The left arrow shape on the scroll bar
              sBar_pgLtRect      ! The page left region on the scroll bar
              sBar_hSliderB      ! The slider box on the scroll bar
              sBar_pgRtRect      ! The page right region on the scroll bar
              sBar_rtArrow       ! The right arrow shape on the scroll bar

sizB                             ! It has a resize box.
```

Table 8.1 Display objects of ThinkEdit

The above table and WinSTD have identified the basic display objects of ThinkEdit.
ThinkEdit _obj_types = {menu, mOpt, wind, tBar, destRect, viewRect, sBar, sizB}

These generic object types have already been introduced in section 5.4.2, except for destRect and viewRect which are introduced for the specification of ThinkEdit, as described in the following section (8.3). Moreover, it can be seen in the above table, a vertical scroll bar consists of a number of lower level display objects :
vBar = {sBar_upArrow, sBar_pgUpRect, sBar_slideBox, sBar_pgDnRect,
sBar_dnArrow}

Functions of horizontal scroll bars are similar to those of the vertical ones. In the following specifications, horizontal scroll bars are not used by assuming viewWidth = destWidth. The above denotation of display objects permits the smallest components to be identified precisely. For instance, the down arrow in the scroll bar of the third editing

window can be unambiguously specified as :
wind_editor#3.vBar.sBar_dnArrow .

The user interaction to scroll the text display downwards by a line can be denoted as :
is_inside (mp?, wind_editor#3.vBar.sBar_dnArrow) Tand mb? = <click>

When there is no risk of ambiguity in identifying the display object being referred to in a specification, the above can be simplified to :
is_inside (mp?, sBar_dnArrow) Tand mb? = <click>

8.3 Text formatting, destRect and viewRect

An interactive editor provides visual feedback by allowing the user to see the text entries and changes on a display screen. Since there can be more text in the file than can be displayed on screen, two terms are used: destination rectangle (destRect) and view rectangle (viewRect). Both rectangles are expressed in the same coordinate system, with the origin (of (x,y) coordinates) fixed at the top left corner of the destRect. The x coordinate increases positively from the origin towards the right, and the y coordinate increases positively from the origin downwards. The destination rectangle marks the boundaries within which the text will be set. The view rectangle defines the portion of the window in which the text will actually be displayed [Chernicoff88]. The general principle is to keep in view the region of the document where changes are being made. The two rectangles need not coincide, and in general they do not. Scrolling is performed by shifting the destRect while holding the viewRect fixed.

Figure 8.2 illustrates the relationship between destRect and viewRect. The coordinates used to specify character positions in the viewRect refer to the same origin (i.e. the top left corner of the destRect), as the viewRect always lies within the borders of the destRect. When the text reaches the right edge of the destRect, it is automatically wrapped to the next line. The destRect is actually bottomless; only its top and sides are significant.

The basic functions of any text editor are the entry of new text, modification of existing text, and eventually saving the edited text to a disk file. An editor normally keeps a working copy of the text file being edited in temporary storage denoted as "text(record)" here. This working copy, text(record), is usually more up to date than the disk file which is denoted as "text(file)". When a "save" command is selected by the user during editing, text(file) becomes the same as text(record).

The relationship between viewRect, destRect, text(record) and text(file) are further explained as follows :

- The viewRect is all that the user can see on the screen. Effectively, the viewRect captures the text that is in view in the editing window at the time. The user can choose to view a different part of the text by scrolling. The viewRect always lies within the destRect, to expose part or all of the text in the destRect. One can image that the viewRect is being moved around to allow the user to see other parts of the text within the destRect.

- The destRect captures all the text, formatted within rectangular borders, ready for screen display. Normally, only a portion of the destRect is actually visible to the user, through the opening that is called the viewRect. Effectively, the text that is seen in the viewRect is actually part of the text in the destRect.

- The text in temporary storage, text(record), is different from the contents of destRect in that the text in text(record) is sequential and not formatted for display as in a rectangular window. It is necessary to introduce text(record), as the content of destRect is subjected to formatting constraints such as line folding at carriage-returns.

- The notion of "text(file)" to represent the content of the disk file, as different from the temporary storage (i.e. text(record)), is purely for the sake of completeness. The specification for ThinkEdit, as given in the following section, does not actually refer to text(file). The task to update the disk file belongs to the underlying application, and not the user interface. The ThinkEdit user interface only sends a message to the underlying application, requesting the content of text(record) to be saved.

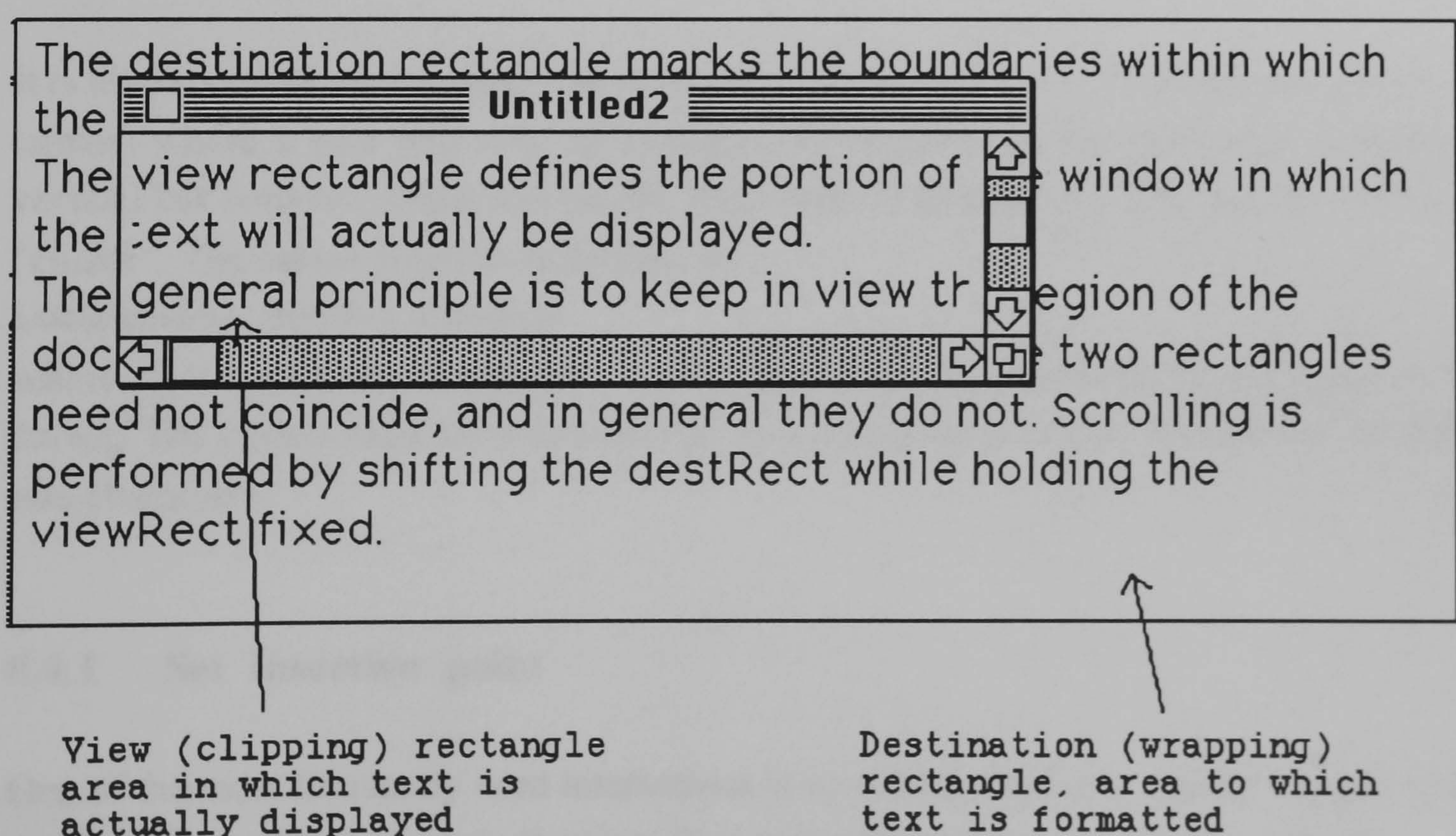


Figure 8.2 destRect and viewRect

8.4 Specification of edit and display functions

When the mouse pointer is inside the editing rectangle (i.e. viewRect), its shape is changed from that of an arrow pointer to that of a marker (a vertical bar resembling a book marker). A variant of the pointer (mp?) notation is used here to represent the marker (mk?) input. With mp? inputs, locations can be represented in the smallest display unit of the screen (i.e. pixels). With mk? inputs, it is necessary to model the viewRect (and the destRect) with coordinates that are effectively represented in terms of line numbers (lineNo) within the text, and character numbers (charNo) within a line. Therefore when the mouse pointer is inside a text editing rectangle, instead of
Loc(mp?) = (x,y) , the notations are changed to :
Loc(mk?) = (lineNo, charNo)
where $1 \le$ lineNo $\le$ destLen and $1 \le$ charNo $\le$ destWidth.

As described in section 8.2, the number of lines (destLen) in a destRect is increased as new text entries are made. The length (destLen) of a destRect is itself a variable. However, the width of a destRect or the maximum number of characters per line (destWidth) is fixed. Text entries longer than destWidth are folded to the next line. Effectively, a character entered at location (lineX,destWidth+1) is placed as (lineX+1,1). Whenever a carriage-return (<cr>) is entered, character positions in the current line are left empty, from where the cursor was located to the last character position (i.e. destWidth), and a new line is created to hold the text entered following the carriage-return.

It is also necessary to introduce the notation of the insertion point (sometimes called the cursor) where a new text may be inserted. An insertion point appears as a blinking vertical bar between characters (in the Macintosh environment). The symbol is simply "cursor". The cursor position is denoted as :
Loc(cursor) = (lineNo, charNo)
where $1 \le$ charNo $\le$ destWidth, is the position of the character to the right of the cursor. The cursor itself does not occupy any character position. It is placed between two characters.

8.4.1 Set Insertion point

One of the most frequently used interactions in an editing GUI is to use the marker to set the insertion point at a certain location. First move the marker to a location (lineNo, charNo), then 'click' the mouse button, as specified in F_set_insertion_pt.

```
-----     Specification_for_function   F_set_insertion_pt(lineNo,charNo) :
|
| Variables :          lineNo, charNo : integer , where
|                      1 ≤ lineNo ≤ destLen    and    1 ≤ charNo ≤ destWidth.
|
| From_state :         post_insert   or   post_select
| F_state_predicate:   true
|
| Inputs :             Loc(mk?) = (lineNo, charNo)   Tand   mb?=<click>
|
| To_state :           post_insert
| T_state_predicate:   Loc(cursor') = (lineNo, charNo)    and
|                      text(destRect')=text(destRect((1,1),(line,char-1)))  //
|                       cursor'  //  text(destRect((line,char),(destLen,destWidth)))
|                      and  text(record')=text(destRect((1,1),(lineNo,charNo-1)))  //
|                       <insPt>  //  text(destRect((lineNo,charNo),(destLen,destWidth)))
|______________________________________________________________________
```

Explanations :

- ThinkEdit has a much larger number of interaction functions than Logon. ThinkEdit functions are given meaningful names (e.g. F_set_insertion_pt) rather than numeric identifications (e.g. F4) as used in the specifications for Logon.

- As can be seen in the above specification for F_set_insertion_pt, the actual WinSpec statements are presented inside a rectangle bordered by a vertical line on the left and a horizontal line at the bottom. This is different from the layout of the Logon specifications (section 5.5). The difference is purely cosmetic so that lines of specification belonging to the same function appear to reside together within a rectangle.

- Variables are introduced in order to model some of the internal states of ThinkEdit. Variables have not been used in the specifications of the Logon interface. This is because all Logon functions are adequately modelled with the display objects and their states.

- The ThinkEdit GUI performs very little communication with its underlying application program. For this reason, the "Output_msg" clause is only included in specifications when it is necessary to do so.

- The cursor appears as a blinking vertical bar between characters. It is also called the insertion point where a new text may be inserted within the editing rectangle.

- The notation cursor' is used to represent the new cursor after the execution of the function.

• "Loc(mk?) = (lineNo, charNo)" denotes an input with the mouse pointer (or marker) which is moved to the location (lineNo, charNo).

• "Tand" is the notation for "temporal logical and" as defined in Chapter 5.
• "mb?=<click>" denotes an input of a mouse button click .

• As a consequence of F_set_insertion_pt, destRect' is displayed with cursor' at (lineNo, charNo).

• A temporary copy of the file being edited text(record) is also updated to text(record') in which the insertion point, denoted as <insPt> , is positioned accordingly.

• An instance of the execution of the F_set_insertion_pt function, as used in a test case, is denoted by substituting integer values for "lineNo" and "charNo" , such as :
F_set_insertion_pt(3,1)
which will move the blinking insertion point onto the 3rd line at a position just before the first character.

8.4.2 Insert text

Once an insertion point is moved to or set at a desired location, a text can be inserted through keyboard inputs (kb?).

```
-----   Specification_for_function    F_insert_text(aString):
|
| Variable :             aString : string
|                        line, char : integer , where
|                        1 ≤ line ≤ destLen    and    1 ≤ char ≤ destWidth.
|
| From_state :           post_insert
| F_state_predicate:     Loc(cursor)=(line,  char)
|
| Inputs :               kb?=aString  and   aString≠<cr>
|
| To_state :             post_insert
| T_state_predicate:     text(destRect')=text(destRect((1,1),(line,char-1)))
|                              // aString // cursor'
|                              // text(destRect((line,  char),(destLen,destWidth)))
|                        text(record')=text(destRect')
|                        and  T_state_predicate(F_update)
|________________________________________________________________________
```

Explanations :

- Initially the cursor is located at (line, char) within the existing text. A non-empty keyboard input is denoted by kb?=aString, where aString≠<cr>.

- An arbitrary piece of text is represented by specifying the begin and end points. The coordinates of points being used are of the form (line, char). For example, text(destRect((3,1),(5,30))) represents a section of text from the beginning (char=1) of the third line to somewhere in the middle (char=30) of the 5th line. This is a piece of text stretching about two and a half lines.

  The entire text in the destRect is expressed in coordinates as text(destRect((1,1), (destLen, destWidth))) where (1,1) represents the position of the first character on the first line, and (destLen,destWidth) represents the last character on the last line.

- The outcome (in T_state_predicate) is that the new keyboard inputs are inserted in the text at the point where the cursor was (i.e. (line, char)).

- For text formatting reasons, the input kb?=<cr> is treated as a separate function in F_insert_cr. This can be seen in the first line of the T_state_predicate.

- The symbol // indicates text concatenation.

- The cursor is moved to its new position following the text insertion. It is moved to the position next to the last character inserted.

- A temporary copy of the file being edited, record' , is updated according to the text in destRect'.

- An example of inserting a piece of text such as "This is new" at the beginning of the 3rd line will be denoted as :
  F_set_insertion_pt(3,1) o F_insert_text("This is new")

- "T_state_predicate(F_update)" is an abbreviation of : all the predicates in the T_state_predicate of function F_update are included here. F_update can be looked at as an internal housekeeping function, and is specified below.

```
-----     Specification_for_function     F_update :
|
| Variables :            dirtFlag : boolean
|
| From_state :           post_insert
| F_state_predicate :  F_state_predicate(F_insert_text or F_insert_cr or F_cut
|                                   or F_paste or F_paste_replace or F_clear)
|
| To_state :             post_insert
| T_state_predicate:   dirtFlag'=true
|                          and is_enabled(mOpt_'Save')
|______________________________________________________
```

Explanations :

- "F_state_predicate(insert_text)" is shorthand for : all the predicates in the F_state_predicate of function F_insert_text are included here.

- It is stated that any of the F_state_predicate of the 'updating' functions, F_insert_text, F_insert_cr, F_cut, F_paste, F_paste_replace, F_clear , can satisfy the F_state_predicate required for function F_update. These are functions that change the content of text being edited, in contrast to other functions that only affect the display of the text. (Specifications for these functions will appear in the following sections.)

- Function F_update is designed so that its T_state_predicate can be used in any of the 'updating' functions, to save repetition.

- As a kind of housekeeping, after the execution of any of the 'updating' functions, the dirtFlag is set. This is to indicate that a temporary copy of the file being edited is not the same as the copy on disk, text(record) ≠ text(file). (See section 8.3.) Consequently the menu option 'Save' should be enabled. This will allow the user to save the updates to the disk file.

```
-----    Specification_for_function    F_insert_cr:
|
| Variables :           line, char :  integer , where
|                       1 ≤ line ≤ destLen    and    1 ≤ char ≤ destWidth.
|
| From_state :          post_insert
| F_state_predicate:    Loc(cursor)=(line, char)
|
| Inputs :              kb?=<cr>
|
| To_state :            post_insert
| T_state_predicate:    text(destRect'((1,1), (line, char-1))) =
|                               text(destRect((1,1), (line, char-1)))
|                       and text(destRect'((line, char), (line,destWidth))) = <cr>//{}
|                       and text(destRect'((line+1, 1), (destLen', destWidth))) =
|                           cursor' // text(destRect((line, char), (line,destWidth)))
|                           // text(destRect((line+1, 1), (destLen, destWidth)))
|                       and Loc(cursor')=(line + 1, 1)
|                       text(record')=text(destRect((1,1), (line,char-1)))
|                               // <cr> // <insPt>
|                           // text(destRect((line, char), (destLen,destWidth)))
|                       and T_state_predicate(F_update)
|________________________________________________________________________
```

Explanations :

- F_insert_cr can be seen as a special case of F_insert_text, where the keyboard input is a carriage-return.

- The T_state_predicates give the changes in content in terms of text(record'), and the changes in visual appearance in terms of text(destRect').

- The content is changed by the insertion of a <cr> code at the cursor location where an input of kb?=<cr> is generated.

- The visual change is that a split of the text occurs at location destRect(line, char). The text from that location to the end of that line will form a new line of text.

- The cursor is moved to the beginning of the new line, Loc(cursor')=(line+1,1).

- Again <insPt> denotes the insertion point (or cursor) in text(record').

8.4.3 Text scrolling

Automatic scrolling takes place when there is a keyboard input at the first line or the last line in viewRect . Generally, the line where new keyboard inputs are being inserted is scrolled to the middle of the viewRect. It is called function F_insert_scroll in the following specification.

```
-----  Specification_for_function   F_insert_scroll:
|
| Variables:            char : integer, where  1 ≤ char ≤ destWidth
|                       offset : integer,
|                       where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state :          post_insert
| F_state_predicate:    Loc(cursor)=(offset+1,  char)
|                          or  Loc(cursor)=(offset+viewLen,  char)
|
| Inputs:               kb?=aString   and   aString≠<cr>
|
| To_state :            post_insert
| T_state_predicate:    if  Loc(cursor)=(offset+1,  char)
|                          then   offset' = offset - viewLen div 2
|                          else   offset' = offset + viewLen div 2
|                       and  T_state_predicate(F_insert_text)
|_______________________________________________________________
```

<u>Explanations :</u>

- The viewing rectangle (viewRect) contains a total number of lines represented by an integer viewLen. Each line consists of a number of character positions. The maximum number of characters allowed on each line is represented by an integer, viewWidth. It is assumed that viewWidth=destWidth. Typical values are :
  viewLen = 20 (20 lines in viewRect, from 1 to 20)
  viewWidth = 60 (60 characters along each line in viewRect, from 1 to 60)

- Recall that a viewRect is like a window exposing only part of the destRect. (See Figure 8.2.) The value of the integer variable "offset" represents the offset between the first line in destRect and the first line in viewRect. Section 8.3.1 shows that the first line of text in destRect is denoted as :
  text(destRect((1,1), (1,destWidth))

  Using the same coordinate system, the first line and the last line of text in viewRect are denoted as :
  text(viewRect((offset+1,1), (offset+1,viewWidth)))
  text(viewRect((offset+viewLen,1), (offset+viewLen,viewWidth)))

- When there is a keyboard input at the first line or the last line in viewRect, automatic scrolling takes place. Generally, the line where new keyboard inputs are being inserted is scrolled to the middle of the viewRect. The new offset between the destRect and the viewRect is decreased or increased by half the number of lines that the viewRect can hold.

- The notation T_state_predicate(F_insert_text) means that the T_state_predicate of F_insert_text is included here. (See section 8.4.2. for the specification of F_insert_text.)

- There is another form of automatic scrolling called F_scroll_select, that is carried out when a piece of text is being selected (highlighted) by mouse interactions. The specification for F_scroll_select will be given following the next section on F_select_text.

8.4.4 Select text

A user selects a piece of text by producing pointer and button inputs with the mouse. When a mouse-down event occurs inside the text rectangle (i.e. the view rectangle), mouse tracking keeps control until the mouse button is released. The text is highlighted as the mouse is dragged through it. When the user finalizes the selection by releasing the mouse button, the selected text remains highlighted and is ready for further manipulation, such as 'cut and paste'.

A selection range is a piece of text selected for editing operations such as delete, cut or copy. The selection range is defined by two variables of coordinate points: selStart and selEnd. They denote character positions at points between characters, not the characters themselves. The text between the two character positions is the selection, and appears highlighted when displayed on the screen. For example, selStart = (1,2) and selEnd = (3, 4) implies that the text from the 2nd character on the first line (lineNo=1, charNo=2) to the 3rd character on the 3rd line (lineNo=3, charNo=4-1) is selected inclusively. Note that the 4th character (charNo=4) is not included in the selection. This is because selStart and selEnd denote character positions at points between characters, not the characters themselves. A left to right convention is used; the location of charNo=4 is the point to the left of the 4th character (as the user sees), lying between the 3rd and the 4th character.

The left to right and top-down convention is assumed to be the normal way of selecting a piece of text. If a user has used a right to left or bottom-up movement instead, the values of selStart and selEnd are simply swapped to fit in with the existing coordinate system. This can be seen in the following specification for F_select_text.

A zero-length selection (selStart = selEnd) is called an insertion point (or cursor), as described earlier. Conversely :

Loc(cursor) = (line, char) => selStart = selEnd = (line, char)

When a new, empty file is opened for editing, both selStart and selEnd are initialized to (1,1). The interaction to make a text selection is F_select_text.

```
--- Specification_for_function   F_select_text((line1,char1),(line2,char2)):
|
| Variables :           line1, char1, line2, char2 :  integer, where
|                       offset+1≤ line1≤ viewLen, offset+1≤ line2 ≤ viewLen ,
|                       1≤ char1≤ viewWidth, 1≤ char2 ≤ viewWidth.
|
| From_state:           post_insert or post_select
| F_state_predicate :   True
| Inputs:               Loc(mk?)=(line1, char1)  Tand  mb?=<down>
|                       Tand  Loc(mk?)=(line2, char2)  Tand  mb?=<up>
|
| To_state:             post_select
| T_state_predicate:    if  (line2*viewWidth + char2) > (line1*viewWidth + char1)
|                         then  selStart'=(line1, char1) and  selEnd'=(line2, char2)
|                         else  selStart'=(line2, char2) and  selEnd'=(line1, char1)
|                       and is_hiLit( destRect'(selStart', selEnd') )
|                       and text(record')=text(destRect')
|___________________________________________________________________
```

<u>Explanations:</u>

- The variables line1, line2, char1 and char2 are used to represent character positions within the viewRect. The viewRect is considered instead of the destRect, because F_select_text is for selecting text within the viewRect. Another function, F_scroll_select, is specified in section 8.4.5 for selecting text outside the viewRect, in which case text scrolling may take place.

- selStart' and selEnd' are notations used to represent the values of selStart and selEnd after the execution of F_select_text. Their values before the execution are immaterial, as they are not required in the F_state_predicate.

- As mentioned earlier, the values of selStart and selEnd may have to be swapped if the user does not follow the "left to right" and top-down convention.

- “is_hiLit” is one of the state primitives defined in section 5.4. The statement is_hiLit(text(selStart', selEnd')) is a predicate stating that the piece of text located between (selStart', selEnd') is highlighted.

8.4.5 Scrolling while selecting text

Whilst making a text selection, if the user drags the mouse pointer outside the viewRect without releasing the button, the window's contents are scrolled continuously to keep the extending end of the selection in view. This is performed by checking the position of the mouse. If it is outside the viewRect, the text is scrolled one line at a time, as long as the button is held down outside the viewRect. The discussion of horizontal scroll is omitted by assuming viewWidth=destWidth.

```
-----   Specification_for_function   F_scroll_select
|                                      ((line1, char1), (line2, char2)) :
|
| Variables :          line1, char1, line2, char2 : integers,
|                      where  1 ≤ line1  ≤ destLen, 1 ≤ line2  ≤ destLen,
|                             1 ≤ char1  ≤ destWidth, 1 ≤ char2  ≤ destWidth
|                      offset : integer,
|                      where  0  ≤  offset  ≤  (destLen  -  viewLen)
|
| From_state:          post_insert or post_select
| F_state_predicate :  True
|
| Inputs:              Loc(mk?)=(line1, char1)  Tand  mb?=<down>
|                      Tand  Loc(mk?)=(line2, char2)  Tand  mb?=<up>
|                      and (  line2 > (offset+viewLen)  or  line2 < offset  )
|
| To_state:            post_select
| T_state_predicate:   T_state_predicate(F_select_text)
|                      and ( if line2 > line1 then offset' = line2 - viewLen
|                                             else  offset' = line2 - 1 )
|______________________________________________________________________________
```

<u>Explanations :</u>

- destLen and viewLen are the length (in terms of the number of lines) of the destRect and viewRect respectively, as defined earlier in table 8.1.

- line1 and line2 are line numbers with respect to the destRect.

- F_scroll_select will take place instead of F_select_text only if the mouse pointer is moved to a value of line2, where it is below the lower border of viewRect :
  line2 > (offset+viewLen) or
  it is above the upper border of viewRect : line2 < offset
  These conditions are stated in the F_state_predicate.

- When the mouse button is finally released, the new value of the offset is calculated as shown in the T_state_predicate. If the mouse pointer has been moved downwards (i.e. line2 > line1), line2 will become the last line in the viewRect. Consequently, the new value of offset is adjusted so that offset' = line2 - viewLen.

Alternatively, the mouse pointer could have been moved upwards from line1 to line2, where line2 < line1. In this case, line2 will become the first line in the viewRect, resulting in offset' = line2 - 1, as the first line in destRect is always line no. 1.

- Although the viewRect may be too small to display the whole of the selected text, the selected text is highlighted within the destRect :
is_hiLit(text(destRect((line1, char1), (line2, char2))))

- As the text selection is larger than the viewRect, only the last portion of the text selection is displayed (in the viewRect). It displays the bottom portion of the text selection if the mouse pointer is moved downwards to make a selection. Alternatively, the top portion of the text selection is displayed if the mouse pointer is moved upwards in making a selection. If the user wishes to view a portion of the text selection other than what is displayed, one of the scroll bar interaction functions can be used. (See Chapter 9 and Appendix C.)

8.4.6 A brief contrast between specification and code

A number of interaction functions have so far been specified. It is felt necessary to perform some intermediate review and evaluation. One way of examining what the specification process is achieving is to compare the specification with the program code. For this purpose, the main parts of the GUI code responsible for the F_select_text function are given below:

```
{---------------------------------------------------------------------}
{ HandleContent is the top level dispatching routine for mouse-downs in the text or }
{ scroll bars of the current front window. Note that mouse-downs in the text of      }
{ non-Edit windows are ignored.                                                     }
{---------------------------------------------------------------------}

procedure HandleContent;
var
    part: integer;
    control: ControlHandle;
begin
    GlobalToLocal(Event.where);
    part := FindControl(Event.where, TheWindow, control);
    if part <> 0 then
        ScrollContent(control, part, Event.where)
        else if EditText then
            with TheWInfo^^ do
                if PtInRect(Event.where, teh^^.viewRect) then
                begin
                    with Event do
                        TEClick(where, BitAnd(modifiers, ShiftKey)=ShiftKey, teh);
                    FixEditMenu
                end
            else
            SysBeep(5)
    end;
```

The source listing of three procedures, HandleContent, AutoScroll and HandleScroll, is examined. The HandleContent procedure attempts to find out where the mouse-down event was generated. If it was within the control regions of the scroll bar, procedure ScrollContent is called. If the mouse-down event was within the text of the viewRect, which is of interest to this discussion, it dispatches the mouse location to the system library TEClick. TEClick tracks the mouse movement with text highlights and calls procedure AutoScroll repeatedly, for as long as the user holds down the mouse button. Procedure AutoScroll is listed in the following page.

```
{-----------------------------------------------------------}
{ AutoScroll is the "ClikLoop" routine which is described in the TextEdit Programmer's}
{ Guide in Inside Macintosh. It is installed in all Edit windows opened by the editor. It is}
{ called repeatedly from the ToolBox, for as long as the user holds down the mouse button}
{ when the mouse was first clicked in the text of some window. The text in the window is}
{ scrolled up or down depending upon whether the mouse is above or below the text.}
{-----------------------------------------------------------}

function AutoScroll; { : boolean }
var
   mouse: Point;
   oldClip: RgnHandle;
begin
   AutoScroll := TRUE;
   oldClip := NewRgn;
   GetClip(oldClip);
   ClipRect(TheWindow^.portRect);
   GetMouse(mouse);
   with TheWInfo^^ do
      if mouse.v < teh^^.viewRect.top then
         HandleScroll(vScrollBar, InUpButton)
         else if mouse.v > teh^^.viewRect.bottom then
            HandleScroll(vScrollBar, InDownButton);
   SetClip(oldClip);
   DisposeRgn(oldClip)
end;
```

Procedure AutoScroll checks whether the mouse is above or below the upper or lower border of viewRect, and calls procedure HandleScroll to calculate the changes in the offset. (Source listing of procedure HandleScroll is given on the next page.) Eventually procedure HandleScroll calls SetCtValue (a system library) to adjust the scroll bar. HandleScroll also calls procedure AdjText, which in turn calls TEScroll (another system library) to scroll the text.

From the above program listings, it can be seen that the code implementing a certain interaction function can be scattered in different routines. In contrast, the state predicates in specifications give an integrated picture as all causes and effects are collected in one place. The WinSpec notations can describe user interactions (causes) and visible outcomes (effects) of functions in a comprehensive manner. When examining the program code, often no information is given about the processing taking place, other than what can be gleaned from the names of the routines, such as TEClick and SetCtlValue. The actual effect of these routines can only be understood by studying the window library programmer's manual. (In this case, TEClick and SetCtlValue can be found in Inside Macintosh vol.1 [Apple85].) These details of a program's internal working are unnecessary in the specification. Instead, the WinSpec notations can effectively expose vital results and visible outcomes, such as is_hiLit(text(destRect(selStart',selEnd'))).

```
{---------------------------------------------------------}
{ HandleScroll handles mouse-downs in the up-line, down-line, up-page, or down-page}
{ regions of the horizontal or vertical scroll bar. We must first determine which scroll }
{ bar the mouse was clicked in so that the proper page size is used. The value of the }
{ associated control is then updated and the text is scrolled if necessary. This procedure}
{ is the same as the actionProc "MyAction" which is described in the Control Manager}
{ section of "Inside Macintosh".      }
{---------------------------------------------------------}

procedure HandleScroll (scrollBar: ControlHandle; part: integer);
var
   delta, pageSize: integer;
begin
   if part <> 0 then
      with TheWInfo^^ do
      begin
         if scrollBar = vScrollBar then
            pageSize := textLines - 1
         else
            pageSize := textWidth div 2;
      case part of
         InUpButton:
            delta := -1;
         InDownButton:
            delta := +1;
         InPageUp:
            delta := -pageSize;
         InPageDown:
            delta := +pageSize;
         otherwise
      end;
   SetCtlValue(scrollBar, GetCtlValue(scrollBar) + delta);
   AdjText(TheWInfo)
  end
end;
```

Another benefit of a functional specification in a notation like WinSpec is that it is more likely to be reusable (at least in parts) for specifying GUIs on different implementation platforms. This is because specifications can model the main effects of functions on a level of abstraction that is higher than, and independent of implementations. It does not depend on routine names that are specific to a window system library, or notations that are restricted by the syntax of an implementation language such as C or Pascal. Although it is not possible to translate WinSpec specifications directly into an implementation, it is useful as an implementation-independent test specification. This section has justified the motivation for a functional specification. The process of specifying ThinkEdit functions is continued in the following sections.

8.4.7 Copy selection

The “Edit” menu option "Copy" allows the text selection to be copied to a scrap (called the clipboard), without deleting the selection from the viewRect. "Copy" doesn't change the existing text in the editing window. The concept of a clipboard, a special text window, is introduced to help to specify the functions of copy, cut and paste.

```
-----   Specification_for_function   F_copy :
|
| Variables :             selStart, selEnd : points of (line, char)
|
| From_state:             post_select
| F_state_predicate :  T_state_predicate(F_select_text   or   F_scroll_select)
|
| Inputs:                 kb?=<cmd-C>    or
|                         (is_inside(mp?,menu_'Edit')    Tand   mb?=<down>
|                         Tand   is_inside(mp?,mOpt_'Copy')   Tand   mb?=<up>)
|
| To_state:               post_select
| T_state_predicate:   text(clipboard') = text(destRect(selStart, selEnd))
|_______________________________________________________________
```

Explanations:

- A text selection range is defined by two variable sets of coordinate points: selStart and selEnd, as defined earlier in section 8.4.4.

- As F_state_predicate for F_copy, a piece of text selection must exist, as consequences of either F_select_text or F_scroll_select :
  T_state_predicate(F_select_text) or T_state_predicate(F_scroll_select)

- The "Inputs" required to execute F_copy are the mouse pointer and button interaction to choose the “Copy” menu option :
  Loc(mp?)=(mOpt_'copy') Tand mb?=<up>

- <cmd-C> is the command key for mOpt_'Copy', see explanations at 8.4.10.

- After the execution of F_copy, the content of the clipboard is set to the selected text between the two points selStart and selEnd :
  text(clipboard') = text(destRect(selStart, selEnd))

8.4.8 Cut selection

The "Edit" menu option "Cut" deletes the text selection from the display and makes a copy of it onto the clipboard. As an outcome of F_cut, the text characters located before and after the previously selected (highlighted) text piece are joined adjacently.

```
-----   Specification_for_function  F_cut :
|
| Variables :            selStart, selEnd : points of (line, char)
|
| From_state:            post_select
| F_state_predicate :    T_state_predicate(F_select_text  or  F_scroll_select)
|
| Inputs:                kb?=<cmd-X>   or
|                        (is_inside(mp?,menu_'Edit')  Tand  mb?=<down>
|                        Tand  is_inside(mp?,mOpt_'Cut')  Tand  mb?=<up>)
|
| To_state:              post_insert
| T_state_predicate:     text(clipboard') = text(destRect(selStart,selEnd))
|                        and  is_not_visible(text(destRect(selStart,selEnd)))
|                        and  text(destRect')=text(destRect((1,1),selStart))//
|                             cursor' // text(destRect(selEnd,(destLen,destWidth)))
|                        Tand (selEnd' = selStart)
|                        and  text(record')=text(destRect')
|                        and  T_state_predicate(F_update)
|_____________________________________________________________________
```

Explanations :

- Specification for F_cut is similar to that of F_copy, except that the text selection disappears, as stated in the T_state_predicate :
  is_not_visible(text(destRect(selStart, selEnd)))

- As the text selection between selStart and selEnd is removed, text(destRect') is formed by joining up the remaining text pieces.

- // denotes text concatenation.
- <cmd-X> is the command key for mOpt_'Cut', see explanations at 8.4.10.

- At the end of the execution of F_cut, the text selection between selStart and selEnd has vanished, and an empty selection is indicated by the predicate :
  selEnd' = selStart

- It is also necessary to perform some housekeeping tasks (e.g. enabling the 'Save' menu option), as the content of text(record) has been changed, by including the T_state_predicates of F_update.

8.4.9 Paste selection

The "Edit" menu option 'Paste' copies the text in the clipboard into the text being edited. There is only one clipboard, which is shared in common among all applications. This allows the user to cut or copy the text from one window and paste it down in another.

```
-----   Specification_for_function   F_paste :
|
| Variables :           line, char :  integer , where
|                       1 ≤ line ≤ destLen    and    1 ≤ char ≤ destWidth.
|
| From_state:           post_insert
| F_state_predicate :   Loc(cursor)=(line,  char)
|
| Inputs                kb?=<cmd-V>  or
|                       (is_inside(mp?,menu_'Edit')  Tand  mb?=<down>
|                       Tand  is_inside(mp?,mOpt_'Paste')  Tand  mb?=<up>)
|
| To_state:             post_insert
| T_state_predicate:    text(destRect')   =   text(destRect((1,1),  (line,char)))
|                              // text(clipboard)    // cursor'
|                              //  text(destRect((line,char),(destLen,destWidth)))
|                       and text(record')  =  text(destRect')
|                       and T_state_predicate(F_update)
|_______________________________________________________________
```

Explanations :

- Recall from section 8.4.4 that the cursor is formed as the result of an empty text selection (i.e. selStart = selEnd). It has a two way implication :
  selStart = selEnd = (line, char) => Loc(cursor) = (line, char)
  Loc(cursor) = (line, char) => selStart=selEnd=(line, char)

  Conversely, selStart ≠ selEnd => is_not_visible (cursor)
  The F_state_predicate for F_paste require the existence of the cursor (or insertion point) where the text will be pasted.

- <cmd-V> is the command key for mOpt_'Paste', see explanations at 8.4.10.

- The T_state_predicate describe how the content of text(destRect) is changed by concatenating the new text in the middle.

- The cursor is effectively placed after the newly pasted-in text.

8.4.10 Replace selection

F_paste_replace replaces the current text selection with the text in the clipboard. It can be looked at as a "cut" followed by a "paste", except the content of the clipboard remains unchanged throughout.

```
-----  Specification_for_function    F_paste_replace :
|
| From_state:          post_select
| F_state_predicate :  selStart ≠ selEnd
|
| Inputs               kb?=<cmd-V>    or
|                      (is_inside(mp?,menu_'Edit')  Tand  mb?=<down>
|                      Tand  is_inside(mp?,mOpt_'Paste')  Tand  mb?=<up>)
|
| To_state:            post_insert
| T_state_predicate:   is_not_visible(text(destRect(selStart,  selEnd)))
|                      and  text(destRect')=text(destRect((1,1),  selStart))
|                               // text(clipboard)   // cursor'
|                            // text(destRect(selEnd,  (destLen,  destWidth)))
|                      Tand (selEnd' = selStart  and  selStart'=selStart)
|                      and text(record')  =  text(destRect')
|                      Tand T_state_predicate(F_update)
|________________________________________________________________
```

<u>Explanations :</u>

- The text between selStart and selEnd is removed.

- The content of the clipboard is pasted in, as similar to F_paste. The position of the cursor is adjusted accordingly.

- Some functions that are usually invoked by selecting menu options can also be invoked through "keyboard accelerators". For example, the keyboard input, kb?=<cmd-V> can be used instead of
  is_inside(mp?, mOpt_'Paste') Tand mb?=<up> .

  The notation <cmd-V> is read as "command V". It denotes the entry of a "V" or "v" key on the keyboard, while the command key is held down.

  The suffix "_k" can be added to names of functions, to indicate that functions are invoked by means of command keys. For example, F_paste_k is used to refer to a F_paste function executed by using the keyboard accelerator <cmd-V>. This notation is used in the listing of test sequences in Chapter 9.

8.4.11 Clear selection

The "Edit" menu option 'Clear' deletes the selected characters from the display without copying them to the clipboard. As a consequence of F_clear, the text characters previously located before and after the text selection are joined adjacently.

```
-----   Specification_for_function  F_clear:
|
| Variables :           selStart, selEnd : points of (line, char)
|
| From_state:           post_select
| F_state_predicate :   T_state_predicate(F_select_text  or  F_scroll_select)
|
| Inputs:               is_inside(mp?, menu_'Edit')  Tand  mb?=<down>
|                       Tand  is_inside(mp?, mOpt_'Clear')  Tand  mb?=<up>
|
| To_state:             post_insert
| T_state_predicate:    is_not_visible(text(destRect(selStart,  selEnd)))
|                       and  text(destRect')=text(destRect((1,1),selStart))  //
|                            cursor'  //  text(destRect(selEnd,(destLen,destWidth)))
|                       Tand (selEnd' = selStart)
|                       and  text(record')=text(destRect')
|                       and  T_state_predicate(F_update)
|______________________________________________________________
```

<u>Explanations :</u>

- Specification for F_clear is similar to that of F_cut, except that the text selection is not saved to the clipboard.

8.4.12 Select All

The "Select All" option of the "Edit" menu allows a user to select or highlight the whole text being edited. It does not just select the text visible in the viewRect, but the entire text in the destRect.

```
----- Specification_for_function   F_selectAll :
|
| From_state:          post_insert or post_select
| F_state_predicate :  True
|
| Inputs:              kb?=<cmd-A>   or
|                      (is_inside(mp?,menu_'Edit')  Tand  mb?=<down>   Tand
|                       is_inside(mp?, mOpt_'Select All')  Tand  mb?=<up> )
|
| To_state:            post_select
| T_state_predicate:   selStart'=(1,1)  and
|                      selEnd'=(destLen, destWidth)
|                      and  is_hiLit(text(destRect'(selStart',selEnd')))
|                      and  text(record')=text(destRect')
|__________________________________________________________________
```

Explanations :

- <cmd-A> is the command key for mOpt_'Select All', see explanations at 8.4.10.

- selStart'=(1,1) indicates that text selection begins from the first character position of the first line.

- selEnd'=(destLen, destWidth) shows that the selection covers throughout the text including the last character position of the last line within destRect.

8.4.13 A review of edit-display functions

The specifications for the edit-display functions of ThinkEdit are now complete. It is vital to analyse what has been achieved. In essence, a total of 11 interaction functions have been identified and specified in this group, not including the internal functions F_update and F_insert_cr. The notation FG_edit_display is used to represent the group of edit-display functions specified.

FG_edit_display = { F_set_insertion_pt, F_insert_text, F_insert_scroll, F_select_text,
F_select_scroll, F_copy, F_cut, F_paste, F_paste_replace,
F_clear, F_selectAll }

A useful step to further our understanding of these interaction functions is to organize them into a structure to visualize how they are related. A simple state transition diagram is given below for this purpose.

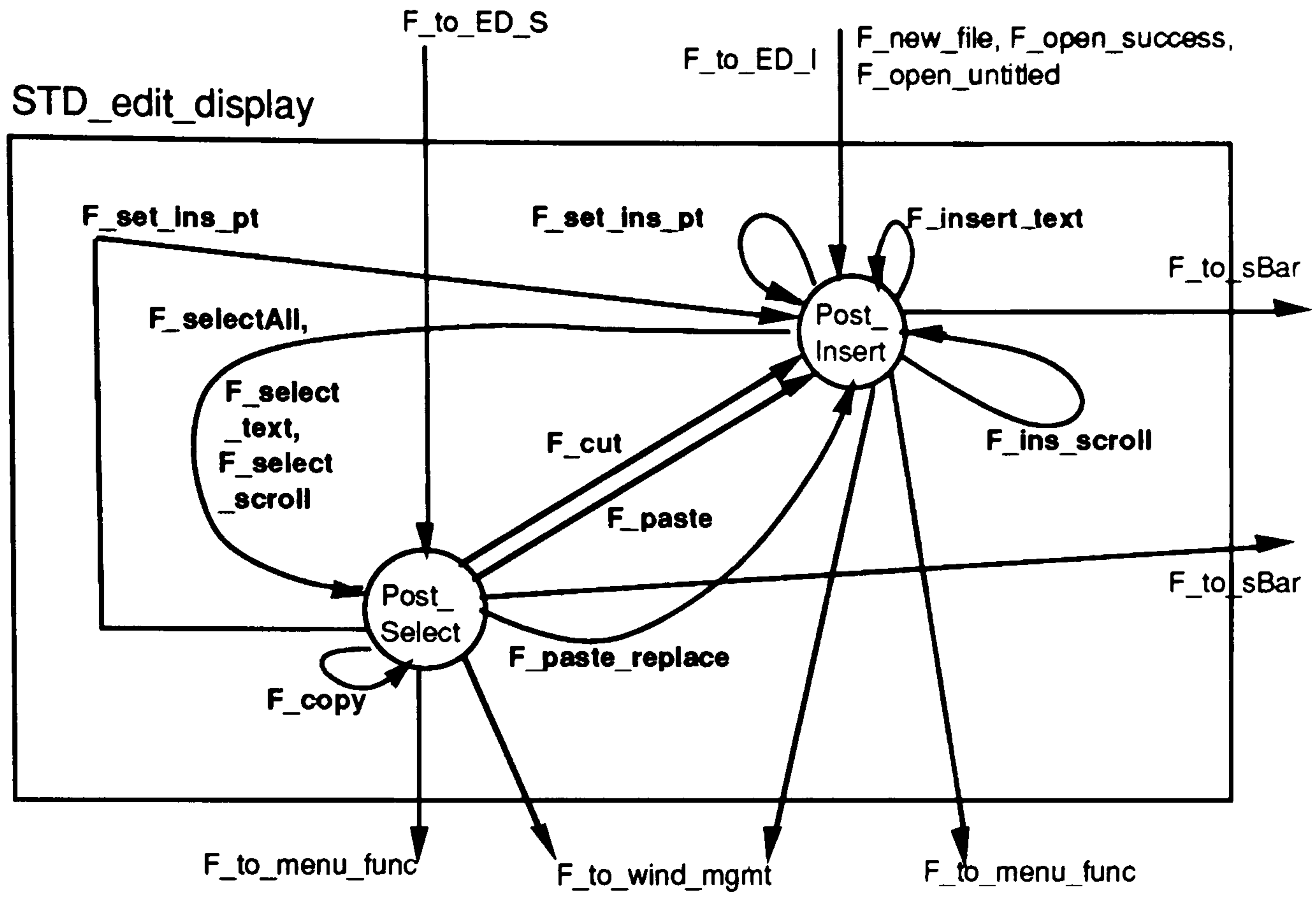


Figure 8.3 A STD showing the relationship amongst edit and display functions

In Figure 8.3, the rectangular box represents the border of the state diagram STD_edit_display, showing the state transitions within the function group FG_edit_display. The two circles (or nodes), Post_Insert and Post_Select, represent the two states between which transitions occur. For instance, the mouse interaction F_select_text causes the state transition from Post_Insert to Post_Select. This may be followed by a F_cut function, which returns the user interface to the Post_Insert state, where the insertion point (or cursor) is visible and no text selections (or highlights) can be seen. STD_edit_display is related to other function groups of ThinkEdit, as shown in the higher level state diagram in the next chapter. The lines and arrows penetrating in and out of the rectangular border of STD_edit_display, as shown in Figure 8.3, represent transitions from and to other state diagrams. (See also Figure 9.1, which includes other function groups to show the overall structure of ThinkEdit.)

8.5 Summary and directions

In this chapter, the specification of a window editor has been investigated. It began by presenting an informal description of ThinkEdit, followed by the detailed formal specifications. Display objects and interaction functions of ThinkEdit have been specified in terms of WinSTDs and WinSpec notations. A total of 11 edit-display functions have been specified, as listed in the last section (8.4.13).

Another main group of ThinkEdit functions is the menu functions. A total of 23 menu functions have been specified in appendix B. They are :

FG_menu_func = { F_new_file, F_open_file, F_open_cancel, F_open_select,
F_open_folder, F_open_open, F_open_success, F_open_fail,
F_clear_open_fail, F_DupFn, F_DupFn_cancel, F_open_untitled,
F_save, F_save_success, F_save_fail, F_clear_save_fail,
F_saveAs, F_saveAs_fn, F_quit, F_quit_warn, F_quit_cancel,
F_quit_save, F_quit_discard }

In addition to the above, there is also a group of scroll bar interaction functions:
FG_sBar = {F_sB_lineUp, F_sB_lineDn, F_sB_pageUp, F_sB_pageDn, F_sB_slider }.
Specification for these are given in Appendix C.

Windows and other display objects used in the ThinkEdit user interface are subject to a set of window management functions. Window management functions are generally provided by the underlying window system, rather than the application user interface. However, it has been found that window management functions are similar across different systems (see Chapter 3, [Myers89], [Yip91d]) and their specifications can be reusable. The specification of a small subset of window management functions, as applicable to ThinkEdit, is given in Appendix D.

FG_wind_mgmt = { F_select_wind, F_resize_wind, F_drag_wind, F_zoom_wind,
F_close_wind }

To summarize, there are four main function groups that have been identified and specified for ThinkEdit. They are the edit-display functions, menu functions, scroll bar functions and window management functions. An analysis of the relationship amongst these four groups of functions is presented in the beginning of the next chapter, leading to the generation of test cases for these functions.

Chapter 9

The testing of ThinkEdit

In the previous chapter, the functions of ThinkEdit were decomposed into four main groups : edit-display functions, menu functions, scroll bar functions, and window management functions. (See section 8.5)

FG_ThinkEdit = { FG_edit_display, FG_menu_func, FG_sBar, FG_wind_mgmt} .

Each function group is in turn made up of a number of interaction functions, as listed in section 8.5. The edit-display functions were specified in terms of WinSTDs and WinSpec notations in section 8.4. Specifications for the menu functions, scroll bar functions, and window management functions are given in Appendix B, C and D respectively. In this chapter, test cases of interaction sequences are derived from specifications to cover these four function groups.

Function groups do not exist in isolation. For instance, a menu function must be invoked to open a file before any of the edit-display functions can be tested. In the last chapter, STDs are used to illustrate the relationship amongst functions within a function group. A top level state diagram is given in Figure 9.1 to show the relationship between different function groups. The rectangular boxes represent individual STDs for the four function groups. The state diagram, STD_edit_display, has already been given in the last chapter (Figure 8.3). The internal details of the other state diagrams, as represented in Figure 9.1, are given in the Appendices.

In Figure 9.1, the transitions between different function groups (each represented as a STD within its rectangular borders), are achieved through mode-free interactions with the mouse device. For example, in the middle of text editing (i.e. in STD_edit_display) the user is free to move the mouse pointer into the scroll bar regions (i.e. F_to_sBar) to invoke functions within STD_sBar, or to invoke functions in STD_menu_func by the interaction to move the mouse pointer (F_to_menu_func) onto the menu bar.

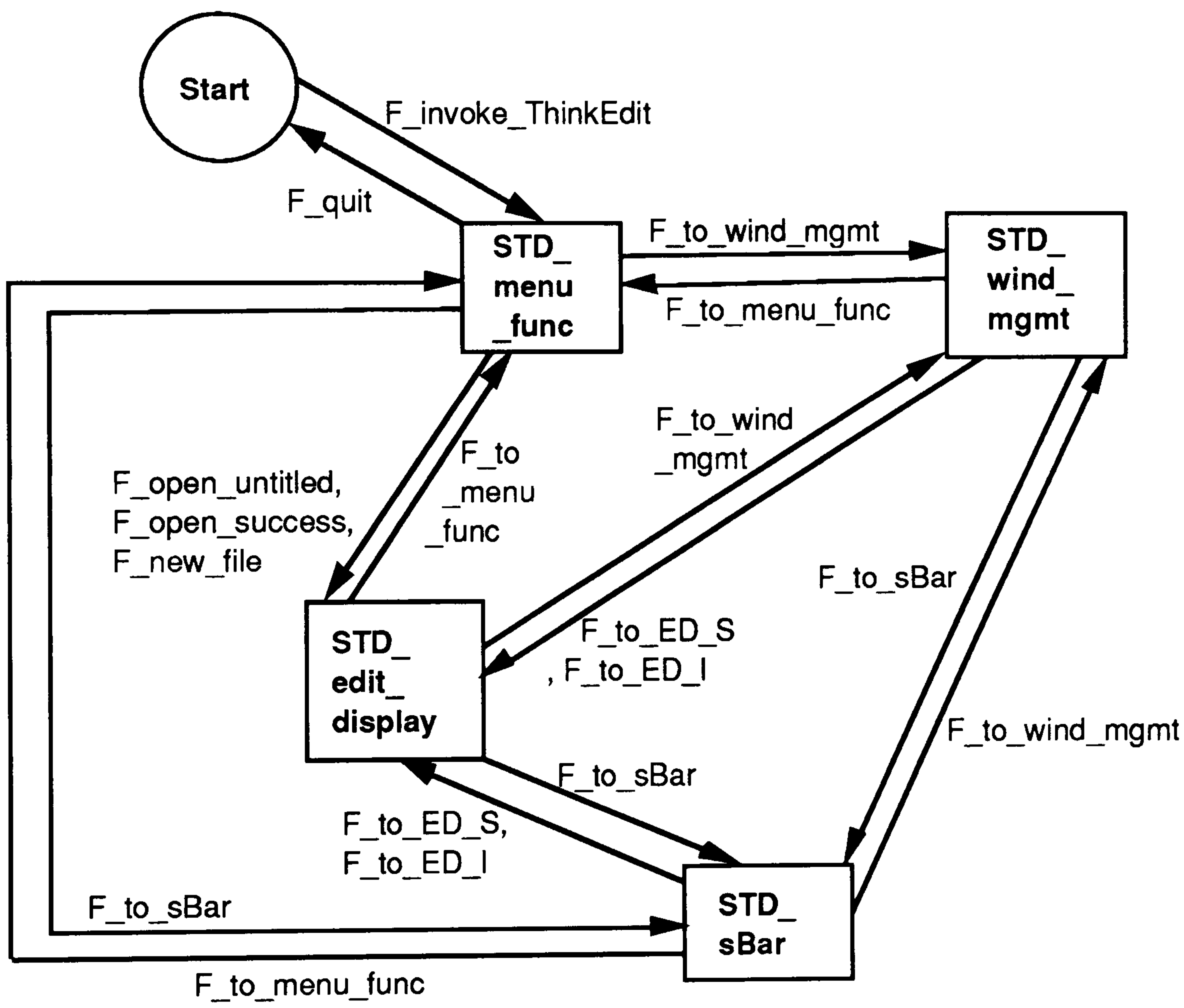


Figure 9.1 A state diagram showing overall control flow of interactions in ThinkEdit

9.1 Test selection criteria

In order to test a GUI, all display objects, interaction functions and application messages are identified. During the testing of the Logon interface in Chapter 7, four different test selection criteria were considered :

TC1 - 100% coverage of objects
TC2 - 100% coverage of messages
TC3 - 100% coverage of functions
TC4 - 100% coverage of all possible combinations of functions

The message coverage criterion is unsuitable on its own as very few application messages are used in the specification of ThinkEdit. This is because editing functions are largely user interactions with little application processing outside the user interface. Application messages are only used in a few functions mainly for opening, reading from and writing to disk files. TC2 is rejected as ThinkEdit cannot be adequately tested by checking application messages alone. In practice, TC2 is covered by TC3, as all applications messages are included in the specification of functions.

The object coverage criterion was also abandoned. The main reason, as found in Chapter 7, is that TC1 is not as powerful as TC3 in detecting errors. For ThinkEdit, it was found that each character in the text being edited can be recognized as a lower level display object within the viewRect (the viewing window). One can argue that the object viewRect is in a different state when text(viewRect((line,char),(line,char+1))) has a different character. This implies that a 100% object coverage may not be practical as the total number of test sequences can be quite large. As discussed earlier, destRect (the destination rectangle) can actually be thought of as a bottomless rectangle. Practically, only a small, representative number of character locations are selected for testing.

A 100% coverage of functions is the main strategy being used to test ThinkEdit, as the criterion of testing all possible combinations of functions (TC4) is impractical. Although TC3 is the chosen criterion for further experiments, it can be complemented if necessary by any of the other three criteria.

9.2 From specification to test sequences

Having decided on a 100% function coverage criterion, it is desirable to generate the smallest number of test cases which, together, will cover all functions. Each test case is the invocation of a sequence of interaction functions. Individual functions are joined together to form a sequence as permitted by their To_ and From_states. For example :

F_set_insertion_pt(8,1) o F_insert_text("This is line No. 8 ...", <cr>)

A basic algorithm for generating test sequences is outlined here. The first step is always to select the function that would invoke the user interface. In this case it is a double mouse click at the ThinkEdit icon (denoted as F_start or F_invoke). Then one of the functions with suitable From_states is selected to be the second function in the test sequence. Another function with From_state matching the To_state of the second function is selected to join the test sequence as the third function. This selection process carries on until all functions are covered at least once, or until the end of a sequence is reached by having chosen a function that will terminate the execution of ThinkEdit. It is possible that a sequence may be terminated before all functions are covered, in which case another test sequence has to be generated to cover the untested functions.

To reduce the cost of testing, ThinkEdit should be invoked the smallest number of times necessary to cover all functions. This is because the startup and termination of programs are usually time consuming. Therefore a test sequence should cover as many different functions as possible, even if some functions are repeated within the same invocation. The execution (or testing) of a certain interaction function is only repeated if it is strictly necessary. For example, a function is repeated in order to reach other yet untested functions, or if repeated execution can expose new errors. As discussed in Chapter 6, it is possible to employ existing techniques, such as RCPT, to assist the derivation of test sequences.

9.3 Test sequences generated

A total of four test cases (or sequences) have been generated, following the basic algorithm outlined above, to cover all functions of the four function groups identified earlier.

9.3.1 Test sequence for edit-display functions :

Test sequence TS1:

F_start
o F_open_cancel o F_new_file
o F_insert_text(“abcdefghijklmnopqrstuvwxyz1234567890-=[];',./”,<del>,<cr>)
o F_set_insertion_pt(1,1) o F_insert_text(“Line No. 1: ”)
o F_select_text((1,1), (2,1)) o F_cut o F_paste o F_paste_k o F_paste o F_paste_k
o F_select_text((1,1), (3,1)) o F_clear o F_paste_k o F_paste_k o F_paste_k
o F_select_text((6,1), (1,1)) o F_copy
o F_paste o F_paste_k o F_paste_k o F_paste_k o F_paste_k
o F_select_text((2,10), (2,11)) o F_insert_text(“2”)
o F_select_text((3,10), (3,11)) o F_insert_text(“3”)
o F_select_text((4,11), (4,10)) o F_insert_text(“4”)
o F_select_text((5,11), (5,10)) o F_insert_text(“5”)
o F_select_text((6,11), (6,10)) o F_insert_text(<del>,“6”)
o F_select_text((7,10), (7,11)) o F_insert_text(<del>,“7”)
o F_set_insertion_pt(8,11) o F_insert_text(<del>,“8”)
o F_set_insertion_pt(9,11) o F_insert_text(<del>,“9”)
o F_set_insertion_pt(10,11) o F_insert_text(<del>,“10”)
o F_select_text((11,1), (1,1)) o F_copy_k
o F_set_insertion_pt(11,1) o F_insert_text(<cr>,<cr>,<cr>,<cr>,<cr>)
o F_set_insertion_pt(destLen, destWidth +1)
o F_insert_text(“Trying function F_paste_replace”)
o F_select_text((destLen,1), (destLen, destWidth+1)) o F_paste_replace
o F_saveAs o F_saveAs_fn(“tf.1”) o F_save_success
o F_select_text((15,1), (20,1)) o F_cut_k o F_paste
o F_set_insertion_pt(destLen, destWidth+1) o F_insert_text(<cr>) o F_paste
o F_scroll_select((destLen, destWidth), (1,1)) o F_sB_slider(at_bottom)
o F_set_insertion_pt(destLen, destWidth+1)
o F_insert_scroll(<cr>, “This is the last line ...”)
o F_selectAll
o F_sB_slider(at_top) o F_set_insertion_pt(2,7)
o F_scroll_select((2,14), (30,5))
o F_selectAll_k
o F_quit o F_quit_warn o F_quit_save o F_save_success
end

An explanation of TS1

Test sequence TS1 is generated by invoking all the functions, where allowable. one after another. All the edit-display functions are covered in TS1, namely :

FG_edit_display = { F_set_insertion_pt, F_insert_text, F_insert_scroll, F_select_text,
F_selectAll, F_scroll_select, F_cut, F_paste, F_paste_replace,
F_copy, F_clear }

TS1 does not attempt to test different combinations of these functions. Most edit-display functions are mode-free, except F_copy, F_cut and F_paste_replace which must be preceded by F_select_text. Usually, F_set_insertion_pt is invoked before F_insert_text as a common pattern of text editing activities.

TS1 contains a number of combinations of F_select_text, F_cut, F_copy and F_paste. These are mainly employed to build up a reasonable length of text for testing the text scrolling functions. TS1 does not give special consideration to boundaries of words, lines and paragraphs as portrayed in one specification approach [Sufrin82]. A blank space in the text is just considered as another character. Characters are placed in a new line because they were preceded by a carriage-return character (denoted as <cr>). However, it is found that ThinkEdit does not fold lines automatically when the last character position (destWidth) of the destRect is reached. Keyboard inputs are allowed to go beyond the edge of the window, and are not visible. This is clearly a design / implementation error.

Observations of users show that an interaction usually starts as an initial long spell of F_insert_text, with the insertion point remaining at the end of the text. Then editing occurs at different parts of the text, through interactions that set insertion points, insert characters and remove unwanted characters. The test sequence TS1 covers a number of these interactions; it also includes cut-and-paste interactions that move text pieces around.

The part sequence " F_quit o F_quit_warn o F_quit_save o F_save_success o end" is used to terminate TS1. Different terminating functions are used in TS1, TS2, TS3 and TS4 to cover the four possible ways of terminating the execution of ThinkEdit : F_quit_save, F_quit_discard, F_close_save and F_close_discard.

9.3.2 Test sequence for menu functions

Test sequence TS2:

F_start
o F_open_file o F_open_folder o F_open_select("tf.1") o F_open_open
o F_open_success
o F_open_file_k o F_open_select("tf.1") o F_open_open
o F_DupFn o F_DupFn_cancel
o F_open_file o F_open_select("tf.1") o F_open_open
o F_DupFn o F_open_untitled o F_open_success
o F_select_text((1,1), (3, destWidth)) o F_cut
o F_save o F_saveAs_fn("tf.2") o F_saveAs o F_saveAs_fn("tf.2")
o F_save_fail o F_clear_save_fail
o F_open_file o F_open_select("tf.2") o F_open_fail o F_clear_open_fail
o F_new_file o F_new_file o F_new_file_k o F_new_file_k
o F_quit o F_quit_warn o F_quit_cancel
o F_quit_k o F_quit_warn o F_quit_discard o
end

An explanation of TS2

TS2 tests the menu options and dialogue boxes in ThinkEdit by opening, closing and saving files. The FG_menu_func functions are not as mode-free as the FG_edit_display functions. For instance, F_quit_warn must be preceded by F_quit, and F_quit_warn must be followed by one of three functions : F_quit_cancel, F_quit_discard or F_quit_save. This actually makes test case design more straightforward, as choices are limited by modes of interactions. Again, the task of test sequence generation is to select functions with matching To_state and From_state from the WinSpec specifications.

There are some exceptions to the simple pattern of joining functions with matching To_states and From_states to form a test sequence. For example, the function F_new_file is invoked repeatedly to open five editing windows to reach the maximum window limit of ThinkEdit.

Although TS2 includes the two part sequences,
F_save_fail o F_clear_save_fail
F_open_fail o F_clear_open_fail ,
the testing of read and write errors during file access requires the simulation of some hardware circumstances, such as a full disk or some disk drive problems.

A total of 23 menu functions have been specified in appendix B. They are :

FG_menu_func = { F_new_file, F_open_file, F_open_cancel, F_open_select,
F_open_folder, F_open_open, F_open_success, F_open_fail,
F_clear_open_fail, F_DupFn, F_DupFn_cancel, F_open_untitled,
F_save, F_save_success, F_save_fail, F_clear_save_fail,
F_saveAs, F_saveAs_fn, F_quit, F_quit_warn, F_quit_cancel,
F_quit_save, F_quit_discard }

All of these menu functions are included in TS2, except F_quit_save and F_save_success, which have already been covered by TS1.

9.3.3 Test sequence for scroll bar interactions

Test sequence TS3:

```
F_start
o F_open_file o  F_open_folder
o F_open_select("tf.1")
o F_sB_slider(at_bottom)                      ! Move slider to bottom of sBar,
o F_sB_lineUp  o  F_sB_lineDn(2)              ! see Appendix C for definitions
o F_set_insertion_pt((destLen+1, 1)           ! of "top", "middle" and "bottom".
o F_insert_text("The end")
o F_sB_pageUp  o  F_sB_lineDn(3)              ! Up 1 page and down 3 lines.
o F_sB_slider(at_top)                         ! Move slider to top of scroll bar.
o F_sB_lineUp(2)  o  F_sB_lineDn
o F_sB_slider(at_middle)                      ! middle = (bottom - top) div 2.
o F_set_insertion_pt(destLen div 2, 1)
o F_insert_text("Approx. in the middle", <cr>)
o F_select_text((destLen div 2, 1), (destLen div 2 +1, 1))
o F_sB_pageUp  o  F_sB_lineDn(2)
o F_sB_pageDn  o  F_sB_lineUp(3)
o F_sB_slider(at_bottom)                      ! Move slider to bottom of sBar.
o F_set_insertion_pt((destLen+1, 1)
o F_close o F_close_warn o F_close_save o
end
```

An explanation of TS3

The scroll bar interactions are completely mode-free. The user is free to drag the slider to any position within the scroll bar, or perform the page-up, page-down, line-up and line-down interactions in any order. However, some of these interactions are only meaningful or visually observable when there is more than one pageful of text. This is why the test inputs in TS1 are designed to ensure a sufficient amount of text is generated in file tf.1 .

Since the FG_sBar functions are all mode-free, it is unnecessary to search WinSpec specifications to look for functions with matching To_states and From_states. However, the "Inputs" clause and the T_state_predicates are useful for generation of test inputs and test oracles. The order of functions chosen in TS3 is designed for visibility. For example, interactions to move one page down F_sB_pageDn, followed by two lines up F_sB_lineUp(2). The idea of boundary value testing is used by first moving the slider to the top position of the scroll bar, then to the middle position, and then to the bottom position.

9.3.4 Test sequence for window management functions

Test sequence TS4:

```
F_start
o F_open_file  ("tf.1")
o F_drag_wind(wind_'tf.1',  (x1,  y1),  (x1+200,  y1+20))
o F_zoom_wind(wind_'tf.1')
o F_zoom_wind(wind_'tf.1')
o F_resize_wind(wind_'tf.1',  (x2,  y2),  (x2+100,  y2-50))
o F_zoom_wind(wind_'tf.1')
o F_zoom_wind(wind_'tf.1')   o   F_zomB_track(wind_'tf.1')
o F_open_file  o  F_open_select("tf.1")  o  F_open_untitled
o F_drag_wind(wind_'Untitled1',  (x3,  y3),  (x3-50,  y3+100))
o F_select_wind(wind_'tf.1')
o F_select_wind(wind_'Untitled1')
o F_new_file  o  F_close_wind(wind_'Untitled2')
o F_select_wind(wind_'Untitled1')  o    F_cloB_track(wind_'Untitled1')
o F_close_wind(wind_'Untitled1')
o F_insert_text ("Now tf.1 should become the active window as others are closed")
o F_close_wind(wind_'tf.1')  o  F_close_warn  o  F_close_discard
end
```

An explanation of TS4

The specification of window management functions can be found in Appendix D. The window management functions of ThinkEdit are mostly mode-free. Windows can be freely moved around, resized, zoomed or selected in any order. Therefore, TS4 is generated simply by including all the window management functions one after another. It is obvious that a window can no longer be manipulated after it has been closed. It is necessary to have more than one window to test the F_select_wind function. The F_zoom_wind function toggles the size of a window between two sets of values. One set of window size is fixed by the program. The other set of window size is adjustable by the user. This is why it is necessary to have two adjacent F_zoom_wind functions in the test sequence. In TS4, F_zoom_wind is tested again after F_resize_wind, which is the function that changes the user-adjustable set of window size. The size of a rectangular window is normally expressed in terms of the (x,y) coordinates of its top-left and bottom-right corners : ((left,top), (right,bottom)). The set of fixed window size for ThinkEdit is ((0,38), (513,342)), which is the zoomed window size. The user adjustable set of window size is initially set at ((3,40), (250,338)).

Regarding the "drag" and "resize" functions, the relative movement of the mouse pointer determines changes in the location and size of a window. The movement of the mouse pointer is relative to the location where the mouse button was pressed down. For example, F_drag_wind (wind_'tf.1', (x1,y1), (x1+200, y1+20)) will move the window 200 units towards the right and 20 units downwards.

There is one minor feature of the window management functions included in TS4. When the mouse pointer is inside the zoom box (or the close box) and the mouse button is pressed down, the mouse movement is tracked until the mouse button is released. If the mouse pointer is subsequently moved outside the zoom box (or close box) before the mouse button is released, the F_zoom_wind (or F_close_wind) functions are not performed. This interaction can be expressed in WinSpec notation as follows.

is_inside(mp?, wind_'tf.1'.zomB) Tand mb?=<down>
Tand is_not_inside(mp?, wind_'tf.1'.zomB) Tand mb?=<up>

As the source code of the window manager is not available for error injection, TS4 is not designed to uncover errors in these libraries. It is to include a basic level of sanity checks of window management functions.

9.4 Error seeding and debugging

As each function within a test sequence is executed, the outcomes are checked against the T_state_predicate for that function. Any discrepancies or unexpected outcomes are regarded as faults or symptoms. They indicate the possible existence of errors, either in the implementation or in the specification. During the main experiment, 40 errors were generated by the human tester to cover all the code modules of ThinkEdit. These 40 errors are listed in table 9.1 in section 9.5. Some minor experiments were also conducted, during which a smaller number of errors were injected in locations unknown to the tester. Experienced programmers outside the testing research group were invited to invent errors that must pass compilation and permit program execution in the first instance. It was found that errors generated by outsiders were more readily exposed than errors injected by a human tester. The human tester was able to invent "harder" errors because of testing experience and familiarity with ThinkEdit. For instance, errors in setting up proper parameter values in window library calls will usually produce obvious symptoms. Ordinary program logic errors, such as forgetting to set the dirtFlag in one location[6], are harder to expose. The error seeding process is shaped by a number of constraints:

• A program module must be re-compiled successfully after the injection of an error. Generally, program units of ThinkEdit are relatively small, with few variables declared in each unit. As references to undeclared variables will cause problems with the compiler, the invention of errors by "misuse of variables" is restricted.

• Pascal performs strong type and number checking of parameters in routine calls. This limits the introduction of errors in window library call statements.

• It was decided that the test object must run successfully initially. This narrows the possibility of injecting errors in program statements responsible for initializing system and window resources. Errors in these area will normally cause program crashes or exceptions as soon as ThinkEdit is invoked.

• As a principle, errors that would produce very obvious symptoms are not used. For example, it is simple to 'comment out' a line of code that displays a window. This would be easily spotted by the tester, as a window is expected to be displayed at that point of interaction, according to the T_state_predicate in the specification. A better or harder error is one that would display a window but with the "down arrow" missing from its scroll bar[7]. To invent such "hard to detect" errors usually requires some knowledge of the window system.

[6] See error E13 listed in the table of results in section 9.5.
[7] See error E2 listed in the table of results in section 9.5.

• Lastly, errors are to be scattered fairly randomly, but covering all program modules. The process of error seeding requires some understanding of the code structure and introduces a sense of code coverage. An example of error seeding is given below.

```
{-------------------------------------------------------------------------------------------}
{ CloseMyWindow is used to close an Edit window. The window is closed, and the }
{ associated data structures are Disposed of.                                  }
{-------------------------------------------------------------------------------------------}

procedure CloseMyWindow (window: WindowPtr);
    var
    wInfo: WInfoHandle;
    begin
    wInfo := WInfoHandle(GetWRefCon(window));
    CloseWindow(window);
    TEDispose(wInfo^^.teh);
    {E11, by commenting out "TEDispose(...)", no visual symptoms, it may
                                          eventually run out of memory}
    DisposHandle(Handle(wInfo))
    end;
```

It was found that after the injection of more than 4 or 5 errors, it was not always easy to associate symptoms with errors on a one to one basis. The approach taken was to fix (or debug) one symptom at a time. A retest (using the same test sequence) was then carried out to see if the symptom was cured, before moving on to fix another symptom. In this way, a code error was then claimed to have been uncovered by a certain function test within a test sequence. Only four main test sequences were derived following the 100% function coverage criterion. In practice, a test sequence was halted as soon as a symptom appeared. An effort was then made to locate the code error(s) responsible for the symptom. Since code errors were introduced artificially, it was most important to check if any seeded errors remained uncovered after all symptoms exposed by the test sequences were fixed. These accounted for the undetected errors in the table of results in the following section.

9.5 Results of testing

The results of testing, in terms of success or failure to detect the seeded errors, are given in the following table. A number is assigned to each of the errors injected, as listed in the first column of the table. The second column gives a brief description of the error, such as its symptom, if one is observable. The third column records if the particular error was detected during testing. The 4th (last) column attempts to give more details about the nature of the error, such as where it was located in the source code, and which functional test exposed the error. This information will be useful for future references when experimenting with new test sequences derived from enhanced specifications. To grasp the full details of these errors, one would require execution of the test sequences in conjunction with the source code listing.

To summarize, this chapter describes a testing experiment with ThinkEdit. It explains how formally specified functions are organized to generate test cases. These test cases (or sequences) are then evaluated by error seeding and debugging exercises. Of the 40 errors seeded, 8 errors remain undetected after the execution of the test sequences. An analysis of these undetected errors is given in Chapter 12, where code coverage measurements are used to further evaluate the FFT approach.

Error No.	Brief description (symptoms, etc)	Detected ? / by	More details (where in code, which function test exposed it)
E1	No vertical scroll with more than one page input	Yes TS1	lineHeight:=ascent+descent+leading detected by ... o F_select_text((11,1), (1,1)) o ... o F_paste
E2	sBar_dnArrow missing	Yes TS3	bottom - SBarWidth + 2 ... o F_sB_lineUp o F_sB_lineDn o ...
E3	r.right + r.left in OpenWindow	No	Does not seem to have any observable functional or visual difference.
E4	SetEnable(...) in OpenFile	No	Would only be exposed when the no. of files opened exceeds Maxfiles.
E5	No diaB to prompt fName to save Untitled	Yes TS2	fName := StandardFile(StandardPut, ...); in function SaveText, detected by F_SaveAS .
E6	mOpt_'Save' not disabled after Save	Yes TS2	DisableItem(FileMenu, SaveItem); in SaveText, Detected by checking Post-conditions after F_Save.
E7	Window title not updated after SaveAs	Yes TS2	ChangeFile(window, fName) ; in SaveText, detected by F_SaveAS.
E8	Cancel and Discard swapped	Yes TS2	Cancel = 3; in function SaveFile, Detected by F_quit_warn.
E9	No diaB to prompt user to save dirty at quit.	Yes TS2	if wInfo = nil ... in function SaveFile Detected by F_quit_warn.
E10	SaveAs dialogue box flashed away.	Yes TS2	ModalDialog(nil, item); in Savefile Detected by F_quit_warn.
E11	Forgot to dispose text record when closing	No	TEDispose(wInfo^^.teh); in CloseMyWindow No visual symptoms, may eventually run out of memory.

Error No.	Brief description (symptoms, etc)	Detected ? / by	More details (where in code, which function test exposed it)
E12	GUI hangs when mOpt_'Close' selected	Yes TS2	window := window^.nextWindow Detected by any quit functions.
E13	textDirty := FALSE;	No	Can only be detected by the sequence : F_start o F_select_text o F_cut o F_quit
E14	"vScrollBar" typed as "hScrollBar"	Yes TS3	in Procedure HandleScroll, Detected by F_sB_pageUp, F_sB_pageDn.
E15	"AdjText(TheWInfo)" missing	Yes TS3	in Procedure ScrollContent, easily detected by any F_sB_slider interactions, no text adjustment.
E16	"oldClip := NewRgn; " missing	Yes TS1	in function AutoScroll, detected by any mouse interaction straight away with "Address error".
E17	"SetClip(oldClip);" missing	Yes TS1	in function AutoScroll, detected easily by any F_select_text interaction.
E18	"if part = 0" as Typo in HandleContent	Yes TS1	Detected by any mouse down interaction with "address error".
E19	"scrollContent(...)" missing	Yes TS3	in Procedure HandleContent, detected by any scroll bar interaction.
E20	ShowSelect(TheWInfo) missing	No	in Procedure HandleKey. Undetected, no obvious functions.
E21u	no text in window at event of open	Yes TS2	"TEUpdate(teh^^.viewRect, teh);" is missing in HandleUpdate, detected by F_open_file .
E22u	Incorrect "duplicate Filename" warning .	No	fName,dName,Null swapped in "ParamText(...);" Detected by opening the same file the second time.

Error No.	Brief description (symptoms, etc)	Detected ? / by	More details (where in code, which function test exposed it)
E23u	incorrect UntitledNum in DuplicateName	No	"UntitledNum + 1" instead of -1 , can be detected by F_open_file o F_open_cancel o F_new_file
E24u	incorrect filetypes	Yes TS2	numTypes=-1,2 instead of 1 in SFGetFile(...), detected straight away with F_open_file.
E25u	No highlight of default command button	Yes TS2	"FrameRoundRect(iBox, 16, 16);" missing in FrameDItem , detected by F_quit_warn.
E26u	Forgot to reset PenSize , no visual effect	No	"SetPenState(oldPenState)" missing undetected, probably taken care by SetPort(...) .
E27u	Address error with DisablItem (WindowMenu, i)	Yes TS2	in unit Editor utilities, detected by : F_start o F_open_cancel o F_new_file o ... o F_close_file
E28g	"New" , "Close" swapped in Globals	Yes TS2	detected by : F_start o F_open_cancel o F_new_file o ...
E29	loss of highlight of zomB after drag_wind	Yes TS4	Forgot to "ValidRect(r)" on zoom Box, detected by F_zoom_wind o F_drag_wind
E30	horizontal scroll bar missing	Yes TS4	forgot to re-adjust "textLines" after resize, detected by F_resize_wind
E31	improper half or part char along right border	Yes TS4	"textWidth" not re-adjusted after resize, detected by F_resize_wind
E32	titleBar gone outside screen, behind the menuBar	Yes TS4	Forgot to adjust "LocalToGlobal" when returning from zoomed to normal, local coordinates start from 0 (i.e. location of menuBar in global co.). Detected by F_zoom_wind o F_zoom_wind

Error No.	Brief description (symptoms, etc)	Detected ? / by	More details (where in code, which function test exposed it)
E33	window zooms with single click at titleBar	Yes TS4	"<" is used instead of ">" by mistake in "if TickCount < (FirstClick + GetDblTime)"
E34	window does not zoom	Yes TS4	Code error in "FirstClick := TickCount;", both E33 and E34 detected by F_zoom_wind.
E35	no mouse tracking in close Box,	Yes TS4	detected by F_cloB_track.
E36	window won't be resized,	Yes TS4	missing call to HandleGrow(), detected by F_resize_wind.
E37s	vScrollBar not to scale with text length, error	Yes TS3	in "ctlMax := LinesInText(wInfo) - textLines;" detected by F_sB_slider(at_bottom), more obvious with a longer text.
E38s	hScrollBar has no effect, but vScrollBar	Yes TS3	affects both horizontal and vertical scrolling , "delta := oldScroll - newScroll;" missing.
E39s	no scrolling of ins_pt to middle of viewRect	Yes TS3	Omitted scroll bar adjustment, detected by inserting text on last line including kb?=<cr>
E40s	ins_pt moved to mid. of viewRect wrongly	Yes TS3	Error in "bottomLine := topLine+ textLines;" , detected by F_insert_text above the bottom line.

u* in program source file: "Editor Utilities"

g* in program source file: "Editor Globals"

s* in program source file: "Show Edit"

All other errors (undenoted) are found in program source file: "Editor TopLevel".

Table 9.1 A list of the 40 seeded errors and results of detection

Chapter 10

Other specification case studies

The preceding case studies of Logon and ThinkEdit expounded the proposed approach of deriving functional tests from formal specifications. In this chapter more case studies are conducted to show that the specification approach is applicable to a number of other user interfaces. These further case studies are presented in a less detailed manner, showing only a small portion of the specifications. Because of space and time constraints, the testing process will not be explored.

10.1 The X-Mail user interface

X-Mail is a program developed under the X Windows System environment [MIT89], intended to be a user interface for the Berkeley Mail system [Kernighan84]. In this case, the concept of dialogue separation is exemplified, as the user interface (i.e. X-Mail) is physically isolated from the application B-Mail (i.e. the Berkeley Mail system). The outputs of X-Mail are text strings corresponding to valid B-Mail commands. The communication between X-Mail and B-Mail is entirely external, via UNIX pipes. Alternatively, the output from X-Mail can be stored in a file, which is then used as an indirect input to B-Mail.

The specification of X-Mail can be viewed as a mapping between input sequences, and a subset of all valid Berkeley Mail commands supported by X-Mail. For example, the user interaction of selecting the "file" command button, together with choosing certain menu options, will result in a command message being passed from X-Mail to B-Mail.

For instance, a command message can be one in the form of : "save [message-list] [filename]" .

Similarly, all valid combinations of command button and menu option selections would result in command messages being sent to the Berkeley Mail system. The functions of the X-Mail user interface are tested by generating input sequences to try out combinations of menu options, and by checking display objects and application messages with the expected outcome. Display objects of X-Mail are identified in a WinSTD as shown in Figure 10.1.

A total of 52 objects and 46 functions have been identified. The 6 composite objects : OBJ00, OBJ10, OBJ20, OBJ30, OBJ210, OBJ220 have no specific functions, other than window management functions. More details of specification can be found in [Yip91b].

A small fraction of the specification of X-Mail is given below. It can be seen that the earlier version of WinSpec notations is used, in that display objects are given numerical names.

<u>Specification for Function</u> F01 :

From_state :	PostF00
F_state_predicate :	is_visible(OBJ00)
Inputs :	is_inside(mp?, OBJ01)
To_state :	PostF01
T_state_predicate :	is_hiLit (OBJ01)
Output_msg	none

<u>Specification for Function</u> F10 :

From_state :	PostF01
F_state_predicate :	is_hiLit (OBJ01)
Inputs :	mb?=<click>
To_state :	PostF10
T_state_predicate :	is_visible (OBJ10)
Output_msg	none

<u>Specification for Function</u> F11 :

From_state :	PostF10
F_state_predicate :	is_visible (OBJ10)
Inputs :	is_inside(mp?, OBJ11)
To_state :	PostF11
T_state_predicate :	is_hiLit (OBJ11)
Output_msg	none

<u>Specification for Function</u> F110 :

From_state :	PostF11
F_state_predicate :	is_hiLit (OBJ11)
Inputs :	mb?=<click>
To_state :	PostF110
T_state_predicate :	is_not_visible(OBJ10)
Output_msg	app_msg_sent("file %")

The text-editing function of OBJ211 is only counted as one of the 46 functions. The idea is that the single function identified for OBJ211 would be decomposed into edit-display functions similar to those specified for ThinkEdit in Chapter 8. This would be an example of the re-use of a functional specification across the platforms of Macintosh (for ThinkEdit) and Unix (for X-Mail).

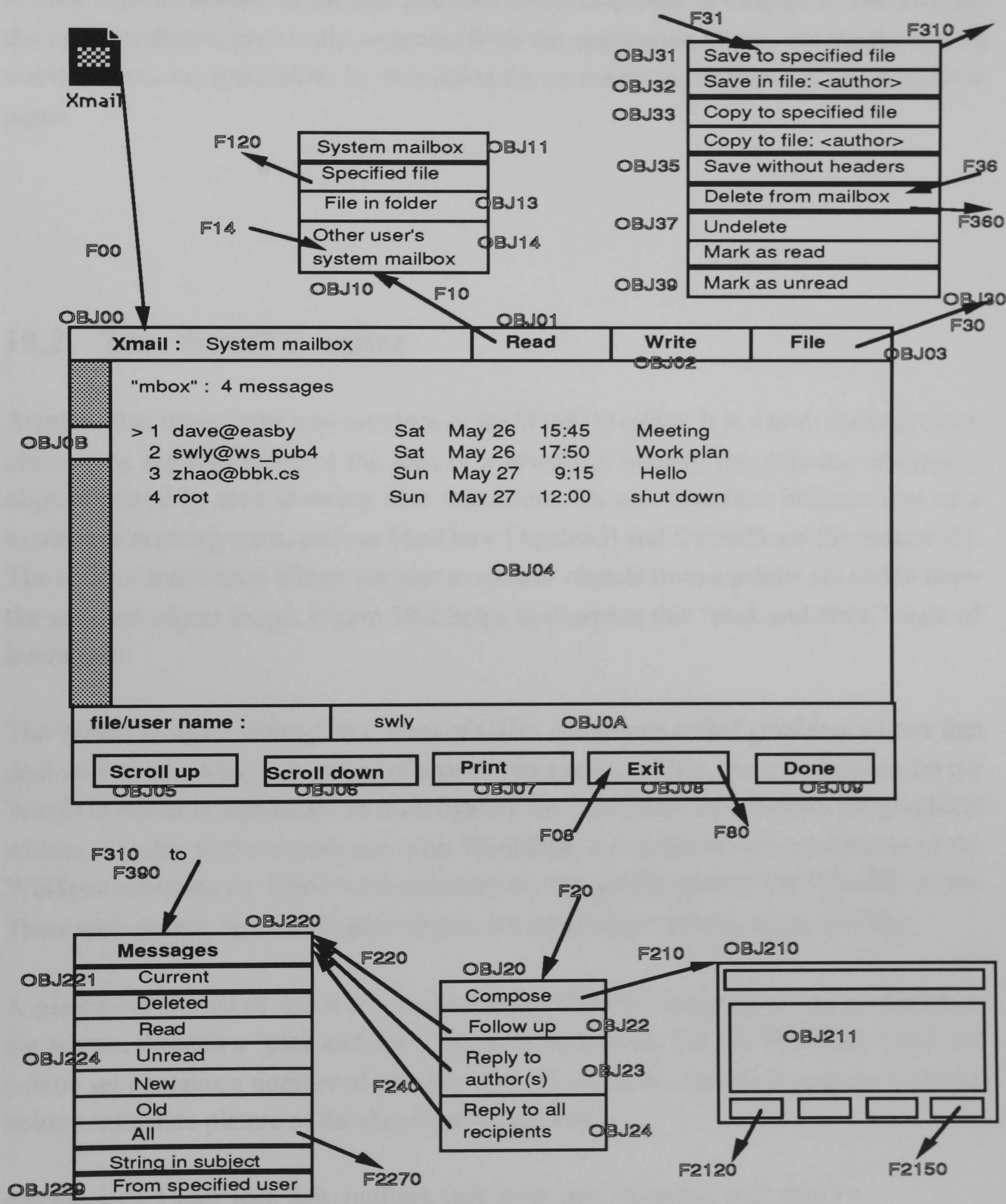


Figure 10.1 A WinSTD for the main parts of the X-Mail user interface

The process of constructing the formal specification has itself helped to identify a number of errors in X-Mail. The implementation was found to be incomplete, with errors in mapping messages to functions, and some interaction objects being unreachable because of missing functions. X-Mail is a locally produced, unreliable and incomplete software product. It is not possible to apply the same analysis and evaluation of the testing process that was carried out with ThinkEdit to X-Mail.

The extensive use of messages to communicate with the underlying application has made X-Mail a good example of the user interface model discussed in Chapter 3. The fact that the user interface is physically separated from the application allows the interface to be tested without the application, by simulating the necessary application messages on Unix pipes.

10.2 The WinSTD editor

Another case study is the user interface of the WinSTD editor. It is a tool constructed to investigate the feasibility of the idea of a WinSTD, namely the drawing of display objects joined by arcs showing state transitions. Its user interface imitates that of a number of drawing tools, such as MacDraw [Apple85] and ObjectDraw [Symantec90]. The style of interaction allows the user to choose objects from a palette set and to draw the selected object shape. Figure 10.2 helps to illustrate this "pick and draw" style of interaction.

The WinSTD editor belongs to a class of GUIs, commonly called graphical editors that deal with the drawing and editing of shapes and graphics. Thus, the specification for the WinSTD editor is significant in investigating the usefulness of WinSpec for graphical editors. Similar to the experience with ThinkEdit, a number of new constructs of the WinSpec notations are found to be necessary for the specification of the WinSTD editor. Three new generic types of display objects are introduced : palette, shape and line.

A palette set is a set of small icons representing different shapes that can be drawn on the screen, through a "pick and draw" style of interaction. For the WinSTD editor, the palette set contains a number of palettes, each of which is a small rectangular icon that holds a miniature picture of the shape that it can draw.

palette_set = { palt_rect, palt_rndRect, palt_oval, palt_rhombus, palt_line }

where palt_rect is a palette for drawing a rectangle
palt_rndRect is a palette for drawing a round rectangle

palt_oval is a palette for drawing an oval
palt_rhombus is a palette for drawing a rhombus
palt_line is a palette for drawing a line

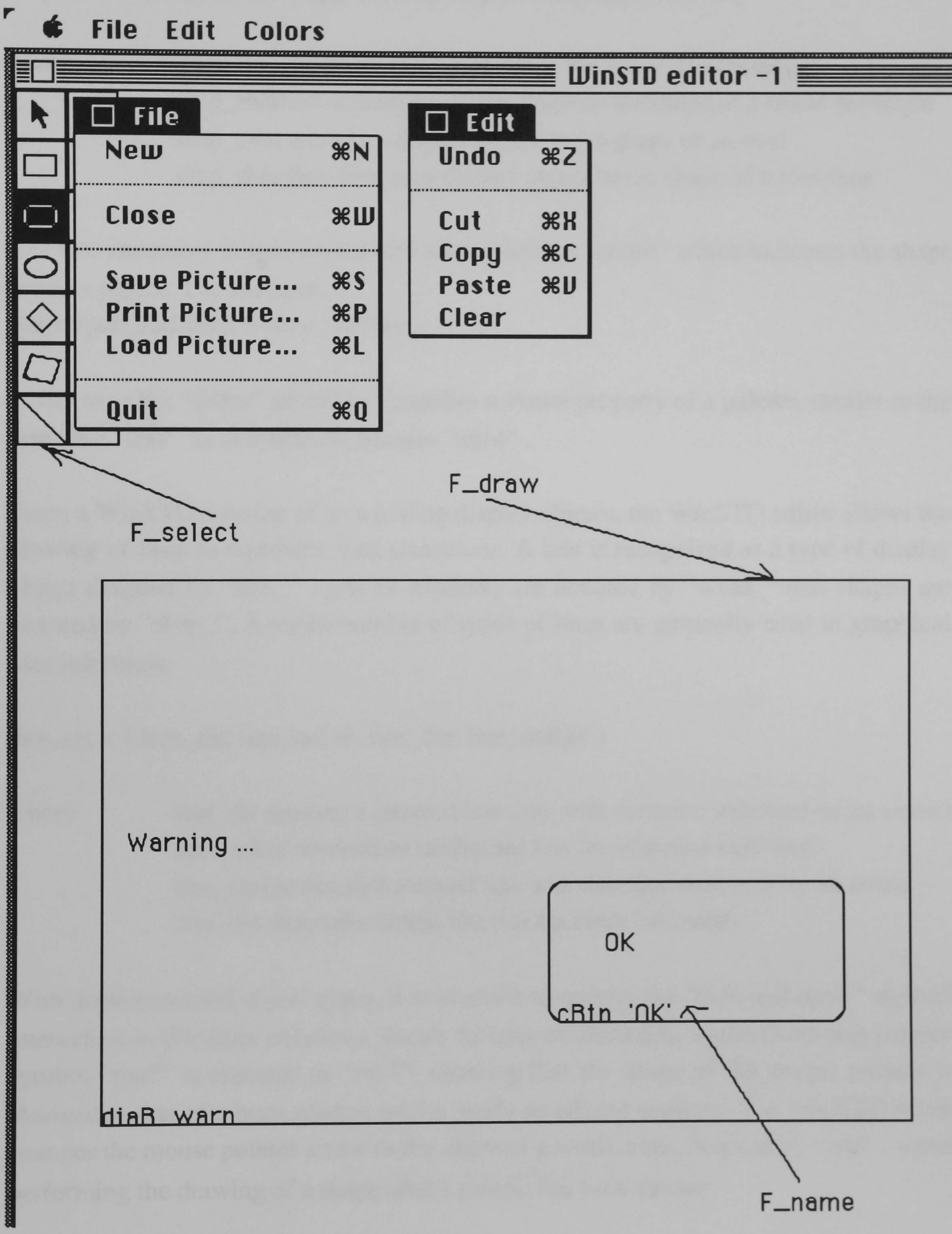


Figure 10.2 A WinSTD for part of the WinSTD editor user interface

The shapes that are being drawn on the screen are denoted by the generic display object type of "shap_" , followed by the denotation of the type of the shape, e.g. "shap_rect" represents the shape of a rectangle. The shapes that the WinSTD editor can draw are the members of the palette set.

shape_set = { shap_rect, shap_rndRect, shap_oval, shap_rhombus }

where shap_rect denotes a display object in the shape of a rectangle
shap_rndRect denotes a display object in the shape of a round rectangle
shap_oval denotes a display object in the shape of an oval
shap_rhombus denotes a display object in the shape of a rhombus

It is also necessary to introduce a new state primitive "shape" which indicates the shape inside a palette. For example,
shape (palt_rndRect) = shap_rndRect .

In this way the "shape" primitive describes a visual property of a palette, similar to the primitive "text" as in text(texB_name)= "abcd" .

Since a WinSTD consists of arcs joining display objects, the WinSTD editor allows the drawing of lines to represent state transitions. A line is recognized as a type of display object denoted by "line_" , just as windows are denoted by "wind_" and shapes are denoted by "shap_". A small number of types of lines are generally used in graphical user interfaces.

line_set = { line_dir, line_unDir, line_dot, line_dotDir }

where line_dir denoted a directed line (one with direction indicated by an arrow)
line_unDir denoted an undirected line (no direction indicated)
line_dotDir denoted a dotted line with direction indicated by an arrow
line_dot denoted a dotted line (no direction indicated)

With these extended object types, it is possible to specify the "pick and draw" style of interaction in WinSpec notations. Recall the case of ThinkEdit where the mouse pointer symbol "mp?" is changed to "mk?", showing that the shape of the mouse pointer is changed to that of a book marker whilst inside an editing window. The WinSTD editor changes the mouse pointer arrow to the shape of a small cross, denoted by "mc?", when performing the drawing of a shape after a palette has been chosen.

Part of the specification of the interactions, starting with the invocation of the WinSTD editor, is given below. The user interface goes through its main processing cycle in three steps : selecting a palette, drawing the selected shape, and prompting the user to enter a name for the shape drawn.

Specification for function **F_invoke**

From_state :	Start
F_state_predicate :	is_visible(icon_'WinSTD')
Inputs :	is_inside(mp?, icon_'WinSTD') Tand mb?=<dClick>
To_state :	Post_Invoke
T_state_predicate :	is_visible(wind_'WinSTD editor')
Output_msg	none

Explanations:

- The function F_invoke starts the execution of the WinSTD editor.
- The window wind_'WinSTD editor' is displayed. The palette set is assumed to be part of the window and thus is also visible.

Specification for function **F_select**(aPalette)

Variable :	aPalette ∈ palette_set
From_state :	Post_Invoke
F_state_predicate :	is_visible(wind_'WinSTD editor')
Inputs :	is_inside(mp?, aPalette) Tand mb?=<click>
To_state :	Post_Select
T_state_predicate :	is_hiLit(aPalette)
Output_msg	none

Explanations:

- The function F_select allows the user to select a shape for drawing by the input of a mouse click within the palette icon for the shape desired.
- The variable aPalette must be a member of the palette_set supported by the WinSTD editor.
- The selected palette will be highlighted throughout the "pick and draw" interaction.

Specification for function **F_draw** (aShape)

Variable :	aShape ∈ shape_set
From_state :	Post_Select
F_state_predicate :	is_hiLit(aPalette)
Inputs :	Loc(mc?)=(pt1) Tand mb?=<down> Tand Loc(mc?)=(pt2) Tand mb?=<up>
To_state :	Post_Draw
T_state_predicate :	is_visible(aShape) and rect(aShape) = (pt1, pt2) and aShape = shape(aPalette) Tand is_modal(diaB_name)
Output_msg	none

Explanations:

- After a shape has been selected, it can be drawn within the WinSTD editor window.
- The location and size of the shape drawn is determined by a sequence of mouse interactions, as stated in the Inputs clause.

- As mentioned earlier, the arrow shape of the mouse pointer is changed to the shape of a cross, indicated by the mouse pointer notation "mc?".
- The location within the drawing window where the mouse button is first pressed down, after a palette has been selected, is remembered as pt1.
- The location where the mouse button is subsequently released is denoted as pt2.
- Between pt1 and pt2, the bounding rectangle is fixed, within which the selected shape will be drawn.
- Eventually, a modal dialogue box diaB_name appears to assist the user to name the object that has just been drawn, as in the specification for F_name.

Specification for function **F_name**(aString)

From_state :	Post_Draw
F_state_predicate :	is_modal(diaB_name)
Inputs :	kb?=aString Tand is_inside(mp?, cBtn_'OK') Tand mb?=<click>
To_state :	Post_Name
T_state_predicate :	text(texB_name) = aString
Output_msg	app_msg_sent= "add:" // aString // aShape // (pt1,pt2)

Explanations:
- The function F_name is the interaction used to name the display object (or shape) that has just been drawn through the "pick and draw" interactions.
- The modal dialogue diaB_name has a text entry field texB_name that allows a text string to be entered, which will be used as the name of the object.
- Eventually, an application message is sent to add the new object onto the database. The message sent contains the name of the object, its type of shape, and the two coordinate points (pt1, pt2) recording the bounding rectangle of the object. This information is vital to the test generation process.
- The symbol // again stands for text concatenation.

A proper testing experiment has not been conducted on the WinSTD editor. Being a primitive prototype, the WinSTD editor has a number of design and implementation deficiencies that render a formal functional testing inappropriate at this stage. However, this prototype has shown that the idea of drawing and storing a WinSTD is feasible, and the "pick and draw" style of interaction can readily be specified in WinSpec notations.

10.3 The JRR tool

A "Journal Record and Replay", or JRR tool, is an automation mechanism whereby keyboard and mouse inputs generated by a user can be recorded for later playbacks or re-runs. The use of a JRR tool for testing GUIs was introduced in Chapter 3. JRR and other automation issues are discussed in Chapter 12.

The discussion in this section is only concerned with how the user interface of a JRR tool can be specified in WinSpec. As part of the research into GUI testing, a rudimentary JRR tool has been developed. It has a very simple user interface consisting only of one menu and one window. The JRR menu has four menu options : Record, Playback, Off and Exit. They are used for turning recording on and off, switching over to playback mode , and eventually exiting the JRR tool. The JRR window is for display only and does not support inputs. The window displays (or echoes) all input events being recorded or replayed by the JRR tool.

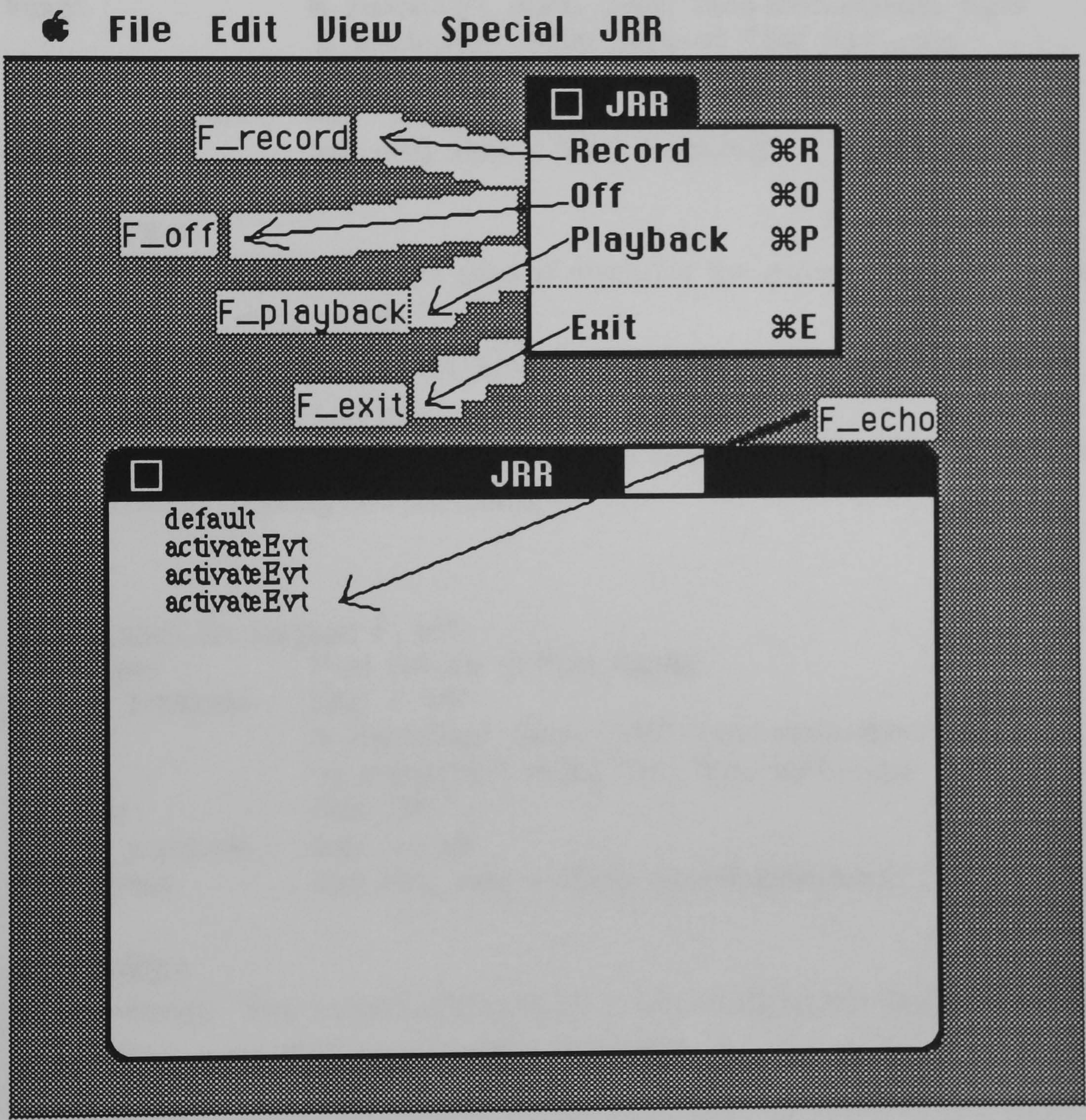


Figure 10.3 The user interface of the JRR tool illustrated as a WinSTD

The user interface specifications are simple. The task of intercepting and recording input events is actually performed by the underlying application, not the user interface. The

JRR application can be grouped amongst programs generally called device drivers, close to the internals of the window system. The displays of events being processed, on the JRR window, are modelled as application message received. The commands to turn on (or off) recording or playback are modelled as messages sent to the application. The WinSpec specifications for the JRR tool are given below.

<u>Specification for function</u> **F_record**

Variable	flag ∈ {record, playback, off}
From_state :	Post_Off
F_state_predicate :	flag = "off"
Inputs :	is_inside(mp?, menu_'JRR') Tand mb?=<down> Tand is_inside(mp?, mOpt_'Record') Tand mb?=<up>
To_state :	Post_Record
T_state_predicate :	flag' ="record"
Output_msg	app_msg_sent = "start recording"

<u>Explanations:</u>

- The global variable "flag" is used to represent the current state of processing : recording, replaying, or off.
- The mouse interaction to select the menu option "Record" is similar to that of other interfaces encountered earlier.
- The message "start recording" is sent to the underlying device driver that actually performs the recording of input events.

<u>Specification for function</u> **F_off**

From_state :	Post_Record or Post_Replay
F_state_predicate :	flag ≠ "off"
Inputs :	is_inside(mp?, menu_'JRR') Tand mb?=<down> Tand is_inside(mp?, mOpt_'Off') Tand mb?=<up>
To_state :	Post_Off
T_state_predicate :	flag' = "off"
Output_msg	app_msg_sent = "Stop recording/playback"

<u>Explanations:</u>

- The message "Stop recording/playback" is sent to signal the device driver to stop processing, as the "Off" menu option is selected.

<u>Specification for function</u> **F_replay**

From_state :	Post_Off
F_state_predicate :	flag = "off"
Inputs :	is_inside(mp?, menu_'JRR') Tand mb?=<down> Tand is_inside(mp?, mOpt_'Playback') Tand mb?=<up>
To_state :	Post_Replay
T_state_predicate :	flag' = "playback"
Output_msg	app_msg_sent = "Start playback"

Explanations:

- The function of the "Playback" menu option is similar to that of "Record", except that the device driver is requested to perform replay, instead of recording.

Specification for function **F_exit**

From_state :	Post_Off
F_state_predicate :	flag = "off"
Inputs :	is_inside(mp?, menu_'JRR') Tand mb?=<down> Tand is_inside(mp?, mOpt_'Exit') Tand mb?=<up>
To_state :	Post_Exit
T_state_predicate :	is_not_visible(wind_'JRR') and is_not_visible(menu_'JRR')
Output_msg	app_msg_sent = "exit JRR"

Explanations:

- Execution of the JRR tool is terminated as the "Exit" menu option is chosen. The menu and window of the JRR tool will disappear from display.

Specification for function **F_echo**

From_state :	Post_Record (or Post_Replay)
F_state_predicate :	flag ≠ "off"
Inputs :	app_msg_recv ≠ {}
To_state :	Post_Record (or Post_Replay)
T_state_predicate :	text'(wind_'JRR') = text(wind_'JRR') // app_msg_recv // <cr>
Output_msg	none

Explanations:

- During processing (record or playback), the application (in conjunction with the device driver) sends messages to the user interface. These contain names of events that are being processed and the user interface displays them on the JRR window.
- The denotation text'(wind_'JRR') represents the text display on the JRR window after the execution of function F_echo.
- Again, the // symbol indicates text concatenation. The new message is effectively displayed by concatenating it onto the existing text on the window, followed by a carriage-return to prepare a new line for the next message.

Recalling the discussion on macro- and micro-communications (chapter 3), it is possible to alter the division of work between the JRR user interface and the underlying application. It is conceivable to model some of the actions of record and playback within the user interface, where every kb?, mp? and mb? inputs received during recording will be sent as messages to the application for storage. The actions of replay can be portrayed in specifications by channelling contents of application messages to become keyboard and mouse inputs.

Proper testing of the JRR tool has not yet been pursued, as it still has the occasional symptom of system crashes during recording. A number of JRR tools are now commercially available and there is little doubt that the idea of JRR for GUIs is feasible. This section has shown that the functions of a JRR tool can be specified in WinSpec notations.

10.4 Summary

The last chapter demonstrated that test case generation is relatively straightforward, provided that formal specifications of the functions are available. This chapter aims to promote the idea that the WinSTD and WinSpec approach to specification is generally usable for different classes of graphical user interfaces. Three GUIs have been considered: X-Mail, the WinSTD editor and a JRR tool. They belong to different classes of user interfaces, as their respective application and style of interaction are different. X-Mail, a user interface for a mail program, is a typical example of ordinary user interfaces. The WinSTD editor is special as it belongs to the class of graphics editors. The JRR tool is an example of a systems tool that has a very simple user interface. In all of these cases, the WinSTD and WinSpec specification approach is adequate. However, extensions to WinSpec notations are necessary when encountering new object types and styles of interactions. All three of these GUIs are still at too immature a stage of development to warrant the formal functional testing process. The argument is that once a user interface is specified in WinSpec, it will be possible to derive test cases from the specification. This chapter has further evaluated the specification process through case studies. The discussion of automation issues is returned to in the next chapter.

Chapter 11

Automation Issues

Automation issues were first discussed in Chapter 3. They are raised again in this chapter to discuss tools that have been explored and to pinpoint future directions for automation. The automation of the GUI testing process can be divided into three sub-processes: test data generation, test execution and test result analysis. As mentioned in Chapter 3, the Journal Record and Replay (JRR) mechanism has been advocated and implemented by a number of researchers and practitioners as the answer to the automation of test execution. The screen snapshot and bitmap comparison methods have been suggested as feasible means of assessing the result of test executions, by a number of software vendors. The problem of test generation remains largely unexplored. This thesis has demonstrated the feasibility of deriving test cases from formal specifications. It proposes that formal functional testing (FFT) should lead the test generation process towards automation.

This chapter describes a set of prototype tools developed to explore the automation of FFT. Recall that test cases are derived from specifications which consist of WinSTDs and WinSpec notations. The tools involved are: a WinSTD editor for constructing WinSTDs, a WinSpec parser, a test case generator (TCG) and a JRR tool. For completeness, the activities of leading software vendors, in the automation of GUI testing, are also described.

11.1 WinSTD editor

The WinSTD editor was introduced in Chapter 10, as a drawing tool for constructing WinSTDs for user interfaces. The design and implementation of a graphics editor like the WinSTD editor is not new. Indeed, there are a range of such products available commercially. The WinSTD editor is different because of its attempt to extract information of graphics objects that are being drawn. In a WinSTD, all interaction objects are enumerated together with the main interaction functions. Information about display objects (such as their name, type of shape and (x,y) coordinates), is essential to test generation, and is stored in the internal database of the WinSTD editor. All other information necessary for test generation is contained within the WinSpec.

As illustrated in Chapter 10, the WinSTD editor uses the "pick and draw" style of interaction to enable a WinSTD to be built from a set of standard shapes. After a display object is drawn, the WinSTD editor will require an unique name to be entered for that object. Interaction functions (or transitions) are shown by drawing and labelling a line from one object to another. It has been observed that a WinSTD clearly enhances human testers' understanding of a user interface and assists in its testing. The prototype has demonstrated that information about display objects can be captured by a WinSTD editor. In the original design of a WinSTD, all interaction functions were to be enumerated and represented by arcs with labels. This idea worked well for a very small interface such as Logon, but turned out to be impractical for more complex user interfaces. Finally, it was decided that a WinSTD should only present the main interaction functions. The task of enumerating all interaction functions is best dealt with in WinSpec, which is a machine-readable denotation.

11.2 WinSpec Parser

A WinSpec parser extracts information about the required inputs and expected outputs from the "Inputs" clauses and T_state_predicates in a specification. The first task of the parser is to detect any syntactical errors in a WinSpec specification. When the specification of an interaction function have been successfully parsed, an entry is added to a table called the F-Table. The F-Table holds information about all interaction functions for the GUI being specified. Information in the F-Table is organized to be readily usable by the Test Case Generation (TCG), to produce the required input sequences to test the user interface.

Each function has its own entry in the F-Table. Each entry is divided into five fields : Name, Tested, From, To and Inputs. The first field contains the name of the interaction function, which is used as the key for accessing the database. The "Tested" field contains an integer which is initialized to zero, indicating that the function has not yet been tested. The “From” field lists the state(s) that can precede (or lead to) the function of this entry. The “To” field lists the state resulting from the execution of this function. The “Inputs” field contains the required input to test the function.

Name	Tested?	From field	To field	Inputs field
F1	0	Start	postF1	is_inside(mp?, icon_Logon) Tand mb?=<click>
F2	0	postF1	postF1	kb?
F2.1	0	postF1	postF4	kb?=<cr>
F2.2	0	postF1	postF2.2	kb?=<tab>
F2.3	0	postF1	postF2.2	is_inside(mp?, texB_pass)
F3	0	postF2.2	postF2.2	kb?
F3.1	0	postF2.2	postF4	kb?=<cr>
F3.2	0	postF2.2	postF1	kb?=<tab>
F3.3	0	postF2.2	postF1	is_inside(mp?, texB_user)
F4	0	postF1, postF2.2	postF4	is_inside(mp?, cBtn_'OK') mb?=<click>
F5	0	postF4	postF5	app_msg_recv=”Logon failure”
F6	0	postF5	Start	is_inside(mp?, cBtn_'Reset') mb?=<click>
F7	0	postF4	postF7	app_msg_recv=”Logon ok”
F8	0	postF7	Start	is_inside (mp?, cloB_term)

Figure 11.1 A Function Table (F-Table) for the Logon user interface

An example of the F-Table for the Logon interface is shown in Figure 11.1. The first entry in this F-Table is function F1. The second entry is F2, which is yet untested, as the "Tested" field is 0. The “From” field of F2 has the content of “postF1”, indicating that F2 can be executed following F1. The “To” field of F2 is also “postF1”, showing that the user interface returns to the same state (i.e. postF1) after the execution of F2.

The third entry is F2.1, which also has “postF1” in its “From” field. This means that either F2 or F2.1 can be executed after F1. If F2.1 is selected for execution, it will lead the user interface into the state “postF4”, as can be seen in the “To” field of the entry for F2.1 .

F2.2, which is the 4th entry in the table, can also follow the execution of F1. The "To" field of F2.2 indicates that “postF2.2” is the state following the execution of F2.2. The contents of “From” and “To” fields are extracted from the WinSpec From_state and To_state. For instance, the From_state for F2 is “postF1”. Similarly, the F_state_predicate for F3 requires “has_kb_focus(texB_pass)”, which is the To_state_predicate of F2.2, as given in the Logon specifications in Chapter 5. The STD for Logon (Figure 5.3) is reproduced in Figure 11.2 for convenient reference.

A parser for an earlier version of WinSpec has been implemented, using lex and yacc [Schreiner86]. However, as explained in Chapter 5, the WinSpec notations have subsequently been changed for improvements. More development is required to update the parser and the TCG for an automated test case generation. Some of the detailed internal workings of the parser and the TCG have yet to be implemented. Nevertheless the idea of Formal Functional Testing (FFT) for GUIs has largely been explored by the case studies in chapters 7 to 10.

11.3 The Test Case Generator (TCG)

The task of the TCG is to select functions from the F-Table to form test sequences. The formation of test sequences, as discussed in section 9.2, requires that all interaction functions be covered at least once. Two adjacent functions in a sequence must have matching To_state and From_state. In order to reduce the number of test sequences, an algorithm is developed in the following section for selecting functions. When a function is selected to go on the test sequence, the required keyboard and mouse inputs for that function are written onto the Test Inputs File (the TI-File). When inputs concerning mouse pointer location are required (e.g.is_inside(mp?, mOpt_'File')), a reference is made to the WinSTD database (also called the W-Table) to obtain the coordinates of the display object (i.e. mOpt_'File' in this example).

11.3.1 Developing a sequence selection algorithm

The algorithm to generate test sequences can be outlined in four steps. These steps are presented in the form of pseudo-code, together with comments. The first step is to determine if all the interaction functions have been covered by test sequences already generated; if so the TCG will terminate processing.

• Step 0:

```
finished := TRUE;                            ! Assuming processing should finish.
LOOP i : 1, I  where I=len(F-Table)          ! A loop to check through all entries,
    IF F-Table(i).Tested = 0 THEN            ! len(F-Table) is the length or total no.
    BEGIN                                    ! of entries in the F-Table. If the Tested
        finished := FALSE ;                  ! field of the i th entry is 0, i.e. untested,
        EXIT LOOP i;                         ! TCG should not terminate processing.
    END ;                                    ! If one untested function is found,
END LOOP i;                                  ! another test sequence has to be
IF finish = FALSE  THEN                      ! generated by executing Step 1 of
    GOTO Step 1                              ! the algorithm.
ELSE
BEGIN
    print("Finished ... all interaction functions covered !")
    STOP;
END;
```

The first entry in the F-Table always has "F_start" (or "F1") in its "Name" field, "Start" in its "From" field. F1 is always the first function in any test sequence.

• Step 1:

```
Last_func_in_seq := "F1"
                              ! Insert F1 onto test sequence.
TI_File := TI_File // F1.Inputs
                        ! Insert what is in Inputs_field of F1 into TI_File
F1.Tested := F1.Tested + 1
                              ! Increment the "Tested?" field of entry F1.
candid_func := "null"         ! Initialize, no candidate chosen to go on the
pass := 0;                    ! test sequence. Set pass mark to 0, only
GOTO  Step 2                  ! functions with Tested?=0 are considered.
```

Having selected F1 (or F_start), the algorithm searches through the F-Table to find the function best suited to be the next function in the sequence being built. Once a candidate (candid_func) passes the selection, the contents of its "Inputs" field is appended to the TI-File, and its "Tested?" field is incremented. Then a new search is started from the top of the function table to find the next function to follow the one that has just been added to the test sequence. This is the main processing loop in Step 2.

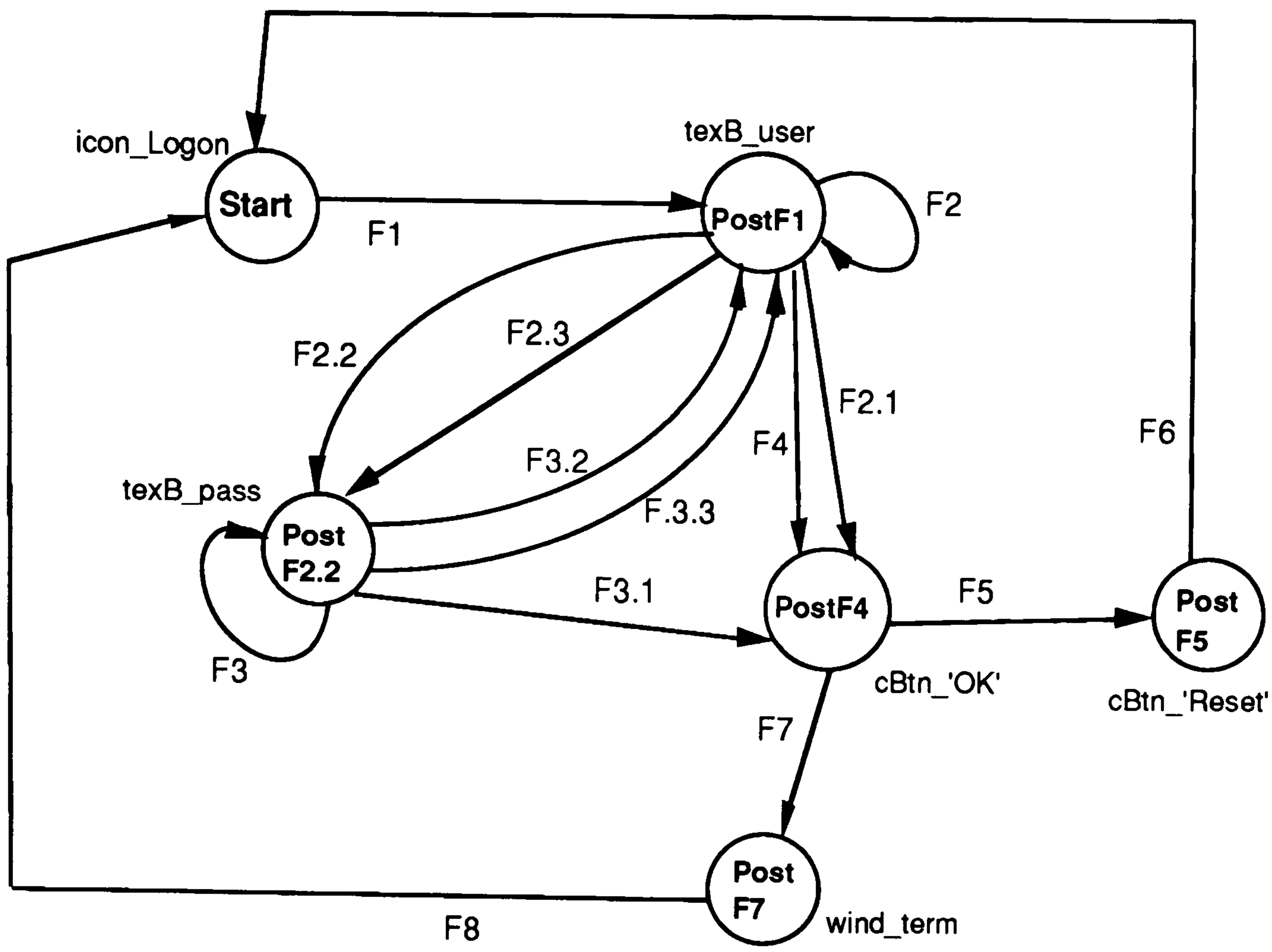


Figure 11.2 An STD for the Logon interface

When searching the F-Table, all entries with 0 in their "Tested?" fields are examined to see if their "From" field matches the last function in the sequence. If a match is found, it becomes the candid_func. Then its "To" field is examined and followed to see if it leads to a loop, or if it leads to any untested functions, as detailed in step 2.

```
• Step 2:  candid_func := null;                              ! Initialize candid_func .

           LOOP i : 2 to I ,   where I=len(F-Table)          ! No need to check 1st
              IF Last_func_in_seq.To = "end"  THEN           ! entry of F-Table which
                 BEGIN                                       ! always contains F1.
                    PRINT "End of test sequence" ;           ! Stop processing if end of
                    STOP ;                                   ! sequence reached.
                 END ;
              func := F-Table(i).Name ;                      ! Examine i th entry.
              IF func.Tested≤ pass AND func.From=Last_func_in_seq.To
              THEN                                           ! If entry is untested, and
              BEGIN                                          ! matches last function.
                 IF pass > 0 THEN pass := pass - 1;          ! If "pass" has been relaxed
                 candid_func := func ;                       ! to > 0, tighten it.
                 IF candid_func.To = Last_func_in_seq.To       ! Choose a function
                   THEN Last_func_in_seq := candid_func        ! that returns to the same
                   ELSE  Step 2a ;                           ! state. 2nd choice is Step 2a.
                 IF Last_func_in_seq = candid_func           ! If candid_func is added to
                   THEN EXIT LOOP i                          ! sequence, exit loop.
                   ELSE Step 2b ;                            ! Otherwise try 3rd choice in
                 EXIT LOOP i ;                               ! Step 2b. Eventually exit
              END ;                                          ! Loop i.
           endLOOP

           IF Last_func_in_seq ≠ candid_func  AND candid_func ≠ null  THEN
              BEGIN                                  ! If none of the 3 choices succeed,
              Last_func_in_seq := candid_func ;      ! just add candid_func to sequence.
              END;                                   ! If no candid_func is found,
           IF candid_func = null  THEN               ! relax pass mark from 0 to 1
           BEGIN                                     ! or from 2 to 3 ...,  to choose
              pass := pass + 1;                      ! a candid_func that's already tested,
              GOTO Step 2 ;                          ! as end of test sequence not yet
           END ;                                     ! reached. Return to repeat Step 2.
```

The line "candid_func.To = Last_func_in_seq.To" above states that the functions which could follow candid_func are any of the functions allowed to follow Last_func_in_seq . Effectively, candid_func takes the user interface back to the state before its execution, as the same set of functions can both precede or follow candid_func. This is the first choice : an untested function that effectively loops back to the same state. The second choice is two untested consecutive functions which together would effectively bring the user interface back to the state existed before their execution. That is when func1.From=candid_func.To AND func1.To=Last_func_in_seq.To, as shown in Step2a.

• Step 2a:
```
LOOP j : 2 to I ,  where I=len(F-Table)
    func1 := F-Table(j).Name ;
    IF func1.From=candid_func.To  AND  func1.To=Last_func_in_seq.To
    THEN
    BEGIN
        Last_func_in_seq := candid_func  ;
        candid_func := func1 ;
        Last_func_in_seq := candid_func  ;
    END;
END LOOP j  ;
```

The third choice is any untested function that would lead to another untested function. That is when func1.Tested = 0 AND func1.From=candid_func.To, as outlined in Step 2b. (Whereas the last choice is any untested function that can follow the Last_func_in_seq. This can be seen in Step 2, on the 9th line from the end of Step 2.)

• Step 2b:
```
LOOP j : 2 to I ,  where I=len(F-Table)
    func1 := F-Table(j).Name ;
    IF func1.Tested = 0  AND  func1.From=candid_func.To
    AND func1≠ candid_func
    THEN
        BEGIN
        Last_func_in_seq := candid_func  ;
        candid_func := func1 ;
        Last_func_in_seq := candid_func  ;
        END;
END LOOP j  ;
```

11.3.2 Observation of sequences

The above algorithm was developed following the observation of a very simple yet useful phenomenon in GUI interactions. Starting from a certain state (or object), the execution of a simple sequence of two functions would often return the user interface to the same state (or object embracing states). There are numerous examples; for instance, a user selects a "Quit" menu option during text editing, then decides to choose the "Cancel" command button to cancel the "Quit" command, when faced with a dialogue box giving warnings such as "Unsaved changes". The two consecutive interaction functions, "Quit" and "Cancel", bring the user interface back to its state before the "Quit" command was initiated. This observation is useful for generating effective test sequences, as the user interface is brought back to a previous state, so that more functions can be tested within the same test sequence.

Referring to Figure 11.2, function pairs (F2.2 o F3.2) and (F2.3 o F3.3) are such examples. The reader may like to refer to Figure 8.3 for similar loop back functional paths in ThinkEdit. A point of wider concern is that graphs of interaction functions, as in Figure 11.2 and Figure 8.3, would lend themselves to analysis methods that are usually applied to structural graphs. This is because the formal specification of program functions can now offer software engineers the kind of formal attributes of software, which were previously mainly obtainable from program code only.

11.3.3 TCG implementation issues

The operations of a Test Case Generator are mainly centred around the F-Table and the selection algorithm. However there are a number of minor implementation issues.

- User interfaces, which have an unbalanced tree structure in their state transition networks (as shown in Figure 11.3), may exist.

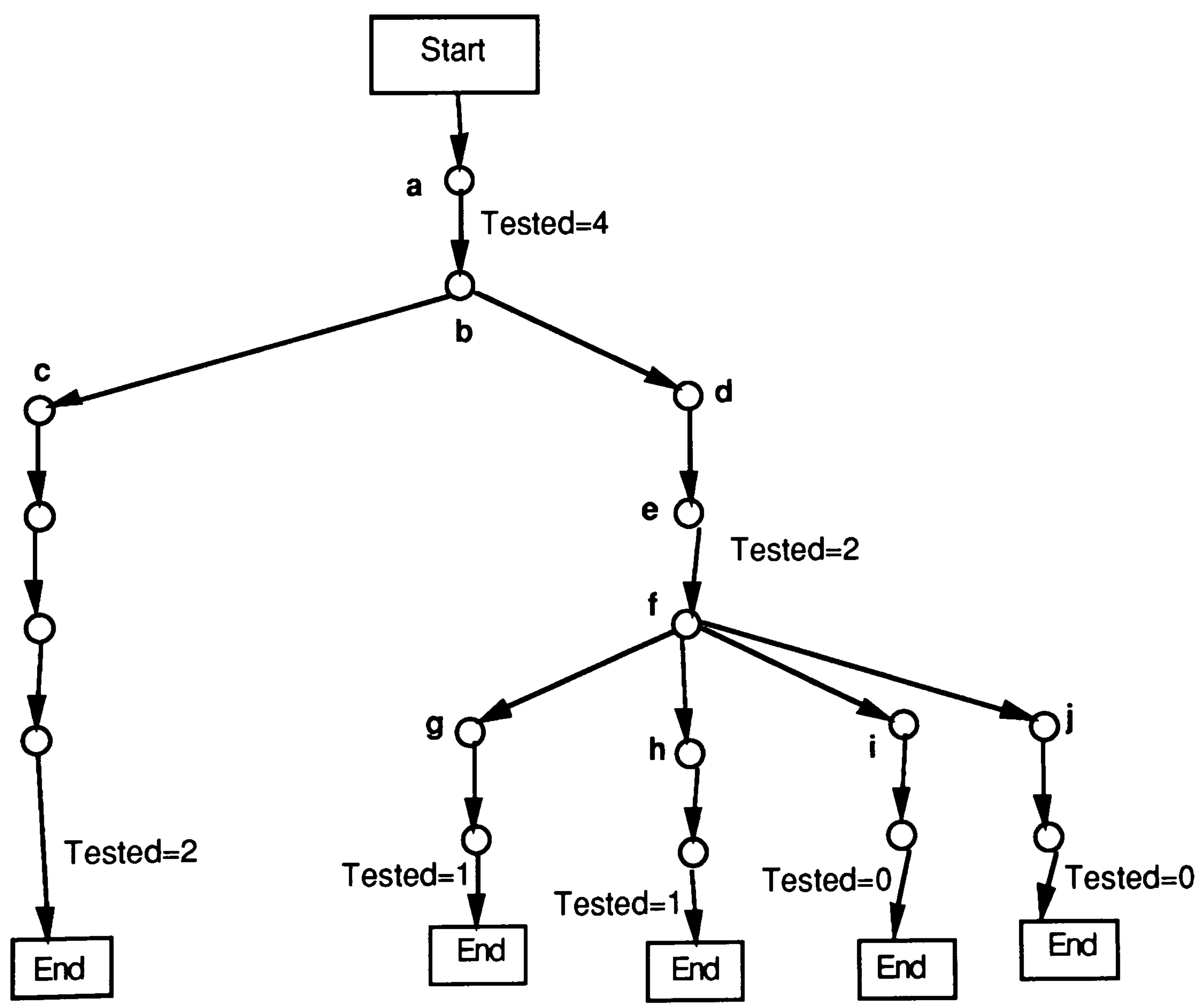


Figure 11.3 An unbalanced tree showing where a 2 step look ahead is insufficient

The selection algorithm in section 11.3.1 would not perform efficiently, as it does not look beyond two nodes ahead for untested functions. It would choose to go down from node b to node c instead of node d, as both node c and node d have been traversed (or tested) twice. The algorithm does not look beyond node f, and does not realize that nodes i and j have not yet been tested. It is necessary to enhance the test generator by implementing the RCPT algorithm as described in chapter 6. This will meet the need for regression testing after the software maintenance process, as discussed in [Yip91d].

- Although the WinSTD parser and the TCG can automate the test generation process, there is still a need for some application-specific or heuristic knowledge. For example, the knowledge of valid sets of username and password (for Logon) still lies with the human tester, and is not known to a general purpose TCG. In the case of ThinkEdit, a human tester uses the heuristic knowledge that at least a page or two of text is essential to make the testing of some of the scrolling interactions meaningful.

- In order to automate the test execution process, the test inputs generated by the TCG must be translated into a format that is acceptable to a JRR tool. This leads to the investigation of JRR tools in the following section.

11.4 Journal Record and Replay tools

The idea of Journal Record and Replay (JRR) is not new. It is a mechanism employed to record user inputs, which are later replayed for automation purposes. A survey of some commercially available JRR tools is given in Chapter 3. The reason for attempting to construct a JRR tool is to investigate the practical problems in translating user inputs from the TCG to the format required by the JRR tool. It was decided to store inputs in a textual format in the Test Inputs File (TI-File), which is convenient for test-design alterations. Eventually, the textual content of the TI-File is converted into window-system event codes for playback with the JRR tool. A prototype JRR tool for the Macintosh environment was constructed as an experiment. Some details of this JRR tool have been given in section 10.3. It was a limited implementation, but has demonstrated the feasibility of translating test inputs from textual format to event codes, for playback purposes.

It seems to be a reasonable suggestion that all window systems should have a built-in journal facility, possibly distributed with a vendor-supported JRR tool. It was decided that a commercially available JRR tool should be used for a future implementation, in view of the features available and the low level programming skill required.

11.5 Software vendors' approaches

This section reviews GUI validation in the software industry in order to balance the academic contents of this thesis. Vendors are making significant efforts to automate the GUI testing process, with new advances evolving. Brief visits to a number of leading window software vendors were made during April 1991, to survey the status quo in the industry. These vendors included the Open Software Foundation (OSF), Digital Equipment Corporation (DEC), Hewlett Packard, and Microsoft. The following points summarize the major findings of the visits.

- The testing of window systems is generally divided into system level and component (or widget) level. The X test suite for the X Windows System, available in the public domain, is such an example. The X Test Suite covers a wide range of tests, from the sanity tests of the X protocol at device server level, through the comprehensive tests of XLIB routines, to the higher levels of volume and stress testing.

- The main mechanism used for automation is JRR. The normal industrial practice is to implement overnight runs using a JRR mechanism, executing large test suites consisting of many previously recorded applications.

- In attempts to reduce the long execution time (well exceeding 10 hours in many cases) of some large test suites, experiments to replay inputs at a faster rate than the real user inputs recorded are carried out. This also promotes an element of performance testing. However, there remains a number of synchronization problems [Su91].

- There is a growing use of bitmap comparison tools to validate results of test runs. However, the difficulties of such approaches still exist; for instance, in choosing appropriate snapshot points, as discussed in Chapter 3 ([Islam89], [Andreas91]).

- The design of test cases (i.e. interaction sequences) is mostly conducted informally, relying on personal creativity and experience. Some vendors have expressed the need to develop metrics to measure the quality of their tests. This may be an area for technology transfer, where experience of academic research in code / function coverage and error detection statistics can be recommended for industrial practice.

- Vendor-supplied assistance towards the testing of application user interfaces is restricted to hooks for the implementation of JRR tools. An example is the Input Synthesis Extension as part of X11 Release 3 from MIT, and the Client Exerciser in the X Test Suite.

11.6 Summary

This chapter has re-explored the testing of GUIs from the automation perspective. The design of test inputs, the execution of test cases, and the analysis of test results are the three main parts of the testing process that require automation. Examination of vendors' testing practices and marketed test tools reveal that JRR is the established strategy for automated test execution. A few major vendors are now pioneering the automation of the results-analysis process, by introducing bitmap comparison tools. The intellectually challenging problem of automating the test design and generation process is largely unaddressed by industry. This thesis has manually demonstrated the derivation of test cases from formal specifications. The WinSTD editor, WinSpec parser, TCG and JRR tool have been developed to explore the automation of the FFT approach. A complete and integrated implementation has not been possible because of time constraints. Nevertheless, the ideas behind these tools are found to be feasible through prototypes, or confirmed to be practical as similar tools are now commercially available. This concludes the discussion of automation issues.

Chapter 12

Review and Evaluation

In order to evaluate the capability of the FFT approach, a number of case studies and experiments have been conducted, as described in Chapters 7, 9 and 10. This chapter presents observations, insights and conclusions derived from the results of these case studies.

12.1 Findings from the testing of the Logon interface

The results of testing the Logon interface are given in section 7.4. The success rate of error detection was 90%, as 9 out of the 10 seeded errors were detected. Only 10 errors were injected, as the logon user interface is too small to warrant any more meaningful errors.

A total of 9 display objects and 14 functions have been identified and specified. The functional coverage criterion requires each interaction function to be invoked at least once. The 100% function coverage criterion appears to have worked well. It was found that only 3 test sequences were necessary to cover all functions. Test sequences were designed, by adopting the graph theoretic algorithms of the Euler tour and the postman tour (in Chapter 6), for an optimal coverage of functions. As a whole, the 3 test sequences cover all 14 functions, with some functions tested more than once.

The undetected error (E2, see section 7.5) is that of a very small shift in the screen position of the username textBox (texB_user). Even in this case, the textBox functions normally, as the defect is purely visual. For a rectangular object, the centre of the object (i.e. half way between the top-left and bottom-right corners) is used by the test case generator as the exact location of the object. This location is used in generating mouse pointer inputs for predicates such as is_inside(mp?, texB_user). The small shift in the position of texB_user is undetected because the generated mouse pointer input is still inside the shifted texB_user. For detecting these kind of errors, visual inspection (manual) and visual verification (automated) are more effective than functional testing.

12.2 Analysis of undetected errors in ThinkEdit

In chapter 9, a table of results was built from the testing of ThinkEdit. The ability of the four test cases (TS1 to TS4) to expose faults in ThinkEdit was as high as 80%. In the 40 errors seeded, 32 errors were detected. In order to improve the FFT approach, the undetected errors were closely examined. All the undetected errors are listed below. Each one is accompanied by reasons explaining why it was not detected by the test cases.

E3: In "textWdith := (r.right + r.left) div HorizUnit;", it should be (r.right - r.left). This error is hardly observable, even with very careful visual inspection. This is because "r.left" has a small value that is close to 0, as is normal for the left edge of an editing window. (In this case, r.left=3, r.right=250 and HorizUnit=8.) This error produces no difference (or failure) in the final outcome of the interactions (i.e. the content of the resultant text file), if the visual differences were visible to the unaided eyes during testing.

E4: "SetEnable(...)" was missing from Procedure OpenFile. This error actually exposed a missing predicate in the specification, concerning the maximum number of files allowed to be opened. The specification was then enhanced and testing exposed this error by opening a number of files to reach the maximum number of files.

E11: The problem is in closing editing windows without releasing the memory held by the text record. There are no quick ways to detect this kind of system problem, except by opening and closing a number of large files, until memory runs out. This problem would be uncovered by stress testing or system testing (where usage of system resources is monitored), rather than functional testing.

E13 Error in not setting "textDirty := FALSE;" in handling "cut" in DoEdit.
This error is hard to detect because the textDirty flag is set correctly in other

procedures which hides this error. The way to expose it is to perform a "cut" operation only, and then quit to see if the warning dialogue "Unsaved changes" will be displayed to indicate the state of the textDirty flag. The following sequence will detect this error :

F_start o F_select_text o F_cut o F_quit.

It is not desirable to modify the test cases for this purpose as it requires the tester to quit from ThinkEdit when testing has just started. Since the textDirty flag is set in a number of procedures; each would then require a quick quit to be tested. This error would only produce a failure when a user quits the editor after making some text selections and cuts, and nothing else. Effectively, the user has not lost any work (i.e. new text entries). The problem is observable in the lack of a warning about "Unsaved changes".

E20 "ShowSelect(TheWInfo)" is missing from Procedure HandleKey.
It is undetected because the window receiving the keyboard input (as pointed to by TheWinfo) is already the front window, even without ShowSelect.

E22u Incorrect "duplicate Filename" warning as "fName,dName,Null" were swapped in "ParamText(...);".
This can be detected by opening the same file a second time. The inability to detect this error has revealed predicates missing from the specification of F_open_file. The enhanced specification would allow detection of this error.

E23u Incorrect UntitledNum ("UntitledNum + 1" instead of -1 in Procedure DuplicateName).
This undetected error reveals another incompleteness in the specification for F_open_file, in not specifying predicates for UntitledNum. With the enhanced specification, this error can be detected by a sequence such as :
F_open_file(x) o F_open_open o F_open_file(x) o F_open_cancel o F_new_file.

E26u: Forgot to reset PenSize; "SetPenState(oldPenState)" is missing.
It is undetected, and has no observable effects, because the pen (with PenSize=3 left) was not used again in that procedure (FrameDItem). When the pen was used later in other procedures, it was then reset to other defaults by SetPort(...). It is fair to say that this error produces no failures in this case. Resetting PenSize is a kind of good programming practice that can only be checked by code inspection, and cannot be detected by a testing approach that uncovers errors by examining the results of program execution.

12.2.1 Reappraisal of results

Given the above analysis of undetected errors, the success rate of test cases (derived from specifications) in the exposure of errors is well above 80%. The reasons are :

- The 3 errors, which help to reveal incompleteness in the specification, will be detected as the specification is enhanced. (E23u, E22u, E4)
- There are 2 errors which would not be detected by this approach nor any other functional testing approach, only code inspection would uncover them. (E26u, E20)
- There are 3 errors that would remain undetectable (E13, E11, E3).

Five undetectable errors in a total of 40 gives a detection rate of 87.5%. If the two errors, which are only detectable by code inspection or analysis, are deducted, 92.5% of all functionally detectable errors are found. This is achieved by four relatively short test sequences, derived from specifications according to the criterion of 100% functional coverage. Even the initial 80% error detection rate is higher than the 61% reported in [Howden76]. (See section 2.3.) The higher detection rate is attributed to the use of formal specification and the observation that GUIs are more amendable to functional testing (see the next chapter). The work in [Howden76] was carried out prior to the advent of GUIs, and formal specifications were not used.

12.2.2 Other observations

The error seeding and debugging process was described in section 9.4. The following paragraphs restate and develop some related observations.

- Errors in window library calls are likely to produce visible symptoms, leading to their detection. Ordinary logical and typographical errors, in code statements other than window library calls, are harder to expose.

- T_state_predicates are very useful in assisting the detection of faults. Yet minor visual differences are hard to detect (e.g. E3). There is a vital balance between keeping the state primitives in WinSpec on a relatively high and comprehensible level, and its usefulness in exposing minor visual differences. The ability to detect functional differences is considered more important than minor visual differences. Perhaps the base line is that these minor visual differences should not hinder user interactions or affect the final outcome (i.e. the content of the text file being edited).

- Although a functional testing approach was followed, the process of injecting errors in the source code provided some insight into structural testing. There are cases (E13, E11 and E3) in which it was found that the errors were seeded in program paths which were untested by the test sequences. Specific combinations of functions are required to traverse the program paths containing these errors. Intuitively, a thorough path coverage in structural testing could be viewed as corresponding to a thorough functional testing that invokes all combinations of functions. An advantage of FFT is that the formal specification provides the source for both test inputs and test oracles. Structural testing uses the program source as a concrete basis for test data generation, but still requires some form of functional specification to serve as a test oracle.

12.3 Complementing functional testing with code coverage

Given the above observation, it was decided that some code coverage measurements should be used to evaluate the FFT approach. A very simple method of code instrumentation was employed to measure code coverage. Additional "check point" statements were inserted throughout the source code, at locations including:

- One check point at the entrance of each procedure or function.
- One check point at the exit of each procedure or function.
- For each "IF ... THEN ... ELSE ..." statement, one check point at the THEN clause, and one check point at the ELSE clause.
- Each item within a CASE statement is inserted with a check point.
- Each "WHILE ... DO" program block is given a check point.
- Each program loop of "REPEAT ... UNTIL ..." is also given a check point.

During execution of the test sequences, if any of these "check point" statements are executed, a flag is set in a matrix recording the check point traversed. At the end of the execution, a simple analysis program is run to reveal the percentage of check points traversed. The following table presents the result of check points exercised by test sequences TS1, TS2, TS3 and TS4.

Source file name	Total No of chk pts	No of chk pts traversed (%)	
Editor TopLevel	177	143	(87%)
Editor Utilities	85	59	(69%)
Editor Init	7	7	(100%)
Change font	17	0	(0%)
Show Edit	18	18	(100%)
Total	304	227	(74%)

Table 12.1 Result of code coverage measurements

The most outstanding observation from these results is that the routines in the "Change font" module were completely untested by the test sequences. It reveals the complete omission of the "Change font" functions in the specification. This reconfirms the general belief amongst researchers that functional and structural tests should be used to complement one another. It would be straightforward task to improve the specification and test cases, and consequently the results, by including the "Change font" functions. However, it was decided to publish the results obtained from the first experiment, as they are more revealing than otherwise.

In the following bar chart, the code coverage measurements are shown in percentage values, together with the earlier results of error detection. Allowing for inaccuracies in these experiments, the 70% to 80% achievement of both error detection and code coverage is encouraging. Recall that the test sequences for ThinkEdit are relatively short; they only require about 10 to 15 minutes for a human tester to perform manually. These results are higher than those reported in [OU84], where functional (non-formal specification based) test cases achieved 44.5% statement coverage and 35% branch coverage.

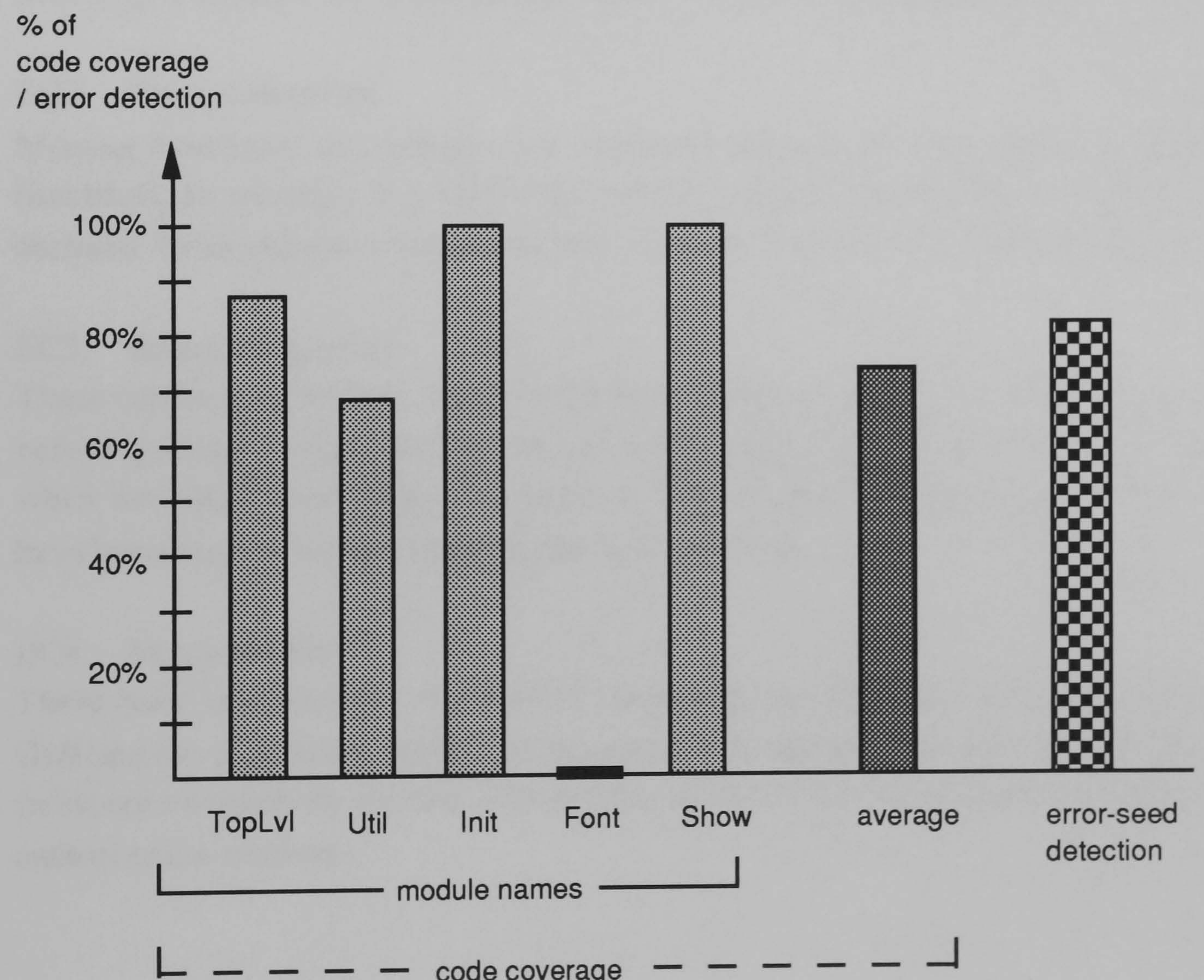


Figure 12.1 A bar chart showing percentages of code coverage and error detection

12.4 Common errors in GUIs

It has been suggested by [Myers79] that some people are good testers because of their experience or intuitive feeling about where errors may lie. There is also research in the more recently proposed fault / error based approach [Morell87] to software testing. From the GUI testing experiments, faults in GUIs are classified into error classes. The error sources are found to be largely related to the understanding of window library routine functions, sequences and parameters of routine calls, in addition to the usual logical and typographical errors. Four different error classes (EC1 to EC4) are explained below.

EC1: Visual errors
Errors in the display of objects, where objects are missing or inappropriately displayed. These can be caused by incomplete visual design, or errors in translating design into code (e.g. a typing error in the integer values of coordinates of objects).

EC2: Functional errors
Missing functions; an example is a command button with no functions. Inadequate functions; an example is a username textBox without text editing functions. It was declared by mistake as a "static text box", instead of an "edit text box", in the program.

EC3: Sequencing errors
These can be seen as logic errors in the flow of interactions. For example, a "save file before quitting ?" dialogue box should not be sequenced indiscriminately to appear when the "quit" menu option is selected. The dialogue is unnecessary if no changes have been made to the file since the last save operation.

EC4: Message errors
These refer to a kind of "cross-wired" problems in switching messages between the GUI and the underlying application program. For example, the username and password fields are swapped, by mistake, in messages sent from the Logon interface to the system authorization routines.

The error classes above are developed subjectively. It is, of course, possible to categorize errors in other ways. Nevertheless, the following list gives the classification of the 10 errors in the logon interface. (Also see sections 7.4 and 7.5.)

EC1: Visual errors

E1) "OK" button is placed in a different location than that which is specified in the WinSTD (error in coding coordinates).

E2) Username field is misplaced (i.e. out of alignment).

E3) Username and password fields are swapped physically (i..e. the label "username" is wrongly attached to the password textBox).

EC2: Functional errors

E4) Username field has no text entry function

E5) "Reset" command button has no function.

E6) "OK" button has no function.

E7) <tab> and <cr> at the username and password textBoxes have no function.

EC3: Sequencing errors

E8) Terminal window is displayed when logon is invalid (wrong display sequence in the GUI).

E9) "Logon Failure" dialogue box is displayed when logon is valid.

EC4: Message errors

E10) Username and password are swapped internally (i.e. keyboard inputs in the username textBox are sent in messages to the application as the password).

The 40 seeded errors, used in the ThinkEdit testing experiment, can also be categorized according to the error classes discussed above. The following table gives the distribution of the ThinkEdit errors. (A complete list of the 40 errors can be found in Table 9.1 in section 9.5.)

Error Classes	Error Nos. in classes	Total
EC1 Visual errors	E2, E3, E6, E7, E23, E25, E28, E29, E30, E31, E32, E37.	12
EC2 Functional errors	E1, E14, E15, E16, E17, E18, E19, E33, E34, E35, E36, E38, E39, E40.	14
EC3 Sequencing errors	E5, E9, E10, E12, E13, E20, E21, E26.	8
EC4 Message errors	E4, E8, E11, E22, E24, E27.	6

Table 12.2 Distribution of ThinkEdit errors according to error classes

12.5 Considerations on design, specification and testing

The use of WinSTDs and WinSpec notations is not limited to testing. They can also be used as design tools. A WinSpec specification can be used to check the *reachability* ([Nicholl90], see Glossary) of objects and functions, and the complexity and *completeness* [Jalote89] of a design. A WinSTD is useful in visualising display objects and the flow of interactions, for early user training and feedback. The user interface model used (see Figure 3.1) encourages dialogue separation and modular programming.

12.5.1 Testability as a design factor

The design concept of dialogue separation is important to the development of user interfaces. Yet it has not been considered from the point of view of software testing. There is a current debate between supporters of UIMS and those supporting user interface toolkits. An important factor in this debate is that UIMS encourages dialogue separation whilst toolkits make it possible for interface processing to be mixed together with computations.

The research conducted in this thesis reveals that dialogue separation would indeed improve the testability, and thus the quality of user interfaces. This is exemplified in the case of X-mail, where dialogue separation makes it possible to test the user interface in the absence of the application. However, there are also examples such as ThinkEdit, where the main application functions lie close to that of the user interface, in handling screen display and user inputs. Dialogue separation is less important in these cases.

12.5.2 The value of formal specifications in software testing

A formal functional specification readily lends itself to test generation, as has been expounded by this thesis. It has the additional validation value of being implementation independent, as specifications are generally not written in a programming language. A formal functional testing approach has the advantage that test oracles as well as test inputs are obtainable from the specification. A structural testing approach would still require some form of (perhaps informal) functional specification for a test oracle.

There are myths concerning formal approaches, such as "Myth 1: Formal methods can guarantee that software is perfect" [Hall90]. However, formal methods are fallible. The specification for ThinkEdit was found to be incomplete, with missing functions in at least three places. This does not undermine the value of formal specification. The value of formal specifications lies in the use of precise, unambiguous notations and a sound mathematical base to enable software engineers to reason about the functional aspects of computer programs, more vigorously than the unaided mind. The experience with the X-Mail user interface was that the process of producing a formal specification has itself uncovered a number of design and implementation errors. There is another myth that "formal methods are unacceptable to users" [Hall90]. As discussed in section 8.4.7, a contrast between specification and code shows that the specification is easier to understand and follow. This is due to both the special notations used, and the power of abstraction that a specification can offer.

12.6 Justifications for the case studies

This section gives an analysis to justify that the small number of case studies pursued in this thesis are representative of graphical user interfaces in general. Due to resource and time constraints, testing experiments were mainly conducted on the Logon and ThinkEdit interfaces. These two user interfaces were chosen for three main reasons :

- They cover most of the basic interaction components commonly used in GUIs (such as windows, menus, icons and dialogue boxes, as discussed in section 3.1).

- The logon interface and text processing are probably amongst the most commonly used types of graphical user interfaces.

- They are also standard examples used in a number of publications concerning user interfaces and specification. By adhering to widely used examples, it is possible to contrast the different approaches to specification of such systems.

Different versions of the logon user interfaces have been the subject of research in [Jacob83, 86], [Green85, 86], [Alexander86] and [Marshall86]. The logon (or login) interface is used in a number of research publications because it is a small and self-contained example, with features familiar to most computer users.

Specifications of text editors are pursued in [Sufrin82] and [Chi85]. Window-based text editors are common in GUI environments, yet their interaction styles are beyond those of icons and menus. An interactive window editor introduces the complications of formatting, highlighting and scrolling of text within the editing window. The specification has to capture both the visible layout of text in the window and the invisible structure of the file being edited. These are the reasons for choosing ThinkEdit as one of the main case studies in this thesis.

Another reason for the small number of testing experiments is the belief that test case generation can be carried out systematically (see Chapters 6 to 9), provided that formal specifications of the functions are available. In order to investigate if the WinSTD and WinSpec approach to specification is generally usable for different classes of graphical user interfaces, the specification of three other GUIs (Xmail, the WinSTD editor and a JRR tool) were examined in chapter 10. The three GUIs belong to different classes of user interfaces, since their respective applications and styles of interaction are different. X-Mail, a user interface for a mail program, is a typical user interface. The WinSTD editor is special as it belongs to the class of graphics editors. The JRR tool is an example of a systems tool which has a very simple user interface. In all of these cases, the WinSTD and WinSpec specification approach is adequate.

From the view of popular usage, logon interfaces, text editors and mail interfaces are probably the most widely used of user interfaces. Considering the styles of user interface interaction, the text and graph editing (ThinkEdit and WinSTD editor), and the use of icons, menus, windows and dialogue boxes (Logon, X-Mail, JRR tool) are representative of current graphical user interfaces. Viewing from the perspective of the communication model between the user interface and underlying application, the case studies cover a wide range, from the "micro-communication" of ThinkEdit, to the "macro-communication" of the JRR tool. (See section 3.3 for descriptions of macro- and micro-communication.)

A further justification for this testing method is that it has been found to be effective in uncovering the commonly occurring classes of faults: visual, functional, sequencing and message errors (as analysed in section 12.4). Since these common classes of faults are not specific to Logon or ThinkEdit, this testing approach will be useful in testing other graphical user interfaces.

Chapter 13

Conclusions

"A number of authors have suggested methods for functional testing, and there are also a substantial number of systems based on this approach. The fundamental idea is that functions be identified within the computer software or elsewhere, and in order to test the program, each of these functions must be tested over appropriately selected test cases. We shall see that the problem is to approach the generation of the functions and test cases systematically, and eventually automatically."

From [White87] in [Yovits87]

This thesis has undertaken an original investigation and analysis of the problems concerning the validation of graphical user interfaces. A systematic approach has been developed for the identification and specification of functions, and the generation of test cases for GUI software.

State Transition Diagrams (STDs), based on the theoretical concept of a Finite State Machine (FSM), are useful for describing the flow of interactions in a GUI. A WinSTD is also useful for the enumeration and visual inspection of display objects for testing. The WinSpec notations make it possible to model and formalize user interactions into functions and sequences of functions. A WinSpec specification presents the required user inputs and expected visual outputs and state changes, in terms of state predicates for each interaction function. When functions and their relationship are specified formally, graph theoretic algorithm such as Euler tour, postman's tour, Hamilton circuit and the travelling salesman's tour can be used for identifying test coverage.

A WinSpec specification is not intended for prototyping, and thus is independent of implementation. This allows both design and implementation errors to be uncovered by tests derived from specifications. This new validation approach is explored in the specification and testing of a number of user interfaces. These include a logon interface and a window editor. A 100% function coverage criterion is used, producing relatively short test sequences that can be manually executed in about 10 minutes. The test sequences derived from formal specifications are evaluated with seeded errors and code coverage measurements. The results obtained show a 80% success rate in the detection of seeded errors and a 70% code coverage. Some knowledge about common GUI faults / errors is gained.

This thesis has argued that a functional testing approach is suitable for graphical user interfaces, and concludes that the derivation of test cases from formal specification is an important step towards automation. Attendees at a recent conference (HICSS-24), from both industrial and academic backgrounds, were in agreement that this should be the future strategy [Andreas91], [Birjandi91], [Yip91a].

One distinctive feature of user interfaces in general, and of GUIs in particular, is the relatively direct, almost one-to-one relationship between user inputs and observable outputs. It is exemplified by visual feedback and direct manipulation of display objects. For the above reason, the derivation of required inputs and expected outputs for GUIs is relatively easy. This may have contributed to the successful results of the FFT experiments. Test derivation may be more difficult for other types of software where outputs (or results) are more dependent on values of internal program storages [Hall91b] which are hidden from testers.

Contrary to the myths about formal specifications [Hall90], the experience of formal specifications for GUIs has not been prohibitively difficult or mathematical. The process of making a specification does not generally demand skills other than those required for writing programs. For instance, a total of more than 40 interaction functions has been specified for ThinkEdit. The mathematical skill required was not beyond that of first order predicate calculus used in state predicates. Logical or conditional statements are also used in programming languages. One difference is that predicates are used almost exclusively in a model-based specification language like WinSpec. In contrast, many other constructs, such as assignment statements and routine calls, are frequently used in procedural programming languages. The kind of arithmetic used in specifications, which is usually application dependent, is largely similar to that used in programs. For instance, simple arithmetic is used to specify the number of text lines to be scrolled, by multiplying the total number of text lines by the fraction of the length of the scroll bar that the slider has been moved. Perhaps the most demanding portion of the specification process is in identifying the type of parameters (or variable) to be used in modelling the essential properties of functions for testing purposes. An example is the use of the two pointer variables, selStart and selEnd, to represent the start and end of a piece of text selection. This kind of modelling is similar to that of working out the required data structure in program designs.

The above justifications for the use of formal specification are based on the assumption that a suitable formal specification language is available. Naturally, it is difficult to write programs in a language at the same time when the language is being developed [Wirth71b]. It has been found that writing WinSpec specifications has become straightforward once a basic set of constructs have been tried and modified in earlier specification attempts. The foundation work of pinning WinSpec on predicate logic and set theory, recognizing the temporal order of I/Os, and the adoption of application messages and visual state primitives to model GUIs is considerably harder than writing WinSpec specifications.

Another value of the FFT approach is in the possibility of expanding WinSpec by introducing new object types and state primitives. This has been demonstrated with user interfaces employing different interaction objects and styles. The flexibility for expansion is vital for adapting to new ideas in graphical user interfaces. This is also important in allowing WinSpec to be used for different window systems. Window systems provide very similar features, even across different hardware platforms. Consequently, in circumstances where a popular application is ported to a different host computer environment, for example from a PC to a workstation, WinSpec specifications and test cases will be largely reusable. The flexibility and reusability of both the specification and test data are valuable to the software maintenance process, the most expensive phase of the software life cycle.

13.1 Assessment: achievements

The main criteria for success were laid down in Chapter 1. These are now examined to judge if the research has been successful.

- "The specification approach and notation should give a precise and comprehensible description of GUI functions ..."
  A novel specification method has been developed, based on WinSpec notations and WinSTDs. The specification approach has improved the understanding of GUIs, as design and implementation errors were found during the specification process, prior to testing. In section 8.4.6, it was argued that WinSpec specifications are more comprehensible than the source code.

- "The approach should be applicable to a wide range of user interfaces possibly on different hardware platforms and window systems."

This is accomplished in the case studies of a number of different classes of GUIs, on more than one hardware platform. These include a logon interface, a window editor, a mail user interface, a graphics editor and a JRR tool. These interfaces employ different display objects and interaction styles. It is possible to expand the WinSpec notations to introduce new object types and state primitives, when necessary.

- "The specification, once written, should lend itself to the systematic generation of test cases."
  This has been achieved and presented in detail for both the Logon and ThinkEdit GUIs. Test design has become a systematic way of deriving test inputs and test oracles from the WinSpec specification. Test sequences are formed by selecting and joining functions of matching To_state and From_state, according to the required coverage criteria.

- "A low success rate in error detection, function or code coverage, should call for improvements in the specification method and notations."
  The test sequences derived from formal specifications are evaluated with seeded errors and code coverage measurements. The results obtained show a 80% success rate in the detection of seeded errors and a 70% code coverage. These compare favourably with one study [OU84], which reported functional test cases (non-formal specification based) of achieving 44.5% statement coverage and 35% branch coverage. The error detection rate of traditional (i.e. not formal specification based) functional testing is 61% as reported in [Howden76]. (See section 2.3.) The use of formal specifications, the nature of GUIs, and the relatively small size of ThinkEdit may have contributed to the higher success rates.

13.2 Assessment: criticisms

The original intention, to capture all interaction functions and state transitions in a WinSTD, has been found to be impractical. The WinSpec notations are subject to a number of restrictions and assumptions as discussed in section 5.8. These are summarized below :

- The existence of a mechanism to serialize inputs, that "type ahead" and "mouse ahead" inputs are blocked (queued or discarded), is assumed.

- It is not possible to specify effects of other user interfaces that may become concurrent with the GUI being specified.

- Predicates in the T_state_predicates only state the changes in display objects that are consequences of the functions being specified. Any unspecified visual changes on the screen (e.g. time clock) are considered irrelevant.

- Although the mouse input device (pointer and button) is used exclusively in specifications, it does not imply that other devices cannot be used in graphical user interfaces.

The Formal Functional Testing (FFT) approach has turned the demanding process of test design into straightforward derivation from specifications. However, it can be argued that it has merely moved the intellectual effort required from the design of test cases to that of writing formal specifications. It is true that some skills are required to produce formal specifications. Perhaps the most important skills are familiarity with the specification language, and understanding of the application being specified.

It is probably true that even in an ad hoc manner, a human tester does attempt to identify display objects and functions before carrying out testing. The difference that FFT has made is in adopting a set of precise and unambiguous notations, through which a human tester can identify objects and functions more properly and formally. These formal notations also enable test designs to be communicated amongst and cross-checked by software engineers.

Some information outside the WinSpec specification is still required. For example, valid pairs of usernames and passwords are required to test the Logon interface. This information is not part of the user interface specification.

Formal methods are fallible. The specification for ThinkEdit is found to be incomplete, with missing functions in at least three places. All the "font" functions were found to be untested, when cross-checked with code coverage evaluation. However, this does not undermine the value of formal specification. The value of formal specification lies in the use of precise, unambiguous notations and semantics with a sound mathematical base, enabling software engineers to reason about the functional aspects of computer programs. It is important to realize that a formal specification can be modified, if necessary, according to the findings of the testing process. There is a most recent and pragmatic advocacy that specification should also be considered as an output, as well as an input to the testing process [Hetzel91].

Despite case studies and exploration of automation issues, a complete tool implementation is beyond the resource constraints of this thesis. The testing process has been demonstrated (in chapters 6 to 11), and prototypes have been explored, but a complete implementation has not been possible because of time constraints.

13.3 Future directions

Following on from the criticism above, a complete tool-implementation and further evaluation will be the main directions for future research. Once the test generation process is completely automated, it will be relatively less time-consuming to conduct testing experiments, in order to obtain further results.

Further experiments, on the testing of user interfaces through dialogue separation, should be pursued. When user interfaces are increasingly designed with dialogue separation in mind, their testability will be improved. The checking of textual messages between an application and its GUI is easier and more reliable than the visual verification of screen outputs.

Perhaps the most important future direction, in concluding this thesis, is the need to carry out further research towards discovering the value of formal specifications to software testing. It would be interesting to investigate how code coverage, achieved by functional test cases, can be improved. Two answers are foreseeable. One is the refinement of the specification to further reduce incompleteness. The other is the testing of combinations of functions, as explained in the last observation in section 12.2.2.

Research on the wider use of FFT in testing programs other than GUIs should be encouraged. The experience gained in this thesis is that functions of a class of software, GUIs in this case, are similar across different implementations. Perhaps research into the formal specification of functions of a number of common applications, such as database access, accounts and ledgers, real time process control, modelling and simulation programs can be pursued. The efforts required by the initial specifications may be justified by the long term view of the reusability of these formal specifications. Would it be a vision or a dream to foresee that one day a box labelled "Formal specification and test cases for databases - universal, hardware and language independent" can be purchased off the shelf ?

Use of formal specifications in software testing should be considered alongside program proving and mathematical proof approaches [Young91]. It is also important that a formal specification should not be used just as an input to the testing process. Knowing that formal methods are fallible, a formal specification should also be improved as an outcome of the testing process.

References

[Abbott86] J., "Software Testing Techniques" , The National Computing Centre Ltd., 1986.

[Abowd90] G.D., Harrison M.D., "On a Constructive Approach to Applying Formal Methods in HCI", Dept. of Computer Science, University of York, Report YCS151, May 1990.

[Aho88] A.V., Dahbura A.T., Lee D., and Uyar M.U., "An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours", in Proc. 8th IFIP WG6.1 Workshop on "Protocol Specification, Testing and Verification VIII", Elsevier Science Publishers (North-Holland).

[Alagar89] V.S., "Fundamentals of Computing, Theory and Practice", Prentice-Hall International Inc., 1989.

[Alexander86] H., "Formally-Based Tools and Techniques for Human-Computer Dialogues", PhD. Thesis, Stirling University 1986.

[Anderson87] Publishing Company, Special Report on "Major Vendors Agree on Window Standard", The Anderson Report, P5-6, February 1987.

[Andreas91] J.R., "Automated Regression Testing of Graphical User Interface Based Applications", in [HICSS91] .

[Apple85] Computers Inc., "Inside Macintosh", Vol 1-5, Addison-Wesley 1985.

[Arthur87] J.D., "Towards a Formal Specification of Menu-based systems", The Journal of System and Software 1987.

[Atly84] J.L., "Use of Path Algebras in an Interactive Adaptive Dialogue System", Proc. INTERACT'84, p351-354, North-Holland 1985.

[Birjandi91] A., Sydorowicz S., "Validation of Motif Graphical User Interface Widget Set", in [HICSS91] .

[Bobrow86] D.G., "Expert Systems: Perils and Promise", Communications of the ACM, p880-894, Sept. 1986.

[Boehm88] B.W., "A Spiral Model of Software Development and Enhancement", Computer, p61-72, May 1988.

[Brady77] J.M., "The Theory of Computer Science, A Programming Approach", Chapman and Hall Ltd., 1977.

[Budd78] T.A., DeMillo R., Lipton R.J., Sayward F.G., "The design of a prototype mutation system for program testing", Proc. ACM Nat. Comput. Conf., p623-627, 1978.

[CAPBAK90], "CAPBAK/X - Test Capture / Replay for X Windows, Technical Specifications", Software Research Inc., San Francisco, USA, May 1990.

[Casey82] B.E., Dasarathy B., "Modelling and Validating the Man-Machine Interface", GTE Labs., Software-Practice and Experience 12(6) p558-569, 1982.

[Carre79] B.A., "Graphs and Networks", Clarendon Press, Oxford 1979.

[Chernicoff88] S., "Macintosh Revealed, Vol. 2: Programming with the Toolbox", Hayden Books, New York 1988.

[Chi85] U.I., "Formal Specification of User Interface: A Comparison and Evaluation of 4 Axiomatic Approaches", IEEE Trans. Software Eng., 11(8), p671-685, 1985.

[Choquet86] N., "Test Data Generation using a Prolog with constraints", in [TAV86], p132-141.

[CMU89] "Serpent Overview", Serpent UIMS user manual, SEI Carnegie Mellon University, August 1989.

[Cockton86] G., "Where do we draw the line?", Proc. of HCI'86 Conf., Harrison M., Monk A.F. (eds), Cambridge University Press.

[Coutu90] D., "Automating X Window System testing by User Synthesis", Digital Equipment Corp., X Technical Conference, MIT Press, Jan 1990. (Abstract only).

[Coward88a] P.D., "A review of software testing", Information and Software Technology , Vol 30, p 189-198, Apr 1988.

[Coward88b] P.D., "Symbolic execution systems - a review", Software Engineering Journal, p229-239, Nov. 1988.

[Crabb89] D., "A Macintosh Retrospective", Byte, p143-146, March 1989.

[Cronin87] P.J., Robson D.J., "Confirmation of some random testing results", University Computing, 9, p153-156, 1987.

[CSR85], "Software : requirements, specification and testing" - Proceedings of CSR Workshop, University of East Anglia, April 10-12, 1984.

[Dijkstra76] E.W., "A Discipline of Programming", Prentice-Hall 1976.

[Duce86] D.A., Fielding E.V.C., "Towards a formal specification of the GKS output primitives", Proc. Eurographics '86, p307-324, 1986.

[Duran84] J.W., Natfos S.C., "An Evaluation of Random Testing", IEEE Trans. Soft. Eng. 10(4), p438-444, July 1984.

[Durham91] University of Durham, Proc. European Workshop of Software Maintenance, 1991.

[Ehrlich89] K. (Sun Micro-systems Inc.), et al., "Incorporating usability studies & Interface design into Software development" , Proc. USENIX Technical Conf., Summer 1989.

[Elmendorf73] W.R., "Cause-Effect Graphs in Functional Testing", TR-00.2487, IBM Systems Development Division, Poughkeepsie, N.Y., 1973.

[Elverex89], "Evaluator" - Sales Literature, in Personal Computer Magazine, August 1989 .

[Fiume89] Eugene, "Towards Realistic Formal Specifications For Non-Trival Graphical Objects", in Proc. EUROGRAPHICS'89, p289-299, Elsevier Science Publishers, 1989.

[Floyd67] R.W., "Assigning meanings to programs", Proc. Symposium Applied Math. Vol 19, p19-32, American Math. Society 1967.

[Galton87] A.P.(Ed), "Temporal logics and their applications", Academic Press Ltd., 1987.

[Gannon81] J., McMullin,P., Hamlet,R., "Data-abstraction implementation, specification, and testing", ACM Trans. Program. Lang. and Syst., 3(3), p211-223, July, 1981.

[Gaudel88] M-C., Marre B., "Generation of Test Data From Algebraic Specifications", in TAV88, p138-139.

[Gehani86] N., McGettrick A.D. (eds), "Software Specification Techniques", Addison-Wesley 1986.

[Glass79] R.L., "Software Reliability Guide book", Prentice-Hall, New Jersey, 1979.

[Goldberg83] A., Robson D., "SmallTalk-80, The language and its implementation", Addison-Wesley, Xerox 1983.

[Goodenough75] J.B., Gerhart S.L., "Towards a Theory of Test Data Selection", IEEE Trans. Soft. Eng., 1(2), p156-173, June 1975.

[Gourlay81] J.S., "Theory of testing computer programs", Ph.D. dissertation, Dept. of Computer and Communication Sciences, Univ. of Michigan, 1981.

[Gourlay83] J.S., "A Mathematical Framework for the Investigation of Testing", IEEE Transaction on Software Engineering, 9(6), p666-709, Nov 1983.

[Gray88] P.D., et al, "Dynamic reconfigurability for fast prototyping of user interfaces", Soft. Eng. Journal, p257-262, Nov 1988.

[Graham90] D., "Computer Aided Software Testing - CAST Report", Unicom Seminars, UK, 1990.

[Green85] M., "The University of Alberta UIMS", Proc. SIG GRAPH 85, p205-213, ACM New York, 1985.

[Green86] M., "A Survey of Three Dialogue Models", ACM Trans. Graphics, p244-275, July 1986.

[Hall87] Frank, "XRLIB: An X Windows Toolkit", p254-263, in [Pacific87].

[Hall88] P.A.V., "Towards testing with respect to formal specification", Proc. 2nd UK IEE/BCS Conf. of Software Engineering, p159-163, July 1988.

[Hall90] Anthony, "Seven Myths of Formal methods", p11-19, IEEE Software, Sept. 1990.

[Hall91a] P.A.V., "Relationship between specifications and testing", p47-52, Information and Software Technology, Jan/Feb 1991.

[Hall91b] P.A.V., Hierons R., "Formal Methods and Testing", Technical Report No.91/16, Computing Dept., The Open University, Aug. 1991.

[Hantler76] S.L., King J.C., "An Introduction to Proving the Correctness of Programs", ACM Computing Surveys, p331-353, Sept 1976.

[Harel88] D., "On Visual Formalisms", Comms. of ACM, 31(5), p514-531, May 1988.

[Harel90] D., et al, "STATEMATE : A Working Environment for the Development of Complex Reactive Systems", IEEE Trans. Soft. Eng., 16(4), p403-413, April 1990.

[Harrison90] M., Thimbleby H. (eds), "Formal Methods in Human-Computer Interaction", Cambridge Univ. Press 1990.

[Harrison91] M.D., Abowd G.D., "Formal Methods in Human Computer Interaction: a Tutorial", Dept. of Computer Science, University of York, Report YCS155, Mar 1991.

[Hartson89] R.,"User-Interface Management Control and Communication", IEEE Software, Jan 1989, p62-70.

[Hekmatpour88] S., Ince D., "Software Prototyping, Formal Methods and VDM", Addison-Wesley 1988.

[Hetzel91] B., Gelperin D., "Software Testing, some troubling issues", p22-27, American Programmer, April 1991.

[HICSS91] , Proc. Hawaii International Conference on System Sciences 1991, session on Graphical User Interface Validation, p89-123, in vol.2 of Proc.

[Hoare69] C.A.R., "An Axiomatic Basis for Computer Programming", Comms of the ACM, 12(10), p576-583, Oct. 1969.

[Hoare85] C.A.R., "Communication Sequential Processes", Prentice-Hall 1985.

[Hopgood86] F.R.A., et al, "Introduction to GKS", Academic Press, 2nd ed., 1986.

[Howden76] W.E., "Reliability of the Path Analysis Testing Strategy", IEEE Trans. Soft. Eng., 2(3), p208-214, Sept. 1976.

[Howden77] W.E., "Symbolic testing and the DISSECT symbolic execution system", IEEE Trans. Soft. Eng., 3(4), p266-278, April 1977.

[Howden78] W.E., "A survey of dynamic analysis methods", in [Tutorial81], p209-231.

[Howden81] W.E., “Completeness Criteria for Testing Elementary Program Functions”, Proc. 5th Intl. Conf. on Soft. Eng., p235-243, 1981.

[Howden87] W.E., "Functional Program Testing & Analysis", McGraw-Hill 1987.

[Hsieh71] E.P., "Checking experiments for sequential machines", IEEE Trans. Comput., 20(10), p1152-1166, Oct., 1971.

[IEEE83], ANSI/IEEE Std 729-1983, Standard glossary of Software Engineering terminology, 1983.

[Ince84] D., Hekmatpour S., "An evaluation of some black-box testing methods", Technical Report No 84/7, Computing Discipline, Faculty of Mathematics, Open University.

[Islam89] N., Ingoglia J.P., "Testing Window Systems", Proc. 28th Annual Technical Symposium "Interfaces : System and People working together", Washington D.C. ACM Chapter.

[Jacob83] R.J.K., "Using formal specifications in the design of HCI", Comms. of ACM, 26(4), p259-264, April 1983

[Jacob86] R.J.K., "A Specification Language for Direct Manipulation User-interfaces", ACM Transactions on Graphics, p283-317, Oct. 1986.

[Jalote89] P., “Testing the Completeness of Specifications”, IEEE Trans. Soft. Eng. 15(5), p526-531, May 1989.

[Jamison90] A., "Enhancing the Input Synthesis Extension with Xtrap", Digital Equipment Corp., Proc. X Technical Conference, Jan 1990. (Abstract only).

[Johnson81] S.C., “Yacc - Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep. No.32.”, Bell Laboratories: Murray Hill, New Jersey, 1981.

[Johnson87] M.A., "Automated Testing of User Interfaces", p285-293, Proc. Pacific North West Software Quality Conference, 1987.

[Jones85] C.B., “Specification, Verification and Testing in Software Development”, in [CSR85], p1-13.

[Jones90] C.B., "Systematic Software Development Using VDM", 2nd edition, Prentice-Hall 1990.

[Kernighan84] B.W., Pike R., “The UNIX Programming Environment”, Prentice-Hall 1984.

[King76] J.C., "Symbolic execution and program testing", Comms. of ACM, 19(7), p385-394, July 1976.

[Kuan62] M.K., “Graphic programming using odd or even points”, Chinese Math., vol.1, p273-277, 1962.

[Leach83] D.M., M.R.Paige, and J.E.Satko, "AutoTester: A Testing Methodology for Interactive User Environments", Wang Laboratories; Software Reliability

Engineering Group. IEEE CHI, p143 - 147, August 1983.

[Lee90] Ed, "User-Interface Development Tools", IEEE Software, p31-36, May 1990.

[Leek81] M.E., "Lex - A Lexical Analyser Generator, Comp. Sci. Tech. Rep. No.39.", Bell Laboratories: Murray Hill, New Jersey, 1981.

[Leler89] W., "PIX, the latest NeWS", Proc. IEEE COMPCON Spring'89, p239-242, Feb. 1989.

[Lientz80] B.P., Swanson E.B., "Software Maintenance Management", Addison-Wesley, London, 1980.

[Lewis89R] R. and D.W.Beck (BTRL, UK) , J.Hartmann and D.J.Robson (Durham University), "ASSAY - A Tool To Support Regression Testing", Published in Procs. of 2nd European Software Engineering Conference, Sept. 1989.

[Lewis89T] T.G., et al, "Prototypes from Standard User Interface Management Systems", IEEE Computer, p51-60, May 1989.

[Linton89] M.A., "Composing User Interfaces with InterViews", IEEE Computer, p8-22, Feb 1989.

[Liskov75] B.H., Zilles S.N., "Specification Techniques for Data Abstractions", IEEE Trans. Software Eng., 1(1), p7-19, March 1975.

[Liskov79] B.H., Berzins V., "An Appraisal of Program Specifications", reprinted in p3-23, [Gehani86].

[Liskov86] B., Guttag J., "Abstraction and Specification in Program Development", MIT Press, 1986.

[Loo88] P.S., Tsai W.K., "Random testing revisited", Information and software technology, 30(7), Sept. 1988.

[Malhortra89] Anil, "Through the X Window", Unix Systems, p30-32, February 1989.

[Mallgren82] W.R., "Formal specification of interactive graphics programming languages", PhD. dissertation, Univ. Washington, Seattle, 1982.

[Marshall85] S.L., "A Formal specification of line representations on graphics devices", in Lecture notes in Computer Science - 186, p129-147, Spring Verlag, 1985.

[Marshall86] S.L., "A Formal Description Method for User Interfaces", PhD. thesis, University of Manchester 1986.

[Maurer83] M.E., "Full-screen testing of interactive applications", IBM Systems Journal, 22(3), p246-261, 1983.

[McMullin83] P.R., Gannon J.D., "Combining Testing with Formal Specifications: A Case Study", IEEE Trans. Soft. Eng., 9(3), May 1983.

[Microsoft88] Corporation, "AUTO MAC III, Macro Recorder" Reference Manual, 1988.

[Minieka78] Edward, "Optimization Algorithms for Networks and Graphs", Marcel Dekker Inc., New York, 1978.

[MIT89], documents in X11R4 distribution tape, MIT 1989.

[MIT90], Proc. 4th X Technical Conf., MIT Press, Jan 1990.

[Morell87] L.J., "A Model for assessing Code-based Techniques", Proc. Pacific North-west Software Quality Conference, p309-325, Oct 1987.

[Morell88] L.J., "Unit Testing and Analysis", SEI Curriculum module SEI-CM-9-1.1, SEI, Carnegie Mellon University, Dec 1988.

[Myers88] B.A., "A Taxonomy of Window Manager User Interfaces", IEEE Computer Graphics and Applications, Page 79-109, Sept. 1988.

[Myers89] B.A., "User-Interface Tools: Introduction and Survey", IEEE Software, p15-23, Jan 1989.

[Myers79] G.J., "Art of Software Testing", John Wiley & Sons 1979 .

[Narayana90] K.T., Dharap S., "Formal Specification of a Look Manager", IEEE Trans. Software Engineering, 16(9), p1089-1103, Sept. 1990.

[Nicholl90] R.A., "Unreachable states in model-oriented specifications", IEEE Trans. Soft. Eng., 16(4), p472-477, April 1990.

[North90] N.D., "Automatic Test Generation for the Triangle Problem", NPL Report DITC 161/90, National Physical Laboratory, Crown copyright, Feb 1990.

[Ostrand84] T.J., Weyuker E.J., "Collecting and Categorizing Software Error Data in an Industrial Environment", J.Syst. and Software, 4(11), p289-300, Nov. 1984.

[OSU89] Delph C., Whiltmore S., "Oregon Speedcode Universe" user manual., Oregon State University, 1989.

[OU84] Open University, Software Engineering course material, Unit 7, PMT600, Volume 7, p8-9, Open University Press, 1984.

[Pacific87], Proc. 5th Annual Pacific NorthWest Software Quality Conf., Oct 1987, Portland, Oregon.

[Parnas69] D.L., "On the use of transition diagrams in the design of a user interface for an interactive computer system", in Proc. 24th National ACM Conference, p379-385, 1969.

[PEI90] Programming Environment Inc., "T - Test Method and Tool", sales literature, 4043 State Highway 33, Tinton Falls, NJ 07753, USA, 1990.

[Petzold89] Charles, "Programming The OS/2 Presentation Manager", Microsoft

Press, Washington 1989.

[Pfaff85] G.E. (ed), "User Interface Management System" (Proceedings of Workshop on UIMS, Seeheim, Germany, Nov. 1983), Springer-Verlag 1985.

[Prime88] M., "User Interface Management Systems - a current product review", Report RAL-88-028, Rutherford Appleton Laboratory, 1988.

[Purvis90] J.B., "The use of LOTOS for the specification of graphics software", Technical Report CSTR-90-5, Dept. of Computer Science, Brunel University, July 1990.

[Richardson89] D.J., O'Malley O., Tittle C., "Approaches to Specification-Based Testing", p86-96, in TAV89.

[Roper87a] R.M.F., Smith P., "A Software tool for testing JSP designed programs", Soft. Eng. Journal 2(2), p46-52, Mar 1987.

[Roper87b] R.M.F., "The derivation of a methodology, with supporting software aids, for testing structured data processing programs", PhD Thesis, Sunderland Polytechnic, UK, 1987.

[Roper90] R.M.F., "The Automatic Generation of Test Cases", in [Wolverhampton90], p43-56.

[Rosson87] M.B., S.Maass, W.A.Kellogg, "Designing for Designers: An analysis of Design and Practices in the Real World" Proc. SIGCHI+GI 87, ACM, P137-142, New York, 1987.

[Royce70] W.W., "Managing the Development of Large Software Systems: Concepts and Techniques", Proc. Wescon, Aug. 1970. Also available in Proc. ICSE 9, Computer Society Press, 1987.

[Scheifler86] R.W. , Gettys J., "The X Window System", ACM Transactions on Graphics, Vol. 5, No. 2, p79-109, April 1986.

[Schmucker86] K.J., "MacApp An Application Framework", BYTE, p189-193, Aug 1986.

[Schreiner86] A.T., H.G.Friedman Jr., "Introduction to Compiler Construction with UNIX", Prentice-Hall Inc., 1986.

[Shneiderman83] B., "Direct Manipulation: A Step Beyond Programming Languages", Computer, p57-69, August 1983.

[Shooman83] M.L., "Software Engineering", McGraw-Hill 1983.

[Shu89] N.C., "Visual programming: Perspectives and approaches", IBM System Journal, 28(4), p525-547, 1989.

[Smith82] D.C., et all, "Designing the Star User Interface", Byte, 7(4), p242-282, Apr. 1982.

[Spivey89] J.M., "An introduction to Z and formal specifications", Software

Engineering Journal, p40-50, Jan 1989.

[Su91] Jason, Ritter P.R., "Experience in Testing the Motif Interface", IEEE Software, p26-33, March 1991.

[Sufrin82] B., "Formal specification of a display-oriented text editor", Sci. Comput. Program., Vol.1, p157-202, 1982.

[Symantec90], Symantec Corporation, THINK Pascal User Manual, Cupertino, CA, USA, 1990.

[Talbot85] D., Foreword to [CSR85], as Software Engineering Director of the Alvey Directorate.

[Took90] Roger, "Putting design into practice: formal specification and the user interface", p63-96, in [Harrison90].

[Tutorial81], Miller E. and Howden W.E. (eds), "Tutorial: Software Testing & Validation Techniques", IEEE Computer Society Press, New York, 1981.

[Weyuker80] Elaine, Ostrand T.J., "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Trans. Soft. Eng., 6(3), p236-246, May 1980, also reprinted in [Tutorial81].

[Weyuker82] Elaine,"On Testing Non-testable Programs", Computer J., 25(4), p465-470, Nov. 1982.

[Weyuker83] Elaine, "Assessing Test Data Adequacy through Program Inference", ACM Trans. Prog. Lang. and Syst., 5(4), p641-655, Oct. 1983.

[White87] L.J. "Software Testing and Verification", Advances in Computers, Vol 26, 1987, Academic Press Inc.

[Winston91] N.K., Baughman T., "Testing a Graphical User Interface, Experiences with Automation", p182-204, Proc. Pacific North West Software Quality Conference 1991.

[Wirth71a] N., "Program development by stepwise refinement", Comms. of the ACM, 14(4), p221-227, April 1971.

[Wirth71b] N., "Design of a Pascal compiler", Software Practice and Experience, Vol 1, p309-333, 1971.

[Wolverhampton90], "Testing Large Software Systems", Proc. Seminar Series on New Directions in Software Development, Wolverhampton Polytechnic, Mar 1990.

[Woodcock88] J., Loomes M., "Software Engineering Mathematics", Pitman Publishing, London, 1988.

[Yip91a] S.W.L., Robson D.J., "Graphical User Interface Validation: A problem analysis and a strategy to solution", Proc. Hawaii International Conference on System Sciences (HICSS-24), Vol.2, p91-100, Jan 1991.

[Yip91b] S.W.L., Robson D.J., "Conformance Validation of Graphical User Interfaces", Proc. Intl. Phoenix Conference on Computer and Communication (IPCCC), p733-739, Mar 1991.

[Yip91c] S.W.L., Robson D.J., "Applying Formal Specification and Functional Testing to Graphical User Interfaces", Proc. IEEE CompEuro'91 Conf., p557-561, Bologna, May 1991.

[Yip91d] S.W.L., Robson D.J., "Window User Interfaces and Software Maintenance" , Journal of Software Maintenance, 3(2), p107-123, John Wiley & Sons Ltd., June 1991.

[Young91] W.D., "Panel: Formal Methods Versus Software Engineering: Is There a Conflict?", in [TAV91], p188-189.

[Yovits87] M.C. (ed), "Advances in Computers", Vol.26, Academic Press Inc., 1987.

Appendix A Glossary

Boundary value analysis is a specification-based functional testing technique. It can be seen as a special case of equivalence partitioning. Boundary value analysis requires the selection of test data directly on, above and below the boundary of equivalence classes.

Branch coverage requires enough test cases to be written so that each direction of branch (or decision) in the program would have a true and false outcome at least once.

Call-Back Routine is a mechanism used in window systems for handling certain input events. For example, a command button (or a menu option) is declared together with the name of a call-back routine. When the command button is selected by a user, the window system will pass control over to the call-back routine to handle the event.

Code coverage is a range of criteria requiring increasing test coverage of all program statements, branches, conditions, combinations of conditions, and lastly, all program paths.

Completeness of specifications requires that all functions (or operations) on all objects of the type of interest are defined by the specification . The most obvious reason for incompleteness is that of missing functions [Howden87].

Condition coverage requires enough test cases to be written so that each condition in the program would be tested for a true and false outcome at least once.

Debugging is different from testing. Debugging is the process of locating and rectifying the textual faults in the program, design or specification, after the existence of errors has been indicated during testing.

Dialogue separation means separating out the user-interface code from the other computing components of the application program. Dialogue separation requires design decisions that affect only the user interface to be isolated from those that affect the other components of the application program [Hartson89]. Dialogue separation is crucial for easy modification and maintenance of user interfaces, and could also increase the portability of software packages.

Direct manipulation is the name of an interaction style by which users perform (or request) operations by manipulating objects that are visible on a computer screen, instead of using a command language to describe operations on objects that are invisible. (e.g. to delete a file by placing the file icon onto the trash can icon.)

Domain testing is a modified form of path coverage. It helps to select a finite set of paths for analysis. Ranges of inputs are deduced from the program structure to establish path domains. This technique reveals errors by picking test data on and slightly off the borders of path domains.

Dynamic analysis is any testing technique that requires the program to be executed.

Equivalence Partitioning is a specification-based functional testing technique. The input domain of a program is partitioned into a finite number of equivalence classes, so that one can reasonably assume a test of a representative value of each class is equivalent to a test of any other value.

An **error** is a mental mistake by a programmer or designer. It may result in textual problem with the code called a **fault**. A **failure** occurs when a program computes an incorrect output for an input in the domain of the specification [Morell87].

Error-based testing is a testing strategy which seeks to demonstrate that certain classes of errors have not been committed in the programming process [Morell87]. Error classes may be derived from a history of programmer errors, measures of software complexity, knowledge of error-prone syntactic constructs, or even error guessing [Myers79].

Functional testing is also known as *black box testing*. It is a testing strategy in which the testers are unconcerned about the internal behaviour and structure of the program under test. They perform testing on their understanding of the intended function of the program.

Graphical User Interfaces (GUI) use interactive display objects such as windows, icons, pop-up (or pull-down) menus, together with user inputs on the mouse pointer, mouse button(s) and keyboard to achieve a Human Computer Interface (HCI). This is generally called a graphical or window-based user interface to distinguish it from the traditional textual command line interface. GUIs are sometimes called WIMP interfaces.

Incremental integration is to add (or integrate) one module to the program at a time, testing is performed before the integration of the next module. Incremental integration generally results in more thorough testing and earlier detection of interface errors between modules [Myers79].

Input synthesis is an approach which simulates keyboard and mouse inputs, so as to relieve human testers from having to execute tests in generating inputs by hand.

Integration testing is a process to test the whole program by combining modules, which have been successfully tested during module / unit testing. Integration testing can be carried out in two alternative ways, incremental or non-incremental.

Journal Record and Replay (JRR) is a mechanism that records user interactions, which are later replayed for automation purposes. A survey of some commercially available JRR tools is given in section 3.7.

Module testing or **Unit testing** is the process of testing the individual subprograms, subroutines, or procedures in a program .

Multiple condition coverage requires enough test cases to be written in order that all possible combinations of conditions are tested. A multiple condition coverage would always satisfy both branch and condition coverage.

Non-incremental Integration is also called "big-bang" integration. In this approach, modules are combined all at once to form an integrated program, before testing is applied.

Path coverage is the strongest code coverage testing technique. It simply requires that all possible program paths be executed at least once.

Reachability of a specification requires that every state which satisfies the state definition can be reached by some sequence of operations applied to the initial state [Nicholl90].

Regression testing is the rerun of some existing tests after changes have been made to a program which had previously been test-accepted. This is to determine if the changes have regressed other aspects of the program.

Software testing is defined in this thesis as the process of revealing the existence of errors in computer programs, by exposing faults or differences in behaviour or code structure from what is expected. Testing is usually carried out by executing the program under test, or by examination and analysis of the program code and design.

Statement coverage requires the design of test cases to ensure that every statement in the program / module is executed at least once.

Static Analysis is any testing technique that does not involve the execution of the program under test.

Structural Testing is also known as *white box testing*. It is a testing strategy. by which the testers, are concerned with the internal structure of the program, can derive test data according to their understanding of the program's logic.

Symbolic execution is a testing technique, also known as symbolic evaluation, which does not execute a program in the traditional sense. Symbolic values of input data, instead of actual values, are fed, together with the program, into a tool that carries out symbolic execution. The outcome of symbolic execution is a set of expressions based on the symbolic values of the data.

A **test case** is a detailed design, consisting of both the required input data for program execution, and a precise description of the correct output of the program for that set of input data.

Test oracle is the name given to an external mechanism which can be used to check test output for correctness. Test oracles can take on different forms. They can consist of tables, hand calculated values, simulated results, or informal design and requirements descriptions [Howden78]. An oracle can exist in the form of a written specification or as a person who has the authority to decide if a program is working correctly [Weyuker82].

A **test plan** is the overall schedule covering all the different stages of testing, from design reviews and module testing, to final regression testing. It may enlist many test cases designed for individual modules and the program as a whole.

Test tools are software tools that assist the testing of programs in different ways, such as analysing program structure, generating test data and recording test execution.

Toolkits
A window system library can be tedious to use, as it generally provides a programming interface of low level routines. To encourage programmers to use windows, low level routines are built together to form a higher level programming interface, generally called a *toolkit.*

User Interface Management System (UIMS) can be perceived as an integrated set of tools that help user interface developers to create and manage many aspects of interfaces. [Myers89] suggests that it is preferable to call them User Interface Development Systems (UIDS) instead of UIMS. The name UIMS is used in this thesis.

Validation is the process of testing software or its specification at the end of the development effort to ensure that it meets its requirements (that it does what it is supposed to do). [IEEE83]

Verification is the process of evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase [IEEE83].

Visual verification is an attempt to validate GUI screen outputs by comparison with previously recorded bitmaps. This approach has a number of difficulties, as outlined in section 3.7.2.

Widget is a special term used in the X Window Systems to represent a fundamental data type abstraction. Logically, a widget is a rectangle with associated input / output semantics. Some widgets display information (for example, text or graphics), and others are merely containers for other widgets (for example, a menu box) [MIT89].

WIMP stands for "Window Icon Menu and Pointer", or "Window Icon Mouse and Pop-up (or Pull-down) menu". WIMPs are also called Graphical User Interfaces (GUIs). It is part of a computer program / system that uses the display objects mentioned above, to achieve interactions with users on a screen. It was first used in the Star and SmallTalk systems at Xerox.

Window systems provide the underlying window graphics libraries and device drivers for the construction of window-based or graphical user interfaces. An example is the X Windows System [Scheifler86].

Appendix B Specification of menu functions

Section 8.4 covered all the edit-display functions of ThinkEdit. In this appendix, specifications are developed for another main group of ThinkEdit functions, the menu functions. There are two menus for ThinkEdit, the "File" menu and the "Edit" menu. The visual appearance of these menus is shown in Figure 8.2.

The "Edit" menu has six menu options :
Edit = {Undo, Cut, Copy, Paste, Clear, Select All}
The "Edit" menu functions have already been dealt with in section 8.4, except the "Undo" command which is not supported in ThinkEdit.

The "File" menu has six menu options[8] :
File = {New, Open, Close, Save, Save As, Quit}

These options affect the content of the file on disk. During editing, a copy of the file (called the 'record') is changed, whilst the disk file itself is not modified until a "File" menu option is selected. The fact that new text entries or changes have been made is recorded by setting a flag called the dirtFlag. In addition to setting the dirtFlag, it is necessary to enable the "save" menu option when the active window is dirty, allowing the user to save the changes to the disk file. When a dirty window becomes clean after a file operation, the "save" menu option is then disabled, as the window's contents are then in agreement with the version on the disk.

The following sections cover menu functions for creating, saving and closing editing windows. The relationship amongst menu functions is illustrated in the state diagram in Figure B.1.

[8] Figure 8.2 actually shows 9 options for the "File" menu, three of them (i.e. Page Setup, Print and Transfer) are either unsupported or irrelevant.

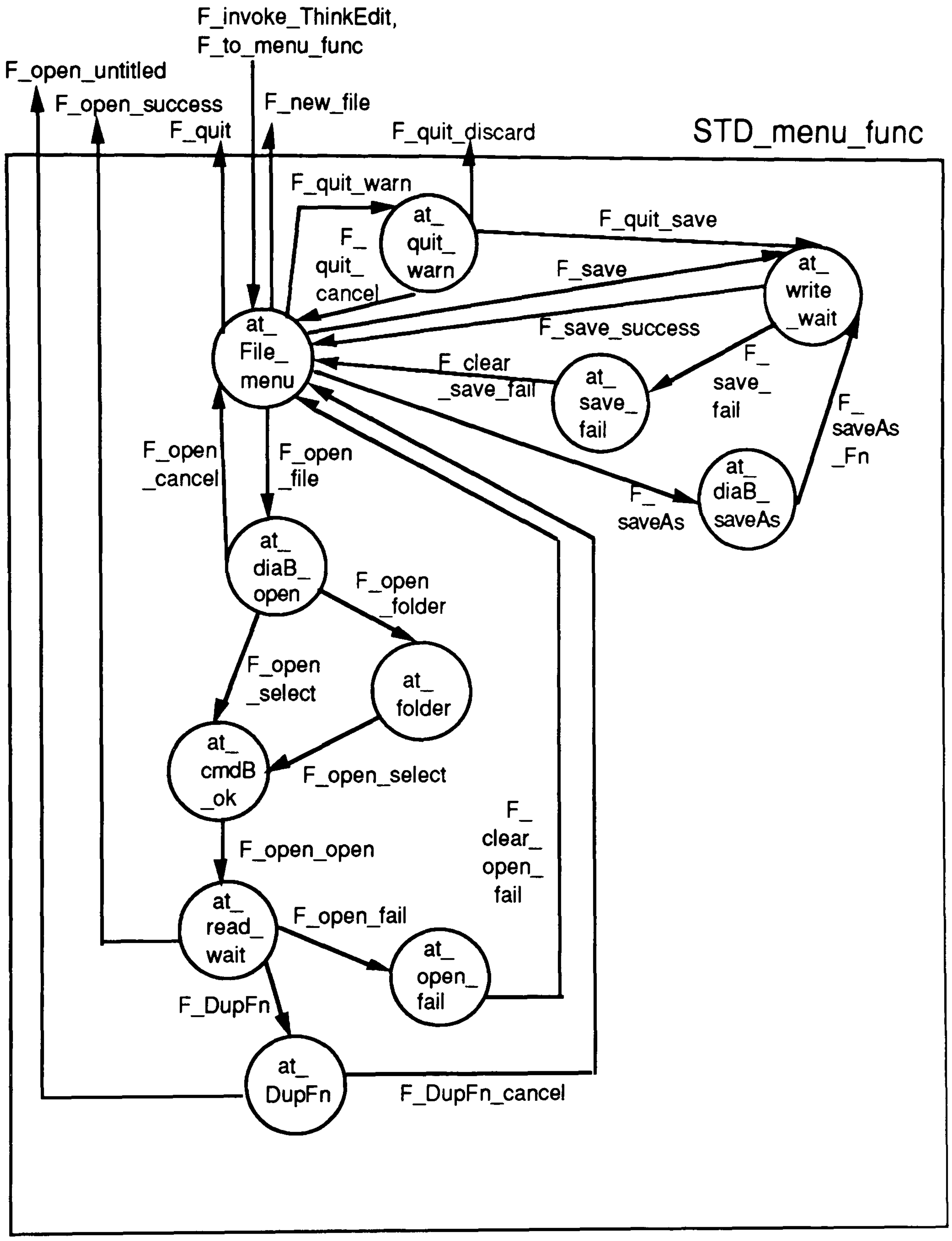


Figure B.1 STD_menu_func for FG_menu_func

B.1 Creating a new window for editing

The 'New' command is the first of the “File” menu options considered here. When the 'New' menu option is chosen, ThinkEdit will display a new editing window with the title “UntitledX”, where X is an integer between 1 and 5. ThinkEdit does not access the file system until the user chooses the 'Save' command later on in order to save the new file onto disk. In the mean time, the content of the text being edited is kept in a temporary buffer called the 'record' .

B.1.1 Function F_new_file

```
-----   Specification_for_function     F_new_file:
|
| Variables :              n, X : integer ,   0≤ X ≤ n, 0 ≤ n ≤ maxWind
|                          maxWind : integer , maxWind = 5
|                          dirtFlag  : boolean
| From_state  :            at_File_menu
| F_state_predicate :      n <  maxWind
|
| Inputs :                 kb?=<cmd-N>    or
|                          (is_inside(mp?, menu_'File')  Tand   mb?=<down>  Tand
|                           is_inside(mp?, mOpt_'New')    Tand   mb?=<up> ))
|
| To_state :               Post_Insert
| T_state_predicate :      n' = n + 1
|                          and is_visible(wind_edit#n')
|                          and X' = X + 1
|                          and text(wind_edit#n'.tBar)    =     “Untitled”//X
|                          and text(record')    = {}
|                          and wind_edit#n'.dirtFlag'=false
|                          and is_disabled(mOpt_'Save')
|                          and if   n'≥ maxWind   then   ( is_disabled(mOpt_'New')
|                              and is_disabled(mOpt_'Open') )
| Output_msg :             none
|______________________________________________________
```

<u>Explanations:</u>

- The variable "n" represents the total number of editing windows that are open under ThinkEdit. The value of "n" is incremented by 1 each time an editing window is opened. The initial value of "n" is 0, and the maximum value is 5, as controlled by maxWind.

- The variable "X" represents the total number of "UntitledX" windows that are open under ThinkEdit.

- The interaction to choose the 'New' command begins by moving the mouse pointer into the “File” menu. Then press the mouse button down, and hold it down whilst moving the mouse pointer into the option "New" within the "File" menu. Then release the mouse button.

- <cmd-N> is the command key for mOpt_'New', see explanations in 8.4.10.
- As part of the T_state_predicate of F_new_file, a new editing window is displayed.

- The construct text(wind_edit#n.tBar) gives the title of the window wind_edit#n. The title is the text on the window's title bar (denoted as tBar).

- The predicate "text(record)={ }" indicates that the editing buffer is initially empty.

- The flag dirtFlag is set to false as the file has not been updated so far. The notation dirtFlag' is used to show the value of dirtFlag after the execution of F_new_file . The notation wind_edit#n'.dirtFlag is used to represent the dirtFlag associated with the editing window wind_edit#n' , as a total of up to 5 editing windows may exist.

- The menu option "Save" is disabled as there is nothing to save.
- If the maximum number of editing windows (maxWind) is reached, both the "New" and "Open" menu options will be disabled.

B.1.2 Function F_open_file

```
-----   Specification_for_function      F_open_file :
|
| Variables :              n : integer,   0 ≤ n ≤ maxWind
|                          maxWind : integer , maxWind = 5
| From_state  :            at_File_menu
| F_state_predicate :      n <  maxWind
|
| Inputs :                  kb?=<cmd-O>    or
|                          (is_inside(mp?, menu_'File')  Tand  mb?=<down>  Tand
|                           is_inside(mp?, mOpt_'Open')   Tand  mb?=<up> ))
|
| To_state  :              at_diaB_open
| T_state_predicate :      n' = n + 1  and
|                          if  n'≥ maxWind  then  ( is_disabled(mOpt_'New')
|                               and is_disabled(mOpt_'Open') )
|                          and   is_modal(diaB_open)
|___________________________________________________________
```

Explanations:

- The interaction to open an existing file for editing is performed by selecting the 'Open' menu option within the 'File' menu.
- <cmd-O> is the command key for mOpt_'Open', see explanations in 8.4.10.

• As in the T_state_predicate of F_open_file, the value of "n" is incremented by 1.

• If the maximum number of editing windows (maxWind) is reached, both the 'New' and 'Open' menu options will be disabled.

• A modal dialogue box (diaB_open) is displayed to assist the user to choose which file is to be opened.

B.1.3 Function F_open_cancel

```
-----   Specification_for_function   F_ open_cancel :
|
| Variables :              n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :            at_diaB_open
| F_state_predicate :      T_state_predicate(open_file)
|
| Inputs :                 is_inside(mp?, cBtn_'Cancel')   Tand   mb?=<click>
|
| To_state  :              at_File_menu
| T_state_predicate :      is_not_visible(diaB_open)
|                          and   n' = n - 1  and
|                          if  n' < maxWind  then  ( is_enabled(mOpt_'New')
|                                 and is_enabled(mOpt_'Open') )
|_____________________________________________________________________
```

<u>Explanations:</u>

• As a consequence of F_open, the user is presented with a dialogue box (diaB_open) that allows a choice of files to be opened.

• If, at this stage, the user decides to abandon the opening of a file, the 'Cancel' command button can be used to clear diaB_open.

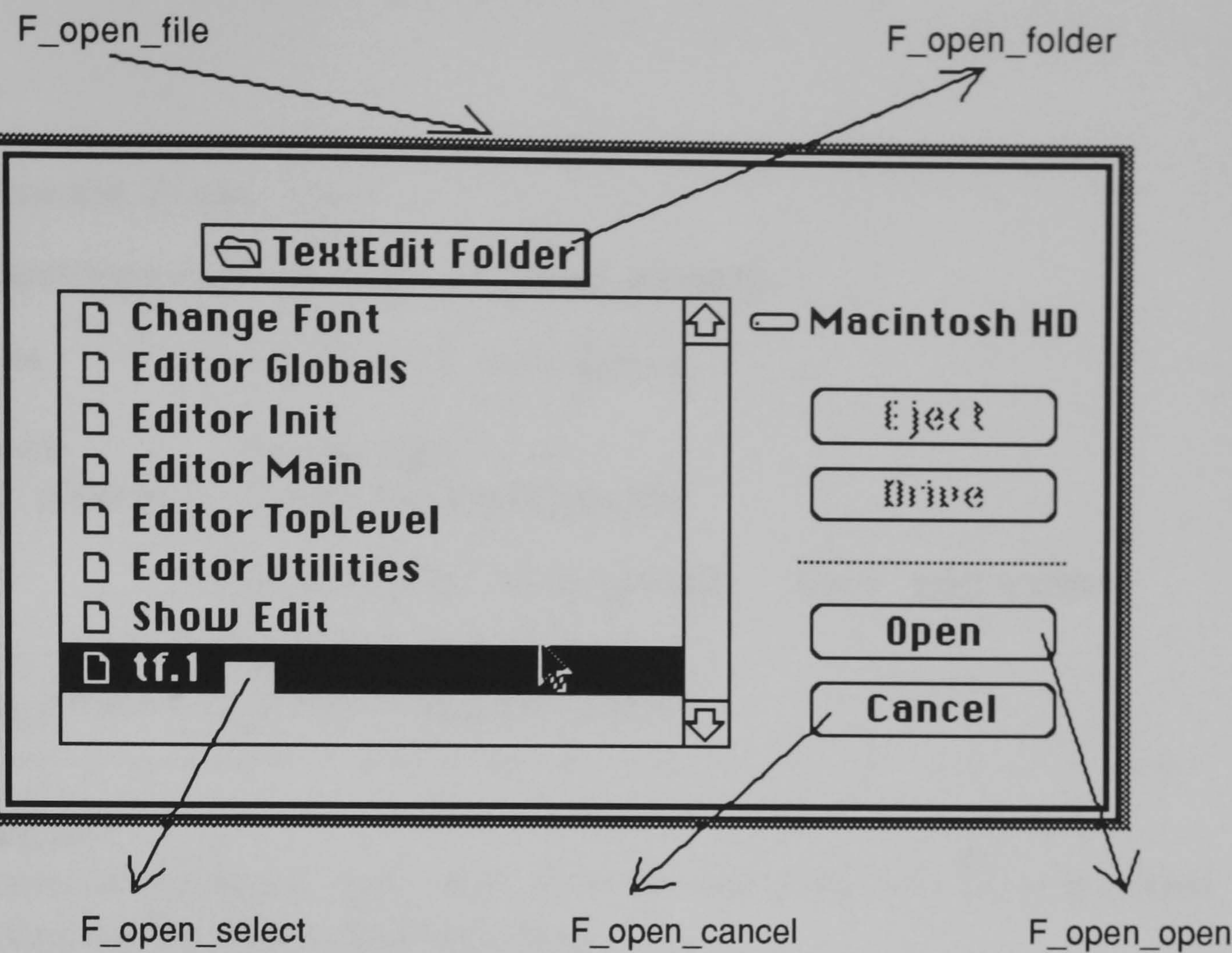


Figure B.2 A WinSTD showing the diaB_open dialogue box and associated functions

diaB_open			
	menu_folder		
		mOpt_folders (1)	
		...	
		mOpt_folders (fdrLen)	
	fileList	lines (1)	
		...	
		lines (listLen)	
	viewList	lines (offset + 1)	
		...	
		lines (offset + vListLen)	
	vBar		
		sBar_upArrow	
		sBar_pgUpRect	(Invisible if listLen ≤ vListLen)
		sBar_slideBox	(Invisible if listLen ≤ vListLen)
		sBar_pgDnRect	(Invisible if listLen ≤ vListLen)
		sBar_dnArrow	
	cBtn_'Drive'		
	cBtn_'Eject'		
	cBtn_'Open'		
	cBtn_'Cancel'		

Table B.2 A table listing the display objects within the diaB_open dialogue box

B.1.4 Function F_open_select

```
-----   Specification_for_function    F_open_select(i)  :
|
| Variables :           i : integer ,  1  ≤ i ≤ listLen
|
| From_state  :         at_diaB_open
| F_state_predicate :   T_state_predicate(open_file)
|
| Inputs :              (is_inside(mp?, viewList.line(i))    Tand   mb?=<click>
|
| To_state  :           at_cmdB_ok
| T_state_predicate :   is_hiLit   (viewList.line(i))
|_______________________________________________________________
```

Explanations:

- The interaction function F_open_select allows the user to select the file to be opened by clicking at a line within a list of file-names.
- As a consequence, the line containing the chosen file-name will be highlighted.

B.1.5 Function F_open_folder

```
-----   Specification_for_function    F_open_folder(j)  :
|
| Variables :           j : integer ,  1  ≤  j ≤  fdrLen
|
| From_state  :         at_diaB_open
| F_state_predicate :   T_state_predicate(open_file)
|
| Inputs :              is_inside(mp?, icon_folder)   Tand  mb?=<down>
|                        Tand  is_inside(mp?, mOpt_folders(j))  Tand  mb?=<up>
|
| To_state  :           at_folder
| T_state_predicate :   text(icon_folder')  =  text(mOpt_folders(j))
|                       Tand    app_msg_sent = ("folder :", text(mOpt_folders(j)) )
|                       Tand    text(fileList') = app_msg_recv
|_______________________________________________________________
```

Explanations:

- If the desired file is not in the current file folder, the interaction F_open_folder can be invoked to select the necessary file folder.

- This interaction is carried out by moving the mouse pointer into the icon resembling a file folder (denoted icon_folder) , and then pressing down the mouse button.

- A list of available file folder names will be displayed in the form of a list of menu options (denoted mOpt_folders(j)).

- While depressing the mouse button, move the mouse pointer into the menu option containing the desired file folder, and then release the mouse button.

- As a consequence, the selected item (i.e. mOpt_folders(j)) in the list will become the current folder , denoted as icon_folder' .

- An application message is then sent to request the list of file-names in the current folder.

- The list of file-names (denoted fileList) is updated upon the receipt of an application message.

B.1.6 Function F_open_open

```
-----   Specification_for_function    F_open_open :
|
| Variables :             i : integer ,  0 ≤ i ≤ listLen
|                         filename : string
|
| From_state :            at_cmdB_ok
| F_state_predicate :     T_state_predicate(open_select)
|
| Inputs :                (is_inside(mp?, viewList.lines (i))   Tand  mb?=<click>
|                         Tand  is_inside(mp?, cBtn_'Open')  Tand  mb?=<click> )
|                         or  (is_inside(mp?, viewList.lines (i))  Tand  mb?=<dClick>)
|
| To_state :              at_read_wait
| T_state_predicate :     is_not_visible(diaB_open)   and
|                         filename = text( viewList.lines (i))
|
| Output_msg :            App_msg_sent = "Read  file:" // filename
|________________________________________________________________
```

<u>Explanations:</u>

- The interaction function F_open_open allows a user to choose a certain file to be opened, amongst a list of file-names (denoted by viewList) belonging to a certain file folder.
- A file is selected by a mouse click input, whilst the mouse pointer is inside the line (within the list viewList.lines) displaying the desired file-name.

- To send the chosen filename to the the application routines responsible for opening the file, a mouse button click must be entered at the "Open" command button (cBtn_'Open'). Alternatively the user can double click at the line containing the file-name (viewList.lines (i)).

B.1.7 Function F_open_success

```
-----   Specification_for_function    F_open_success :
|
| Variables :               n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :             at_read_wait
| F_state_predicate :       T_state_predicate(F_open_open)
|
| Inputs :                  App_msg_recv = "readSuccess"
|                           and
|                           ∀ i ∈ {1, ..., n-1} • text(wind_edit#i.tBar)≠filename
|
| To_state  :               Post_Insert
| T_state_predicate :       text(wind_edit#n')= text(filename)
|                           and  text(wind_edit#n'.tBar') = filename
|                           and  is_visible(wind_edit#n')
|_____________________________________________________________________
```

Explanations:

- The function F_open_success is executed if an application message of "readSuccess" is received following F_open_open .

- A new editing window wind_edit#n is displayed. The content of the window is initially the same as that of the file chosen by F_open_select.

- If the file is not already opened in any of the existing windows, its file-name will be used as the title of the new window. Otherwise, a modal dialogue box (diaB_DupFn) is displayed, to warn the user that the file is already open in another editing window.

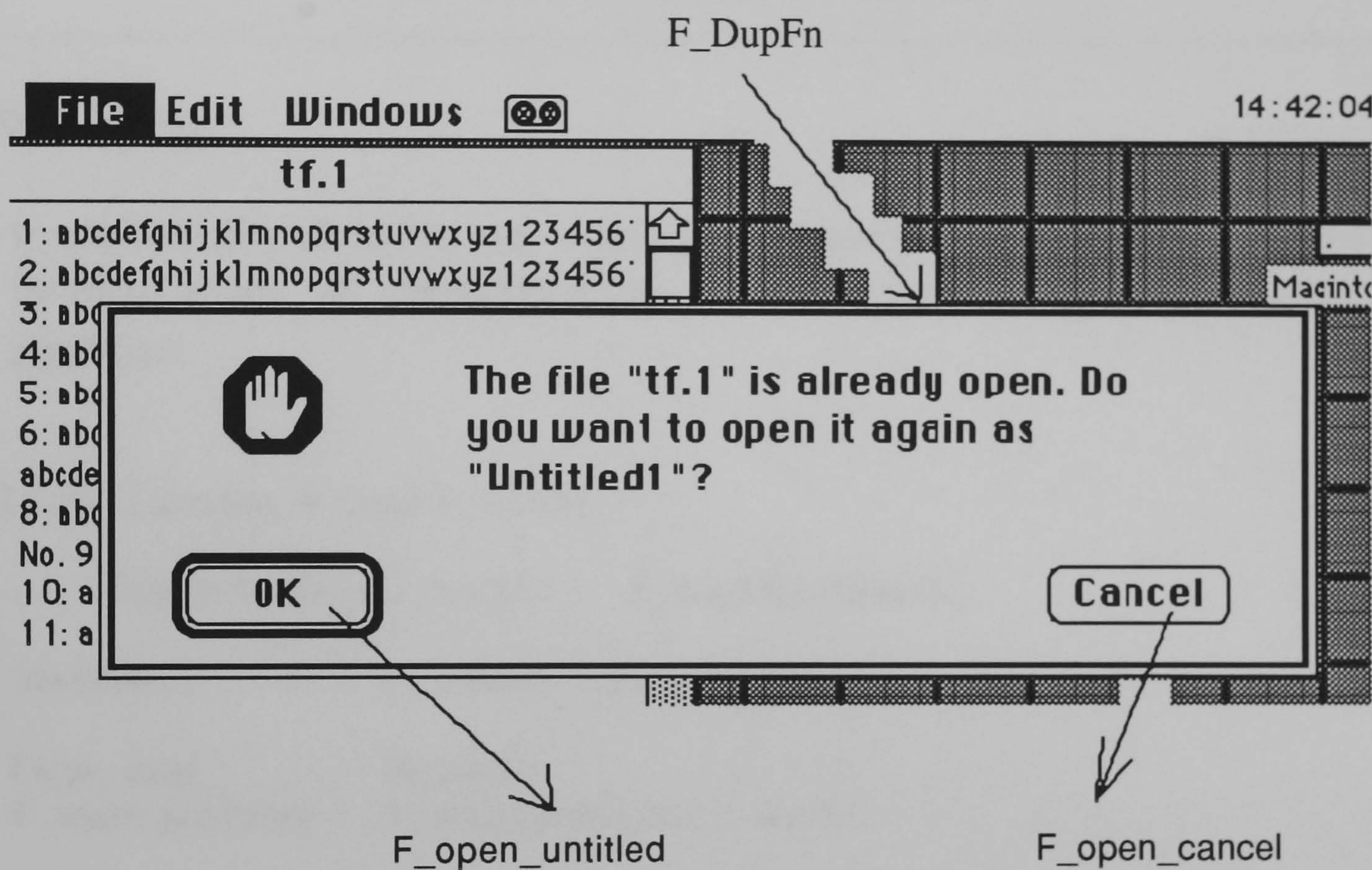
B.1.8 Function F_DupFn

```
-----   Specification_for_function    F_DupFn :
|
| Variables :               n : integer,   0 ≤ n ≤ maxWind
| From_state  :             at_read_wait
| F_state_predicate :       T_state_predicate(F_open_open)
|
| Inputs :                  App_msg_recv = "readSuccess"
|                           and
|                           ∃ i ∈ {1, ..., n-1}  • text(wind_edit#i.tBar) = filename
| To_state  :               at_DupFn
| T_state_predicate :       text(wind_edit#n')= text(filename)   and
|                           is_modal(diaB_DupFn)
|_____________________________________________________________________
```

Explanations:

- The F_state_predicate for F_dupFn are similar to those of F_open_success, except the file being opened is found to be already open.

- F_DupFn displays the dialogue diaB_DupFn to warn the user of this duplication, and suggests that the file should be opened as "UntitledX" .



diaB_DupFn

cBtn_'OK'
cBtn_'Cancel'

Figure B.3 A WinSTD showing the diaB_DupFn dialogue box and associated functions

B.1.9 Function F_open_untitled

```
-----   Specification_for_function     F_open_untitled :
|
| Variables :            n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :          at_DupFn
| F_state_predicate :    T_state_predicate(F_DupFn)
|
| Inputs :               is_inside(mp?, cBtn_'OK')   Tand   mb?=<click>
|
| To_state  :            Post_Insert
| T_state_predicate :    is_visible(wind_edit#n)   and
|                        X'= X +1
|                        filename' = "Untitled"// X'
|                        and    text(wind_edit#n.tBar') = filename'
|___________________________________________________________
```

Explanations:

• F_open_DupFn allows the user to open the duplicated file for editing as "UntitledX", by choosing the 'OK' command button. X is an integer of a value ranging from 1 to maxWind.

B.1.10 Function F_DupFn_cancel

```
-----   Specification_for_function     F_DupFn_cancel :
|
| Variables :            n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :          at_DupFn
| F_state_predicate :    T_state_predicate(F_DupFn)
|
| Inputs :               is_inside(mp?, cBtn_'Cancel')   Tand   mb?=<click>
|
| To_state  :            at_File_menu
| T_state_predicate :    is_not_visible(DiaB_DupFn)
|                        and   n' = n - 1   and
|                        if  n' < maxWind   then   ( is_enabled(mOpt_'New')
|                              and is_enabled(mOpt_'Open') )
|___________________________________________________________
```

Explanations:

• As an alternative to F_open_DupFn, the user could choose to abandon the opening of the duplicated file, by selecting the 'Cancel' command button in the diaB_dupFn dialogue box.

B.1.11 Function F_open_fail

```
-----   Specification_for_function     F_open_fail:
|
| Variables :              n : integer,   0 ≤ n ≤ maxWind
|                          filename : string
|
| From_state  :            at_read_wait
| F_state_predicate :      T_state_predicate(F_open_open)
|
| Inputs :                 App_msg_recv  =  "readError"
|
| To_state  :              at_open_fail
| T_state_predicate :      is_modal(diaB_readError)    and
|                          text(texB_errMsg)  =  "I/O error while reading from file"
|                                // filename
|___________________________________________________________________
```

Explanations:

- Following the execution of F_open_open, a file-name is sent to the application routines responsible for opening files, a reply will eventually be received from the application routines.
- If the message received (App_msg_recv) is "readError", the interaction function F_open_fail will take control.
- The user is presented with a modal dialogue box, which communicates that a read-error was encountered while opening the file.

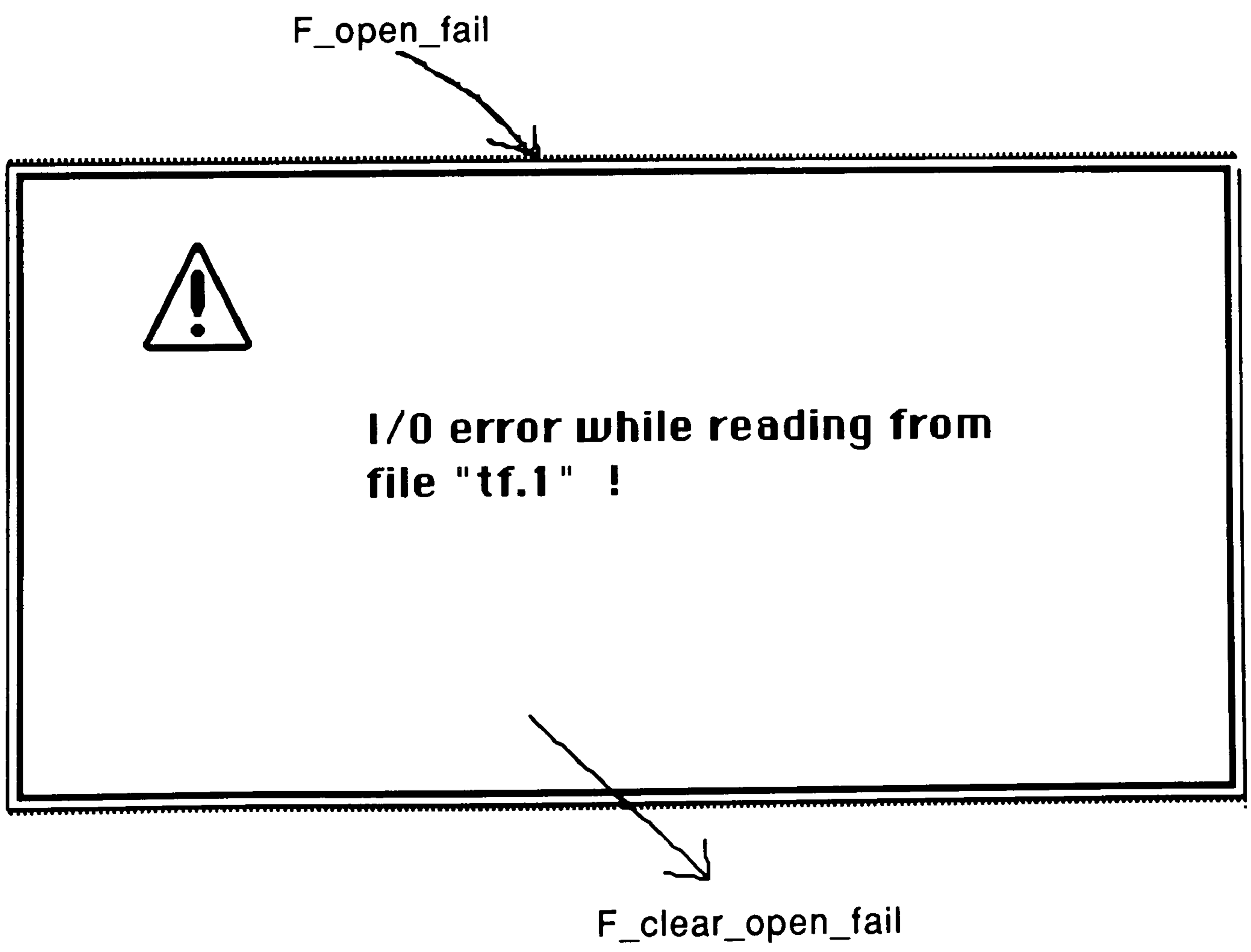


Figure B.4 A WinSTD showing the diaB_readError and associated functions

B.1.12 Function F_clear_open_fail

```
-----   Specification_for_function    F_clear_open_fail:
|
| From_state :         at_open_fail
| F_state_predicate :  T_state_predicate(F_open_fail)
|
| Inputs :             is_inside(mp?, diaB_readError)  Tand  mb?=<click>
|
| To_state :           at_File_menu
| T_state_predicate :  is_not_visible(diaB_readError)
|                      and  n' = n - 1  and
|                      if  n' < maxWind  then  ( is_enabled(mOpt_'New')
|                           and is_enabled(mOpt_'Open') )
|___________________________________________________________________
```

<u>Explanations:</u>

- In the presence of the modal dialogue box (diaB_readError), the only allowable user action is a mouse button click within the dialogue box. This will clear the dialogue box, and reverts n' to n - 1, to keep an accurate count of the number of editing windows opened.

B.2 Saving the contents of an editing onto a disk file

Having covered the 'New' and 'Open' menu options of the 'File' menu, the 'Save' option of the 'File' menu is specified in this section.

B.2.1 Function F_save_file

```
-----   Specification_for_function    F_save :
|
| Variables :              n : integer,  1 ≤ n ≤ maxWind
|                          filename : string
|
| From_state  :            at_file_menu
| F_state_predicate :      dirtFlag = true
|
| Inputs :                 is_inside(mp?, mOpt_'save')    Tand  mb?=<up>
|
| To_state  :              at_write_wait
| T_state_predicate :      filename' = text(wind_edit#n.tBar)
|
| Output_msg :             App_msg_sent = "write file:" // filename'
|__________________________________________________________________
```

Explanations:
- When the 'Save' menu option is selected, an application message is sent to the application routines, requesting that the content of the editing be saved.
- It will be saved to the disk file having the same name as the title of the editing window.

B.2.2 Function F_save_success

```
-----   Specification_for_function    F_save_success :
|
| From_state  :            at_write_wait
| F_state_predicate :      T_state_predicate(F_save  or  F_saveAs_fn)
|
| Inputs :                 App_msg_recv = "writeSuccess"
|
| To_state  :              at_file_menu
| T_state_predicate :      dirtFlag'=false
|                          and  is_disabled(mOpt_'Save')
|__________________________________________________________________
```

Explanations:
- Following the execution of F_save, an application message of "writeSuccess" may be received. This signals that the content of the editing has been saved onto a disk file.
- A couple of housekeeping tasks, clearing the dirtFlag and disabling the 'Save' menu option, are performed.

B.2.3 Function F_save_fail

```
-----   Specification_for_function     F_save_fail:
|
| From_state  :          at_write_wait
| F_state_predicate :    T_state_predicate(F_save  or  F_saveAs_fn)
|
| Inputs :               App_msg_recv  =  "writeError"
|
| To_state  :            at_save_fail
| T_state_predicate :    is_modal(diaB_writeError)    and
|                        text(texB_errMsg)  =  "I/O  error  while  writing  to  file"
|                             //  filename
|_____________________________________________________________
```

<u>Explanations:</u>

- Following the execution of F_save, the modal dialogue box diaB_writeError will be displayed if an application message of "writeError" is received.
- The dialogue box warns the user that the 'Save' command has failed, because of errors in writing to the disk file.

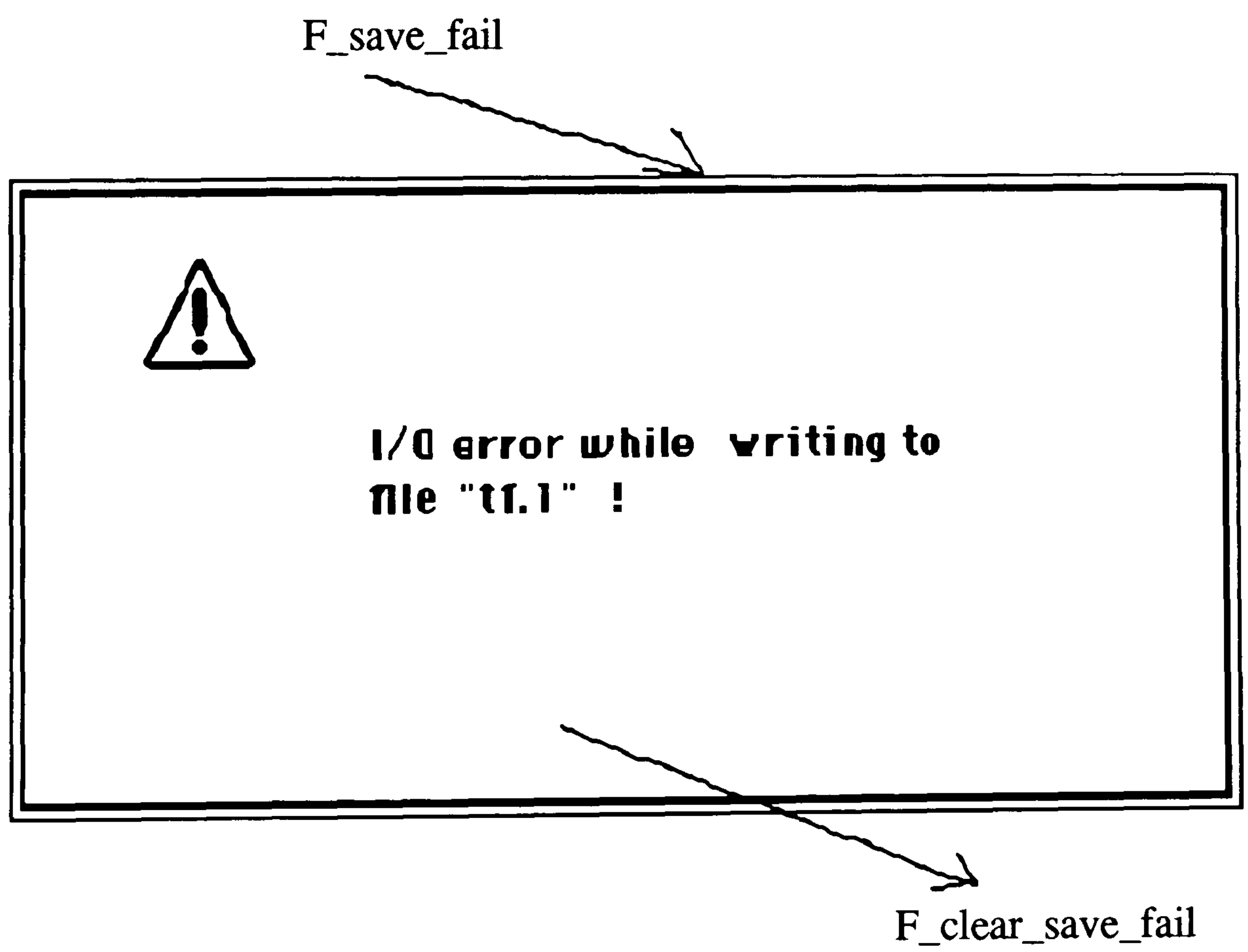


Figure B.5 A WinSTD showing the diaB_writeError and associated functions

B.2.4 Function F_clear_save_fail

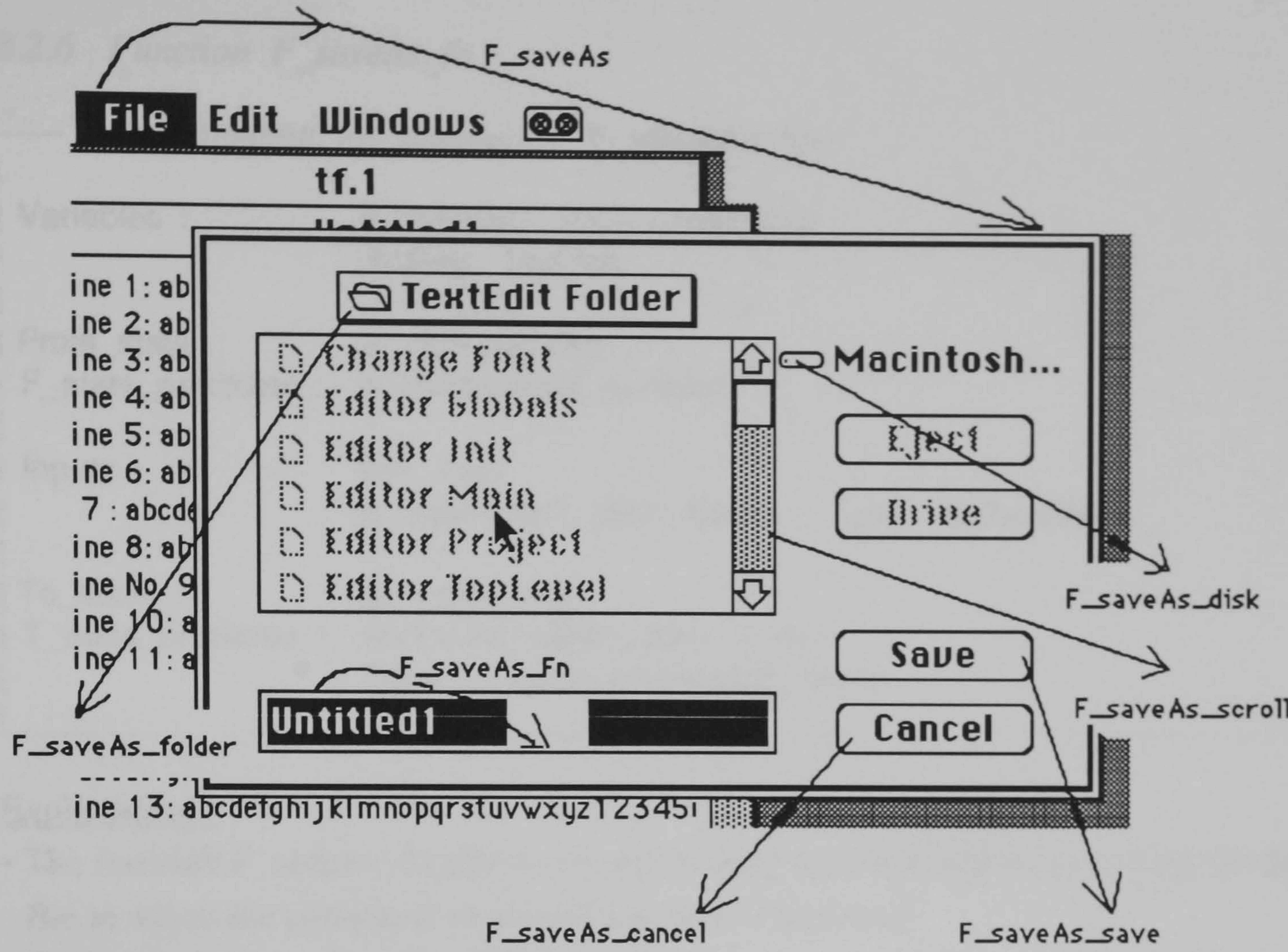
```
-----   Specification_for_function    F_clear_save_fail:
|
| From_state :          at_save_fail
| F_state_predicate :   T_state_predicate(F_open_fail)
|
| Inputs :              is_inside(mp?, diaB_writeError)   Tand   mb?=<click>
|
| To_state :            at_file_menu
| T_state_predicate :   is_not_visible(diaB_writeError)
|_____________________________________________________________
```

Explanations:

- The dialogue box diaB_writeError can be cleared by a mouse click within it.

B.2.5 Function F_saveAs

```
-----   Specification_for_function    F_saveAs:
|
| Variables :           dirtFlag : boolean
|
| From_state :          at_File_menu
| F_state_predicate :   dirtFlag = true
|
| Inputs :              is_inside(mp?, mOpt_'SaveAs')   Tand   mb?=<up>
|
| To_state :            at_diaB_saveAs
| T_state_predicate :   is_modal(diaB_saveAs)
|_____________________________________________________________
```




```
diaB_saveAs
            menu_folder
                        mOpt_folders (1)
                        ...
                        mOpt_folders (fdrLen)

            destList    lines (1)
                        ...
                        lines (listLen)

            viewList    lines (offset + 1)
                        ...
                        lines (offset + vListLen)

            vBar
                        sBar_upArrow
                        sBar_pgUpRect   (Invisible if listLen ≤ vListLen)
                        sBar_slideBox   (Invisible if listLen ≤ vListLen)
                        sBar_pgDnRect   (Invisible if listLen ≤ vListLen)
                        sBar_dnArrow
            cBtn_'Drive'
            cBtn_'Eject'
            cBtn_'Save'
            cBtn_'Cancel'
```

Figure B.6 A WinSTD showing the diaB_saveAs dialogue box and associated functions

B.2.6 Function F_saveAs_fn

```
-----   Specification_for_function    F_saveAs_fn :
|
| Variables :               n : integer,   0 ≤ n ≤ maxWind
|                           dirtFlag : boolean
|
| From_state  :             at_diaB_saveAs
| F_state_predicate :       is_modal(diaB_saveAs)
|
| Inputs :                  kb? Tand
|                           is_inside(mp?, cBtn_'Save')   Tand   mb?=<click>
|
| To_state  :               at_write_wait
| T_state_predicate :       text(wind_edit#n.tBar)  =  kb?
|                           Tand    T_state_predicate(F_save)
|__________________________________________________________________
```

Explanations :

- The function F_saveAs_fn allows the user to enter keyboard inputs, to specify the disk file in which the content of current editing should be saved.
- As a consequence, the title of the current editing window is changed to the file-name entered, and the other parts of T_state_predicate are the same as those of F_save.

B.3 Closing or quitting an editing window

The last of the 'File' menu options being considered are the “Close” and 'Quit' options, used for terminating an editing window. The operations of the 'Quit' command vary according to the value of the flag dirtFlag, which states whether or not the text has been updated since the last save.

The functions of “Close” are almost identical to those of “Quit”. There are only two differences. The first one is that “Close” can be selected by a mouse click at the close box of the window, as well as by choosing the “Close” menu option. The second difference is the wording in the dialogue box that gives a warning against quitting (see Figure B.7). The warning for “close” reads as “Do you want to save or discard ... before closing?”, instead of “... before quitting?”.

Specifications for the “Close” functions are omitted, as they are almost identical to those for “Quit”. The set of “Close” functions can be derived, by replacing the word “Quit” with “Close”, in the specifications for “Quit” functions given in the following sections.

F_Close_func = { F_close, F_close_warn, F_close_cancel, F_close_discard,
F_close_save }

B.3.1 Function F_quit

```
-----   Specification_for_function     F_quit :
|
| Variables :            n : integer,   0 ≤ n ≤ maxWind
|                        quitFlag : boolean
|
| From_state  :          at_File_menu
| F_state_predicate :    true
|
| Inputs :               kb?=<cmd-Q>   or
|                        (is_inside(mp?, menu_'File')    Tand   mb?=<down>
|                         is_inside(mp?, mOpt_'Quit')    Tand   mb?=<up> )
|
| To_state  :            Start
| T_state_predicate :    if  n ≤ 0    then    is_not_visible(icon_ThinkEdit)
|                        if  n ≥ 1   then   quitFlag'=true
|_______________________________________________________________
```

Explanations :

- The F_quit function is invoked by selecting the "Quit" option in the "File" menu.

- <cmd-Q> is the command key for mOpt_'Quit', see explanations at 8.4.10.

- If an editing window has not been opened (n≤ 0), ThinkEdit will be terminated.
- Otherwise, the quitFlag will be set to true.

```
-----   Specification_for_function     F_quit_n :
|
| Variables :            n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :          at_File_menu
| F_state_predicate :    quitFlag=true   and   n ≥ 1
|                        and  wind_edit#n.dirtFlag = false
| Inputs :               none
|
| To_state  :            at_File_menu
| T_state_predicate :    is_not_visible(wind_edit#n)
|                        Tand  n' = n -1
|_______________________________________________________________
```

Explanations :
- Window wind_edit#n , being the front window, can be closed by choosing the 'Quit' menu option within the 'File' menu.
- If the content of the editing window has not been updated (i.e. the dirty flag is false), the window is simply closed as there is no need to save to disk.

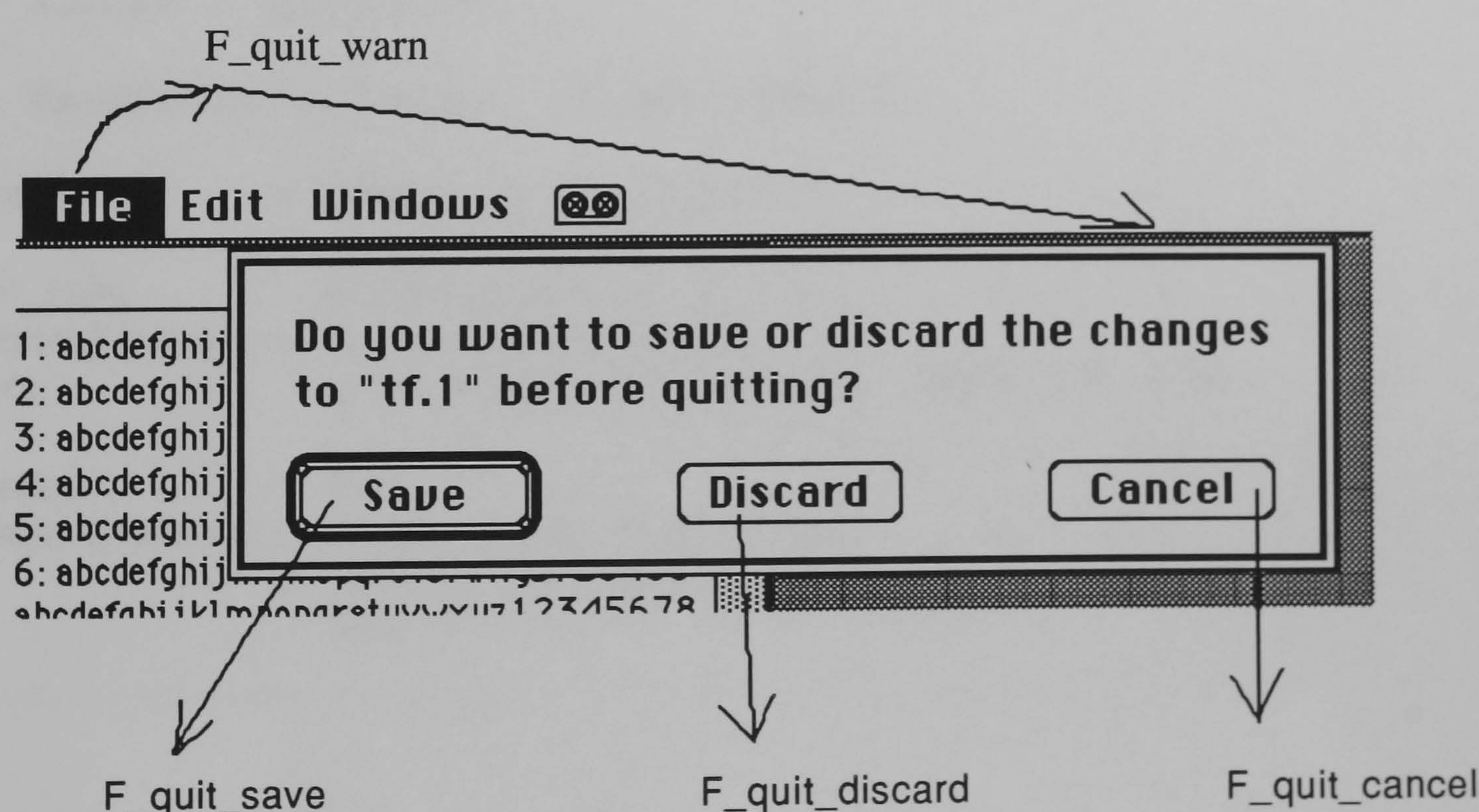
- The value of "n" is decremented by 1.
- Execution of F_quit_n will be repeated, as long as quitFlag=true and n≥1.
- There is an alternative function to F_quit_n called F_quit_warn, which displays a warning dialogue suggesting the user saves the file before quitting, if dirtFlag=true.

B.3.2 Function F_quit_warn

```
-----    Specification_for_function      F_quit_warn :
|
| Variables :               n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :             at_File_menu
| F_state_predicate :       quitFlag=true    and    n ≥ 1
|                           and   wind_edit#n.dirtFlag = true
| Inputs :                  none
|
| To_state  :               at_quit_warn
| T_state_predicate :       is_modal(diaB_quit#n)
|_______________________________________________________________
```

Explanations :

- diaB_quit denotes a dialogue box that warns the user of quitting from an unsaved editing.
- is_modal() is a predicate stating that diaB_quit is a modal dialogue that will block all other inputs until it is cleared.



diaB_quit .cBtn_'Save'
.cBtn_'Discard'
.cBtn_'Cancel'

Figure B.7 A WinSTD showing the diaB_quit dialogue box and associated functions

B.3.3 Function F_quit_cancel

```
-----   Specification_for_function      F_quit_cancel :
|
| Variables :              n : integer,   0 ≤ n ≤ maxWind
|
| Inputs :                 none
|
| From_state  :            at_quit_warn
| F_state_predicate :      is_modal(diaB_quit#n)
| Inputs :                 is_inside(mp?, cBtn_'Cancel')    Tand   mb?=<click>
|
| To_state  :              at_File_menu
| T_state_predicate :      is_not_visible(diaB_quit#n)    and    quitFlag'=false
|______________________________________________________________
```

Explanations :

- The dialogue box diaB_quit gives the user a choice of three command buttons: 'Save', 'Discard', and 'Cancel'; each would clear the dialogue box with a different action.
- The function F_quit_cancel is executed if the user chooses the 'Cancel' command button. The dialogue box diaB_quit is cleared. The editing window is not closed, nor is any action taken to save the content of the editing window to disk.

B.3.4 Function F_quit_discard

```
-----   Specification_for_function      F_quit_discard :
|
| Variables :            n : integer,   0 ≤ n ≤ maxWind
|
| From_state  :            at_quit_warn
| F_state_predicate :      is_modal(diaB_quit#n)
| Inputs :                 is_inside(mp?, cBtn_'Discard')    Tand   mb?=<click>
|
| To_state  :              Start
| T_state_predicate :      is_not_visible(diaB_quit#n)
|                          Tand  is_not_visible(wind_edit#n)
|                          Tand  n' = n -1
|______________________________________________________________
```

Explanations :

- Another way to clear the diaB_quit dialogue box is to select the 'Discard' command button.
- This will close the editing window wind_edit#n without saving its contents to a disk file, discarding any updates.

B.3.5 Function F_quit_save

```
-----  Specification_for_function     F_quit_save :
|
| Variables :             n : integer,  0 ≤ n ≤ maxWind
|
| From_state :            at_quit_warn
| F_state_predicate :     is_modal(diaB_quit#n)
| Inputs :                is_inside(mp?, cBtn_'Save')   Tand   mb?=<click>
|
| To_state :              at_write_wait
| T_state_predicate :     is_not_visible(diaB_quit#n)
|                         Tand  T_state_predicate(F_save)
|                         Tand  is_not_visible(wind_edit#n)
|                         Tand  n' = n -1
|________________________________________________________________
```

Explanations :

- The third alternative to clear the diaB_quit dialogue box is to select the 'Save' command button.
- This interaction will produce the same T_state_predicate as those of F_save, before the window is cleared.

B.3.6 Function F_to_menu_func

The functions : F_to_menu_func, F_to_sBar, F_to_wind_mgmt are dummy functions. They allow the user to change to different function groups, simply by moving the mouse pointer into the regions of the menu bar, the scroll bar or the window management region on the screen.

Appendix C Specification of scroll bar functions

The set of scroll bar interaction functions identified for ThinkEdit is:

FG_sBar = {F_sB_lineUp, F_sB_lineDn, F_sB_pageUp, F_sB_pageDn, F_sB_slider }

These functions correspond to the five control regions within a scroll bar (see section 8.3.1). These control regions (or components) are :

```
vBar                               ! A vertical scroll bar, having components :
            .sBar_upArrow          ! The "upArrow" shape at the top of a scroll bar.
            .sBar_pgUpRect         ! The area between upArrow and slider.
            .sBar_slideBox         ! The rectangular shape slider in the middle.
            .sBar_pgDnRect         ! The area between dnArrow and slider.
            .sBar_dnArrow          ! The "downArrow" shape at the bottom.
```

The physical appearance of a scroll bar and its control regions can be seen in Figure 4.2 and Figure 8.1.

C.1 Using the slider to scroll text

If the mouse button is held down when the mouse pointer is inside the slider (sBar_slideBox), the slider can be dragged up or down the scroll bar by subsequent mouse pointer movements. Scrolling of text in the editing window (viewRect) takes place in correspondence to the movement of the slider. This interaction is called function F_sB_slider in the following specification.

```
-----  Specification_for_function   F_sB_slider:
|
| Variables:            y1, y2 : integer,
|                       top_s : integer, the y-coordinate of the top of the sliderBox.
|                       slidLen : integer, length that the sliderBox can be moved.
|
|                       offset : integer,
|                       where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state  :         at_SB
| F_state_predicate :   true
|
| Inputs :              (is_inside(mp?, sBar_slideBox) and top_s=y1)
|                       Tand mb?=<down> Tand
|                       (is_inside(mp?, sBar_slideBox) and top_s=y2)
|                       Tand mb?=<up>
|
| To_state  :           at_SB
| T_state_predicate :   offset' = offset + destLen *( (y2 - y1)/slidLen)
|                       top_s'=y2
|_____________________________________________________________
```

- The length along the scroll bar which the sBar_slideBox can be moved is represented by the integer "slidLen".

- The geometry of the scroll bar can be modelled as follows.
  Given that the rect() primitive returns the coordinates of the top-left and bottom-right corners of a rectangular object,
  rect(sBar_upArrow) = ((left_u, top_u), (right_u, bottom_u))
  rect(sBar_slideBox) = ((left_s, top_s), (right_s, bottom_s))
  rect(sBar_dnArrow) = ((left_d, top_d), (right_d, bottom_d)) .

  The symbol left_u is the x coordinate of the top-left corner of sBar_upArrow.
  The symbol left_s is the x coordinate of the top-left corner of sBar_slideBox.
  The symbol left_d is the x coordinate of the top-left corner of sBar_dnArrow.
  The symbol top_u is the y coordinate of the top-left corner of sBar_upArrow.
  The symbol top_s is the y coordinate of the top-left corner of sBar_slideBox.
  The symbol top_d is the y coordinate of the top-left corner of sBar_dnArrow.
  The symbols for the x and y coordinates of the bottom-right corners of these objects can be derived in a similar fashion.

  Effectively, the length that the slider can be moved is from the bottom of sBar_upArrow to the top of sBar_dnArrow, minus the length of the slider itself;
  slidLen = top_d - bottom_u - (bottom_s - top_s) .

- To describe the position of the sBar_slideBox more effectively, a few abbreviations are useful.

F_sB_slider(at_top)	=>	top_s = bottom_u
F_sB_slider(at_middle)	=>	top_s = bottom_u + slidLen/2
F_sB_slider(at_bottom)	=>	top_s = bottom_u + slidLen or bottom_s = top_d
F_sB_slider(at_x%)	=>	top_s = bottom_u + slidLen * x%

C.2 Scrolling one line of text

If the mouse button is clicked when the mouse pointer is inside sBar_upArrow, any text in the viewRect is scrolled upwards by one line. This interaction is called function F_sB_lineUp in the following specification.

```
-----   Specification_for_function   F_sB_lineUp:
|
| Variables:             x : integer, where  1 ≤ x ≤ viewLen
|                        y : integer, where  1 ≤ y ≤ viewWidth
|
|                        offset : integer,
|                        where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state  :          at_SB
| F_state_predicate :    true
|
| Inputs :               is_inside(mp?,  sBar_upArrow)
|                        Tand  mb?=<click>
|
| To_state  :            at_SB
| T_state_predicate :    offset' = offset - 1
|                        top_s' = top_s  - slidLen / destLen
|_______________________________________________________________
```

Similarly, if the mouse button is clicked when the mouse pointer is inside sBar_dnArrow, any text in the viewRect is scrolled downwards by one line. This interaction is called function F_sB_lineDn in the following specification.

```
-----   Specification_for_function   F_sB_lineDn:
|
| Variables:             x : integer, where  1 ≤ x ≤ viewLen
|                        y : integer, where  1 ≤ y ≤ viewWidth
|
|                        offset : integer,
|                        where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state  :          at_SB
| F_state_predicate :    true
|
| Inputs :               is_inside(mp?,  sBar_dnArrow)
|                        Tand  mb?=<click>
|
| To_state  :            at_SB
| T_state_predicate :    offset' = offset + 1
|                        top_s' = top_s + slidLen / destLen
|_______________________________________________________________
```

C.3 Scrolling one page of text

If the mouse button is clicked when the mouse pointer is inside sBar_pgUpRect, any text in the viewRect is scrolled upwards by one page. This interaction is called F_sB_pageUp in the following specification.

```
----- Specification_for_function   F_sB_pageUp :
|
| Variables:             x : integer, where  1 ≤ x ≤ viewLen
|                        y : integer, where  1 ≤ y ≤ viewWidth
|
|                        offset : integer,
|                        where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state  :          at_SB
| F_state_predicate :    true
|
| Inputs :               is_inside(mp?, sBar_pgUpRect)
|                        Tand  mb?=<click>
|
| To_state  :            at_SB
| T_state_predicate :    offset' = offset - viewLen
|                        top_s' = top_s - slidLen * (viewLen / destLen)
|_____________________________________________________________
```

Similarly if the mouse button is clicked when the mouse pointer is inside sBar_pgDnRect, any text in viewRect is scrolled downwards by one page. This interaction is called F_sB_pageDn in the following specification.

```
----- Specification_for_function   F_sB_pageDn :
|
| Variables:             x : integer, where  1 ≤ x ≤ viewLen
|                        y : integer, where  1 ≤ y ≤ viewWidth
|
|                        offset : integer,
|                        where  0 ≤ offset ≤ (destLen - viewLen)
|
| From_state  :          at_SB
| F_state_predicate :    true
|
| Inputs :               is_inside(mp?, sBar_pgDnRect)
|                        Tand  mb?=<click>
|
| To_state  :            at_SB
| T_state_predicate :    offset' = offset + viewLen
|                        top_s' = top_s + slidLen * (viewLen / destLen)
|_____________________________________________________________
```

Appendix D Specification of window management functions

D.1 Selecting a window

```
---Specification_for_function  F_select_wind (wind_x):
|
| variables :           wind_x, wind_y : windows
|
| From_state :          at_WM
| F_state_predicate :   is_at_front(wind_y)
|
| Inputs :              is_inside(mp?, wind_x)  Tand  mb?=<click>
|
| To_state :            at_WM
| T_state_predicate :   is_at_front(wind_x)    and
|                       is_next_behind(wind_y , wind_x)
|____________________________________________________________
```

The object state primitive is_at_front() was introduced in section 5.3, to denote that a certain window is currently the front or active window. In an expanded specification, it is necessary to expose the visual states of a front window, in terms of lower level objects (such as the window's title bar, scroll bar and close box) that become visible as the window is brought to the front.

is_at_front(wind_x) =>
is_visible (wind_x.(tBar ∧ sBar ∧ cloB ∧ zomB ∧ sizB)) .

D.2 Moving a window

The function F_drag_wind allows a user to move a window around and within the screen, an operation commonly called dragging. It begins when the user moves the mouse pointer inside the drag region of the window. In general, the title bar is the drag region supported by most window managers. A mouse-down input is then generated by the user. The point where the mouse button was depressed (denoted as (x1,y1) in global coordinates), is remembered as the startPt. The user then moves the mouse pointer to a desired location within the screen, whilst holding the mouse button down. A dotted outline of the window, following the movements of the mouse, is displayed until the button is released at a location endPt, with coordinates (x2, y2). When the mouse button is released, the window is moved to its destination with the displacement calculated from (x2, y2) - (x1, y1). This window will become the front or active window, if it was not already so before being moved.

```
---Specification_for_function   F_drag_wind(wind_x, x1, y1, x2, y2) :
|
| Variables :           wind_x : window
|                       x1, y1, x2, y2  :  integer
|                       top, left, bottom, right : integer
|
| From_state  :         at_WM
| F_state_predicate :   rect(wind_x)=(top, left, bottom, right)
|
| Inputs :              is_inside(mp?, wind_x.tBar) and Loc(mp?)=(x1,y1)
|                       Tand mb?=<down>
|                       Tand  Loc(mp?)=(x2, y2)  Tand  mb?=<up>
|
| To_state  :           at_WM
| T_state_predicate :   rect(wind_x')=(top+y2-y1,  left+x2-x1,  bottom+y2-y1,
|                             right+x2-x1)
|                       and is_at_front(wind_x')
|___________________________________________________________________
```

Explanations:

- The notation rect(wind_x) represents the rectangular borders within which window wind_x lies.
- The borders of wind_x can be specified in terms of its top-left hand corner having (x,y) coordinates (left,top) , and its bottom-right hand corner having (x,y) coordinates (right,bottom) .
- The drag region for wind_x is anywhere inside its title bar, excluding its close box and zoom box, which also lie within the title bar.
- (x1, y1) and (x2, y2) are the coordinates of the drag action's startPt and endPt as described above.
- The rectangular border of wind_x, after the execution of F_drag_wind, is denoted rect(wind_x'), which is calculated from its previous value rect(wind_x), and the displacement between the startPt and the endPt, as shown in the T_state_predicate.

D.3 Resizing a window

The interaction needed to resize a window is similar to that of dragging a window, except the "size box" is used instead of the title bar. A user changes the size of a window by holding down the mouse button while the mouse pointer is inside the size box. The point within the size box where the mouse button was depressed is denoted as (x1, y1) in global coordinates. The user then moves the mouse pointer to a desired location within the screen, while still holding the mouse button down . A dotted outline of the bottom and right hand borders of the window is displayed, following the movements of the mouse, until the button is released at a location denoted as (x2, y2). When the mouse button is released, the dotted outline of the window's bottom and right hand borders will become the resultant borders. Since the top and left hand borders remain fixed, while the bottom and right hand borders are moved, the size of the window is effectively varied, in proportion to the displacement calculated from (x2, y2) - (x1, y1).

```
---Specification_for_function    F_resize_wind(wind_x, x1, y1, x2, y2) :
|
| Variables :              xtop, xleft, xbottom, xright : integer, initial borders of wind_x
|                          x1, y1, x2, y2  :  integer
|                          top, left, bottom, right : integer, current borders of wind_x
|                          ztop, zleft, zbottom, zright : integer, zoomed borders of wind_x
|
| From_state  :            at_WM
| F_state_predicate :      is_at_front(wind_x)    and
|                          rect(wind_x)=(top, left, bottom, right)
|
| Inputs :                 is_inside(mp?, wind _x.sizB) and Loc(mp?)=(x1, y1)
|                          Tand  mb?=<down>
|                          Tand  Loc(mp?)=(x2, y2)  Tand  mb?=<up>
|
| To_state  :              at_WM
| T_state_predicate :      rect(wind_x')=(top, left, bottom+y2-y1, right+x2-x1)  and
|                          if   rect(wind_x') ≠ (xtop, xleft, xbottom, xright)
|                               then  (ztop, zleft, zbottom, zright)  =  rect(wind_x')
|                               else  (ztop, zleft, zbottom, zright)  =  rect(wind_x)
|_____________________________________________________________________________
```

<u>Explanations:</u>

- The notation rect(wind_x) = (top, left, bottom, right) again represents the current rectangular border of wind_x.
- (xtop, xleft, xbottom, xright) denotes the initial border of wind_x when it was created.
- (ztop, zleft, zbottom, zright) represents an alternative (also called zoomed) border of wind_x.
- The size box of wind_x is denoted as wind_x.sizB
- The resultant rectangular border of wind_x is denoted as rect(wind_x'), with its new bottom and right hand border calculated as shown in the T_state_predicate.
- If rect(wind_x') is not the same as the initial borders, it is stored in (ztop, zleft, zbottom, zright), as the zoomed border.
- If rect(wind_x') is the same as the initial border before the execution of F_resize_wind, the old zoomed borders are retained.

D.4 Zooming a window

A window has two alternative sets of borders, representing two rectangles, one larger than the other. The user can choose between the alternative sizes of a window by generating a mouse click within the "zoom box".

```
---Specification_for_function   F_zoom_wind(wind_x) :
|
| Variables :           xtop, xleft, xbottom, xright : integer, initial borders of wind_x
|                       wind_x : window
|                       top, left, bottom, right : integer, current borders of wind_x
|                       ztop, zleft, zbottom, zright : integer, zoomed borders of wind_x
|
| From_state :          at_WM
| F_state_predicate :   is_at_front(wind_x)   and
|
| Inputs :              is_inside(mp?, wind_x.zomB)  Tand  mb?=<click>
|
| To_state :            at_WM
| T_state_predicate :   if  rect(wind_x)=(xtop, xleft, xbottom, xright)
|                             then  rect(wind_x')=(ztop, zleft, zbottom, zright)
|                             else   rect(wind_x')=(xtop, xleft, xbottom, xright)
|_____________________________________________________________
```

Explanations:

- The notation rect(wind_x) = (top, left, bottom, right) again represents the current borders of wind_x.
- Upon a mouse button click at the zoom box, the borders of wind_x will be toggled between its initial set of borders (xtop, xleft, xbottom, xright), and its zoomed borders (ztop, zleft, zbottom, zright) .

```
---Specification_for_function   F_zomB_track(wind_x) :
|
| From_state :          at_WM
| F_state_predicate :   rect(wind_x)=(top, left, bottom, right)
|
| Inputs :              is_inside(mp?, wind_x.zomB)  Tand  mb?=<down>
|                       Tand is_not_inside (mp?, wind_x.zomB)  Tand  mb?=<up>
| To_state :            at_WM
| T_state_predicate :   rect(wind_x')=rect(wind_x)
|_____________________________________________________________
```

Explanations:

- The notation rect(wind_x) = (top, left, bottom, right) again represents the current borders of wind_x.
- If the mouse pointer is moved outside the close box before the mouse button is released, the size of the window remains unchanged.

D.5 Closing a window

```
---Specification_for_function    F_close_wind(wind_x) :
|
| From_state :           at_WM
| F_state_predicate :    is_at_font (wind_x)
|
| Inputs :               is_inside(mp?, wind_x.cloB)  Tand  mb?=<click>
|
| To_state :             at_WM
| T_state_predicate :    is_not_visible (wind_x')
|___________________________________________________________
```

Explanations:

- The notation is_at_front(wind_x) states that wind_x is currently the front or active window on the screen, which implies wind_x is visible.
- A mouse button click when the mouse pointer is inside the close box of wind_x will cause wind_x to be closed (i.e. disappear from the screen).

```
---Specification_for_function    F_cloB_track(wind_x) :
|
| From_state :           at_WM
| F_state_predicate :    is_at_font (wind_x)
|
| Inputs :               is_inside(mp?, wind_x.cloB)  Tand  mb?=<down>
|                        Tand  is_not_inside (mp?, wind_x.cloB)  Tand  mb?=<up>
| To_state :             at_WM
| T_state_predicate :    is_at_font (wind_x')
|___________________________________________________________
```

Explanations:

- The F_state_predicate for F_cloB_track is the same as that of F_close_wind, except that the mouse pointer is moved outside the close box before the mouse button is released.
- Effectively the movement of the mouse pointer is tracked (or followed) between the mouse-down and the subsequent mouse-up input. As the mouse button is released outside the close box, wind_x is not closed and remains as the front window.
- F_cloB_track appears to have achieved nothing, functionally speaking. However the availability of F_cloB_track may be considered important in human-factors designs. It allows the user a "safety exit" between a mouse-down and a mouse-up. The mistake of closing the wrong window frustrates a user and upsets the planned sequence of interaction, inevitably costing time and effort to rectify .
- Function F_cloB_track provides an escape path from F_close_wind, in order to get back to the previous state without closing the window. This can be seen in a WinSTD as an arc that starts from one state and finishes by returning to the same state. A WinSTD is useful in exposing possible “safety exits” in a user interface design, such as F_cloB_track.

Appendix E Bibliography

[Alexander87] Heather, "Executable Specification as an aid to dialogue design", in [INTERACT'87], p739-744.

[Alty87] J.L., Mullin J., "The Role of the Dialogue System in a User Interface Management System", in [INTERACT'87], p1007-1012.

[Apple87] Computers Inc., "Human Interface Guidelines: The Apple Desktop Interface", Addison-Wesley 1987.

[Balcer89] M.J., Hasling W.M., Ostrand T.J., "Automatic Generation of Test Scripts from Formal Test Specifications", in [TAV89].

[Bauer79] J.A., Finger A.B., "Test plan generation using formal grammars". Proc. 4th Intl. Conf. on Soft. Eng., p425-432, Munich 1979.

[Bird83] D.L., Munoz C.U., "Automatic generation of random self-checking test cases", IBM Systems Journal, 22(3), p229-245, 1983.

[Bloomfield86] R.E., Froome P.K.D., "The Application of Formal Methods to The Assessment of High Integrity Software", IEEE Trans. Soft. Eng., 12(1), p988-993, Sept 1986.

[Boehm76] B.W., "Software Engineering", IEEE Trans. on Computers, 25(12), p1226-1241, Dec. 1976.

[Boniwell88] S., "Portable User Interfaces", Computer Graphics' 88, p125-133, published by On-Line Publications (Middx, U.K.).

[Brown89] J.M., Gilg T.J., "Sharing testing responsibilities in the Starbase/X11", Hewlett-Packard Journal, 40(6), p42-46, Dec 1989.

[Buckley79] F., "A standard for software quality assurance plan", Computer, 12(8), P43-50, Aug 1979.

[Buxton83] W., et al, "Towards a comprehensive user interface management system", ACM Computer Graphics, 17(3), p35-42, July 1983.

[Chang87] S.K., "Visual Languages : A Tutorial and Survey", IEEE Software, p29-39, Jan 1987.

[Chang89] S.K., et al, "A Visual Language Compiler", IEEE Trans. Soft. Eng., 15(5), p506-525, May 1989.

[CHI'8x], Proceedings of ACM CHI'8x Conf. on Computer-Human Interaction, e.g. CHI'88, CHI'89, published by Addison-Wesley.

[Cohen86] B., Harwood W.T., Jackson M.I., "The Specification of Complex Systems", Addison-Wesley Publishing Company, 1986.

[Davison88] A., et al, "Current technology in distributed window systems", Computer Graphics'88, p75-83, On-Line Publications (Middx, U.K.).

[DeMillo87] R.A., McCracken W.M., Martin R.J., Passafiume J.F., "Software Testing and Evaluation", Benjamin/Cummings Publishing Company, Inc. 1987.

[Deutsch82] M.S., "Software Verification and Validation, Realistic Project Approaches", Prentice-Hall Inc., 1982.

[Dijkstra75] E.W., "Guarded Commands, non-determinacy & formal derivation of programs", Comms. of the ACM, 18(8), p453-457, 1975.

[Dix87a] A.J., Harrison M.D., "Formalising models of interaction in the design of a display editor", in [INTERACT'87], p409-414.

[Dix87b] Alan John, "Formal Methods and Interactive Systems: Principle and Practice", PhD thesis, University of York, 1987.

[Duce88a] D.A., "Integration through standards", Computer Graphics'88, p135-144, published by On-Line Publications (Middx, U.K.).

[Duce88b] D.A., et al, "Formal specification of a small example based on GKS", ACM Trans. Graphics, 7(3), p180-197, 1988.

[Dudley87] Tim, "Report Generation Using a Visual Programming Interface", in INTERACT'87, p521-528.

[Dunham89] J.R., "V & V in the next decade", IEEE Software, p47-53, May 1989.

[Durham88] University of Durham, Faculty of Science, "Administrative Notes for the Guidance of Supervisors and Postgraduates", Sept 1988.

[Dyer87] M., "A Formal Approach to Software Error Removal", The Journal of Systems & Software 7, p109-114, 1987.

[Edmonds84] E.A., Guest S., "The SYNICS2 Interface Manger", Proc. INTERACT'84, p52-56.

[Fairley85] R.E., "Software Engineering Concepts", McGraw-Hill 1985.

[Foley87] James D., Kim W.C., Gibbs C.A., "Algorithms to transform the formal specification of a user-computer interfaces", in [INTERACT'87], p1001-1006.

[Gimnich87] R., Ebert J., "Constructive Formal Specifications for Rapid Prototyping", in [INTERACT'87], p1047-1052.

[Goodfellow86] M.J., "WHIM, the Window Handler and Input Manager", IEEE Computer Graphics & Applications, 6(5), p46-52, May 1986.

[Green83] M., "Report on Dialogue Specification Tools", Proc. Seeheim Workshop, p9-20, Nov 1983.

[Guest82] S.P., "The Use of Software Tools for Dialogue Design", Intl. Journal of Man-Machine Studies, 16, p263-285.

[Guttag78] J.V., Horning J.J., "The algebraic specification of abstract data types", Acta Informatica, 10, p27-52, 1978.

[Hamlet88] R., Special section on software testing, Comms. of the ACM 31(6), June 1988.

[Harbert90] A., et al, "A Graphical Specification System for User Interface Design". IEEE Software, p12-20, July 1990.

[Hayes86] I.J., "Specification Directed Module Testing", IEEE Trans. Soft. Eng., 12(1), p124-133, Jan. 1986.

[HCI'8x], Proceedings of HCI'8x Conf, e.g. "People and Computer IV" for HCI'88, published by Cambridge University Press.

[Henderson86] P., "Functional Programming, Formal Specification and Rapid Prototyping", IEEE Trans. Soft.Eng., 12(2), p241-250, Feb. 1986.

[Hennell90] M.A., "A comparison of static and dynamic conformance analyses", in [Wolverhampton90].

[Hetzel88] Bill, Gelperin D., "The Growth of Software Testing", Comms. of the ACM, 31(6), p687-695, June 1988.

[Hoeber88] T., "Face to Face with Open Look", BYTE, p286-296, Dec 1988.

[Holcombe87a] M., "Goal Directed Task Analysis and Formal Interface Specifications", Intl. CIS Journal, 1(4), p14-22, 1987.

[Holcombe87b] M., "Formal methods in the specification of the human-machine interface", Intl. CIS Journal, 1(2), p24-34, 1987.

[Howden80] W.E., "Functional Testing and Design Abstractions", The Journal of Systems and Software, 1, p307-313, 1980.

[Hurley89] W.D., Silbert J.L., "Modelling User Interface-Application Interactions", IEEE Software, p71-77, Jan 1989.

[Ince87] D.C., "The Automatic Generation of Test Data", Computer Journal, 30(1), p63-69, 1987.

[INTERACT'8x], Proceeding of Human-Computer Interaction - INTERACT'8x Conf., e.g. INTERACT'87, Bullinger H.J., Shackel B. (eds), published by North Holland.

[Jeffries91] Robin, "User Interface Evaluation in the Real World: A Comparison of Four Approaches", p205-220, Proc. Pacific North West Software Quality Conference 1991.

[Johnson92] P., "Human Computer Interaction, Psychology, Task Analysis and Software Engineering", McGraw-Hill Book Company, 1992.

[Jones89] Oliver, "Introduction to the X Window Systems", Prentice-Hall 1989.

[Kantorowitz89] E., Sudarsky O.(Technion-Isreal Institute of Technology). "The Adaptable User Interface", Comms. of ACM, 32(11), p1352-1358, Nov. 1989.

[Karam91] G.M., Buhr R.J.A., "Temporal Logic-Based Deadlock Analysis For Ada". IEEE Trans. Software Engineering, 17(10), Oct. 1991.

[Kemmerer85] R.A., "Testing Formal Specifications to Detect Design Errors". IEEE Trans. Soft. Eng., 11(1) p32-43, Jan 1985.

[Kernighan88] B.W., Ritchie D.M., "The C Programming Language", 2nd Ed., Prentice-Hall 1988.

[Konsynski85] B.R., et al, "A View on windows: Current approaches and neglected opportunities", Proc. AFIPS Conf. vol 54, NCC 1985.

[Kooij89] M., "Interface specification with temporal logic", Proc. 5th Intl. Workshop on Software Specification and Design, May 1989.

[Kopetz79] H., 'Software Reliability', Technical Univ. of Berlin, The MacMillan Press Ltd., 1979.

[Kuo88] F.Y., Karimi J., "User interface design from a real time perspective", Comms. of the ACM, 31(12), p1456-1466, Dec 1988.

[Laski88] J.W., "Testing in Top-down Program Development", in [TAV88], p72-79.

[Laski89] J., "Testing in the program development cycle", Soft. Eng. Journal, Mar 1989.

[Leveson90] N.G., et al, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study", IEEE Trans. Soft. Eng., 16(4), p432-443, April 1990.

[Lindquist88] T.E., Jenkins J.R., "Test-Case Generation with IOGen", IEEE Software, p72-79, Jan 1988.

[Lutz90] Mike, "Testing Tools", IEEE Software, p53-57, May 1990.

[Marick91] B., "The Weak Mutation Hypothesis", in [TAV91], p190-199, Oct 1991.

[McMullin82] P.R., "DAISTS: A system for using specifications to test implementations", Ph.D. dissertation, Dep. Comput. Sci., Univ. of Maryland, 1982.

[Miriyala91] K., Harandi M.T., "Automatic Derivation of Formal Software Specifications from Informal Descriptions" , IEEE Trans. Soft.Eng., 17(10), Oct., 1991.

[Morris88] D., et al, "Human-Computer Interface Recording", The Computer Journal, 31(5), p437-444, 1988.

[Newton90] Jenny, Hanlon M., "Data Logic FOREST Case Study : A MAL Specification of the MVC paradigm as applied to SmallTalk-80", Internal report (Data Logic), Sept. 1990.

[Niguidula87] D.A., vanDam A., "Pascal on the Macintosh, A Graphical Approach", Addison-Wesley 1987.

[Olsen84] Dan R. Jr, "Push Down Automata for User Interface Management", ACM Trans on Graphics 3(3), p177-203, July 1984.

[Olsen85] D.R.Jr, "Input / output linkage in a user interface management system", Computer Graphics, p191-197, July 1985.

[Olsen86] D.R.Jr, "MIKE: The Menu Interaction Kontrol Environment", ACM Trans on Graphics, 5(4), p318-344, Oct 1986.

[Olsen86a] D.R.Jr, "Editing Templates: A User Interface Generation Tool", IEEE CG&A, p40-45, Nov. 1986.

[Olsen87] D.R.Jr, "Larger Issues in User Interface Management", Computer Graphics, p134-137, April 1987.

[Ostrand88] T.J., Balcer M.J., "The category-partition method for specifying and generating functional tests", Comms. of the ACM, 31(6), p676-685, June 1988.

[Ould86] M.A., Unwin C. (eds), "Testing in Software Development", British Computer Society Monographs in Informatics, Cambridge University Press 1986.

[Ould91] Martyn, "Testing - a challenge to method and tool developers", Software Engineering Journal, p59-64, March 1991.

[Pagan81] F.G., "Formal Specification of Programming Languages", Prentice-Hall, 1981.

[Peeling89] N.E., Youll D.P., "Past and future trends for portable tools interfaces", Information and Software Technology, 31(4), p175-180, May 1989.

[Perlman88] Gray, "User Interface Development", SEI Curriculum Module SEI-CM-17-1.0, April 1988.

[Philips87] E.M., Pugh D.S., "How to get a PhD", Open University Press 1987.

[Ramamoorthy76] C.V., Ho S.F., Chen W.T., "On the Automated Generation of Program Test Data", IEEE Trans. Soft. Eng., 2(4), p293-300, Dec 1976.

[Roberts88] W.T., et al, "NeWS and X, Beauty and the Beast?", Dept. of Computer Science, Queen Mary College, 1988.

[Roper88] R.M.F., Smith P., "A specification-based functional testing method for JSP designed programs", Information and Software Technology, p89-98, March 1988, Butterworth & Co (Publishers) Ltd.

[SDT85] Proceedings - A Second Conference on "Software Development Tools, Techniques, and Alternatives", December 2-5, 1985, San Francisco, California.

[Stott88] J.W., Kottemann J.E., "Anatomy of a Compact User Interface Development Tool", Comms. of the ACM, 31(1), p56-67, Jan 1988.

[Sum86] R.N., "An Approach to Operating System Testing", The Journal of System and Software 6, p273-284, Elsevier Science Publishing Co.

[SummerSchool81], Chandrasekaram B., Radicchi S. (eds), "Computer Program Testing", North-Holland 1981.

[Szekely87] P., "Modular Implementation of Presentations", Proc. SIGCHI+GI87. ACM New York, p235-240, 1987.

[TAV86] Clarke L.A.(ed), Proceedings - ACM SIGSOFT Workshop on Software Testing. July 1986, Banff, Canada.

[TAV88] White L.(ed), Proceedings of the ACM SIGSOFT'88 Symposium on Software Testing, Analysis, and Verification (TAV2), 1988, Banff, Canada.

[TAV89], Kemmerer R.A.(ed), Proceedings of the ACM SIGSOFT'89 Symposium on Software Testing, Analysis, and Verification (TAV3), in Software Engineering Notes Vol.14, No.8, December 1989.

[TAV91], Leveson N. G. (Program Chair), Proceedings of the ACM SIGSOFT'91 Symposium on Software Testing, Analysis, and Verification (TAV4), Victoria, Oct 1991.

[Tsai90] W.T., et al, "Automated Test Case Generation for Programs specified by Relational Algebra Queries", IEEE Trans. Soft. Eng., 16(3), p316-324, Mar 1990.

[Velasco87] F.R.D., "A Method for Test Data Selection", The Journal of Systems & Software 7, p89-97, 1987.

[Vince86] John, "The Computer Graphics Jig-Saw Puzzle", Computer Graphics'86, published by On-Line Publications, London.

[Visual86], Ichikawa T., Korfhage R.R. (eds), Proc. IEEE Computer Society Workshop on Visual Languages, Dallas, June 1986.

[Wall89] C.T., "An evaluation of three user interface management systems", UMIST 1989.

[Wallace89] D.R., Fujii R.U., "Software Verification and Validation: an overview" IEEE Software, p10-17, May 1989.

[Wasserman86a] A.I., et al, "Building Reliable Interactive Information Systems", IEEE Trans. Soft. Eng., 12(1), p147-156, Jan 1986.

[Wasserman86b] A.I., et al, "Developing Interactive Information Systems with the User Software Eng. Methodology", IEEE Trans. Soft. Eng., 12(2), p326-345, 1986.

[Weyuker86] Elaine, "Axiomatizing software test data adequacy", IEEE Trans. Soft. Eng., 12(12), p1128-1138, Dec 1986.

[Weyuker88] Elaine, "The Evaluation of Program-Based Software Test Data Adequacy Criteria", Comms. of the ACM, 31(6), p668-675, June 1988.

[Williams86] A., "An Architecture for User Interface R&D", IEEE CG&A, 6(7), p39-50, July 1986.

[Wood89] C.A., Gray P.D., "User Interface - Application Communication in the Chimera UIMS", Druid Project Research Report R-89-2, Dept. of Computing Science, University of Glasgow 1989.

[Zeil88] S.J., "Complexity of the EQUATE Testing Strategy", The Journal of Systems & Software, 8(2), p91-104, Mar 1988.