# Graphical User Interfaces Validation: A problem analysis and a strategy to solution

Stephen W.L. Yip and David J. Robson, September 1990

Computer Science Dept., University of Durham, South Road, Durham DH1 3LE,
England, UK. Telephone +44 91 374 3651, Email S.W.L.Yip@UK.AC.DURHAM

**Abstract**

This paper begins by justifying the importance of graphical user interfaces (GUIs) and the need for proper validation. The various problems in GUI validation are classified into 3 categories : *functional*, *structural* and *environmental* issues. The functional aspects of GUI are examined from the mapping of display objects on screen, interaction functions, to basic interaction components and window management functions. The largest functional issue identified is the lack of a formal specification suitable for deriving test cases. The main structural problem is in deciding on which of the software levels (i.e. window systems, toolkits, UIMS and applications) to target tests. The environmental issues concern human testers, automation, input synthesis and output visual verification.

At the heart of all software testing activities, whether GUI or conventional, lies the problem of test case selection as testing budgets are finite. This paper concludes with a strategy for validation, based on derivation of test cases from a formal specification.

## 1 The advent of GUI software

Human Computer Interface (HCI) is an important subject. Command Languages have been the major means of HCI until the recent arrival of window or graphical user interfaces. With the advent of Graphical User Interfaces (GUI), a new style of user interaction called *Direct Manipulation* has emerged [38]. 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" that requires no embedded formatting commands), user interactions 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 file 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 recognised that user-interface software is often large, complex, difficult to create, test and maintain [33]. Surveys of artificial intelligence applications, for example, reported that 40% to 50% of the code and run time memory are devoted to user-interface aspects [4] . [14] reveals reports of 50% to 80% of interactive systems are devoted to user interfaces.

Over the last decade, research and development efforts towards better or more formalized design of user-interface software has 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, [36]), a number of models and specification methodologies have been published. The term UIMS (User Interface Management System) was first coined at the Seattle workshop. 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.

The user-interface issue is further promoted, with the emergence of the X Window System in 1987 as the *de facto* standard window system, upon which applications can build their graphical user interfaces. This promotes the portability of X Window System based applications [2], [32]. However 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*.

## 2 Problems confronting GUI validation

In contrast, very little effort has been directed towards more systematic and automated validation of user-interfaces. Prototyping is the only accepted requirements testing practice, in both the industrial and academic worlds. The aim of prototyping is to get users to try out prototypes and then introduce modifications according to their comments [33], [10]. Prototyping as a means of testing the specification of user requirements is useful for obtaining feedback from the users about the overall usability and acceptability of the user-interface. However it is by no means a thorough testing procedure. In many cases the final implementation is likely to be quite different from the prototype. Proper testing is needed to uncover bugs and to establish an acceptable level of confidence in the user-interface.

Until now the testing of GUIs is usually undertaken by human testers who exercise the system to check its functionality. Often these tests are managed in an *ad hoc* manner. When an error is discovered, it may well depend on 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. It is important that the problems of GUI testing be investigated, with the goal of finding ways towards systematic, thorough and automated testing.



0073-1129/91/0000/0091$01.00 © 1991 IEEE

The problems in validating GUI software are similar to that of highly interactive programs, being difficult to automate. The need to test interactions has outdated the old practice of running a long script in batch mode to exercise programs thoroughly. There are the usual testing need of test case selection and test oracles (see Appendix A). Some fundamental questions are also useful to rouse a wider understanding of the problem areas.

Q1- How is a GUI different from other software and does it deserve 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 to help to reason about GUI software and its validation ?

Some answers to the above questions will emerge in our discussion. In the following sections, the various problem areas within the domain of GUI validation are exposed and analysed in 3 categories : *functional*, *structural* and *environmental* issues.

## 3 Functional aspects of GUIs and testing

The general functions of user interfaces is illustrated in the Seeheim Model ([36], probably the most well known model for user interfaces) as shown in the following diagram :

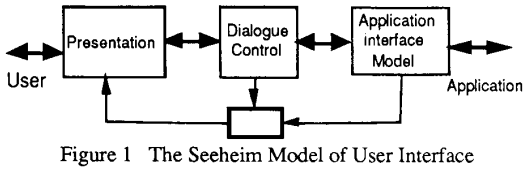


Figure 1 The Seeheim Model of User Interface

In the Seeheim Model of user-interface, 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.

From a functional view, GUIs are similar to communication programs or other highly 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.

### 3.1 Highly interactive and modeless

Like other highly interactive systems, GUIs are largely mode-free [33]. 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 quickly move 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 interaction could easily lead the user-interface into a state that has not been foreseen by the designers. There is an obvious need for the formal specification of user interfaces, where a sound mathematical base and precise denotations allow a user interface design to be checked for consistency, completeness, and reachability.

### 3.2 Screen presentation

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 I/O functions (e.g. selecting a menu option) rely on this accurate mapping between screen positions and the internal (x,y) coordinate representations used in the software. This is a main functional difference 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) are to 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.

### 3.3 Display objects and interaction functions

In graphical user interfaces, display objects are given interaction functions. For example if we move the mouse pointer 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. [39] has shown that a proper enumeration of program paths is a non-trivial problem and is vital to structural (code based) testing. For our GUI validation work, we have developed an algebraic notation for the enumeration of objects and functions. An example is given in section 7 where we also discuss how path algebra can be used to select interaction sequences for testing.

### 3.4 Basic Interaction Components

Although window user interfaces are highly interactive and modeless, so far only a few basic types of interaction components are in common use. Our survey has revealed the following common interaction components :

- Terminal emulation windows
- Icons
- Menus (Pop-up or pull-down, and variants such as command and radio buttons, check boxes)
- Text editing windows
- Scroll bars (sliders, dials and other "control panel" component variants)
- Dialogue boxes (combination of command buttons and text editing fields, which may block processing until the dialogue box is cleared)

The identification and breaking down of a GUI into basic interaction components is a process of functional decomposition for validation purposes [17]. In this way we have reduced a large problem of validating the whole user interface into smaller

problems of validating the basic interaction components that make up the user interface. The interaction (or display) objects that we have discussed earlier can be looked at as instances of the different types of interaction components. Interaction objects that are instances of the same basic component are expected to behave in similar (or even equivalent) manners.

### 3.5 Window Management Functions

Window management is concerned with the arrangement of display objects on the screen. A literature survey has revealed that most window managers on different systems seem to share a similar set of features [32]. A list of the basic window management functions are :

- Move display objects (windows, icons, menus, etc.).
- Resize display objects.
- Create and destroy display objects.
- Iconize windows.
- Hide and raise overlapping windows.

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 repaint 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.

## 4 Structural aspects of window software and testing

The structure of GUIs varies to a large degree depending on the underlying software, such as window systems, toolkits or UIMSs. For example the program interface of a UIMS is of a higher level than that of a window system (see Figure 2). Code based (structural) testing of GUIs has to adapt to the underlying software. The problem is in determining the software level to target tests, so that these tests can be reusable.

### 4.1 GUI Program Code

The modeless nature of GUI user-interfaces is generally implemented in terms of events and call-back routines. Call-back routines are part of the user interface code, which would be given control to handle certain pre-declared events as they occur. The testing of interaction functions (e.g. clicking on a menu option) would in turn test the asynchronous event handling of these call-back routines. Often the main program of a GUI includes a loop waiting for the next event (or user input).

We have observed that window based applications could have a significant increase in its user-interface source code, when compared with the conventional character based version. This becomes apparent when comparing the two versions of the "Hello world" program in C [45]. Additional code is required to open and close windows, set up various window attributes (position, size, colour, etc.), different styles of fonts for textual output, and graphics for display objects (e.g. icons).

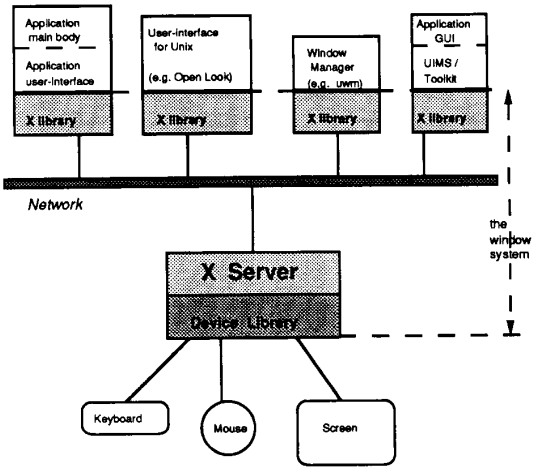


Figure 2 The Structure of the X Window System [37]

### 4.2 Structural testing and functional testing

Research in software testing has traditionally been classified [34] into two main categories : *functional testing* and *structural testing*. Functional testing (also known as Black Box testing) is a testing strategy in which the testers are unconcerned about the internal behaviour of the program under test, they perform testing on their understanding of the intended *functions* of the program. On the other hand, structural testing (also known as White Box testing) is a testing strategy, under which the testers are concerned about the internal *structure* of the program, they derive test data according to their understanding of the logic of the program code.

#### 4.2.1 Static Structural Testing

Most GUI software is seen to contain a large number of library calls to the underlying window system. The X Window System provides more than 200 different routines that can be called from GUI applications. Since these routines are external to the application packages, it gives rise to difficulties with some structural testing techniques like code inspection and source analysis. This is because the correctness of the GUI 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. This information is not always available, nor is it likely to be precisely, unambiguously or formally stated .

For instance, consider the simplest program that uses the X Window System, as show in Figure 3. 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 so as 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 and this argument is applicable to other large or complex Application Program Interfaces.

```
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 An example application program

### 4.2.2 Dynamic Structural Testing

It is possible to take a dynamic approach (as different from the static code analyser approach) to the structural testing of GUIs by attempting to ensure that every line of code is executed during testing. This requires the tester to validate the behaviour of the user interface as each line of code is being executed. 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.

### 4.2.3 Functional Testing

Structural testing tools and techniques are more developed as they are considered to be more reusable. For GUIs, functional testing appears to have the benefit of being generally applicable to different window user interfaces. This is due to the observation [45] that features and basic interaction components provided by different window systems are very similar even across different hardware platforms.

Ideally a user interface should have the same functions disregarding 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.

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 [18]. 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.

## 5 Environmental aspects of GUI testing

A "record and playback" mechanism seems to be a viable approach that has been pursued towards the automation of interactive system testing. One of the early attempts to address the problem of testing interactive systems was the AutoTester project at Wang Laboratories [24]. Other investigations, such as [5], [29], [22], [25] which also proposed the use of Journal Record and Replay (JRR) for user interface testing. There are a small number of commercial products available for JRR, such as "Auto Mac" [30], "Evaluator" [11], "CAPBAK/X" [41] for specific hardware platforms.

### 5.1 Limitations of the JRR approach

It is important to stress the fact that a JRR mechanism would only repeat the tests (or interactions) that a human tester has carried out by hand previously. JRR does not help to solve the problem of test case design. Technically there are three problems that are mixed together :

P1- Knowing what to test (i.e. identifying and selecting items to be tested).
P2- Knowing how to carry out the execution of the test cases (i.e. interaction sequences required).
P3- Finding a tool to automate test case execution.

A JRR tool is only an answer to the third problem listed above. Our approach to solve the first two problems is the use of a formal specification that will identify all items to be tested with pre- and post- conditions for test case generation. Assuming we have the answers to all the three problems above, the next problem is :

P4- How do we know if the GUI being tested is functioning correctly ? (the need for test oracles)

### 5.2 Visual verification

Some research [22], [19] has made attempts to validate GUI screen outputs by comparison with previously recorded good bitmaps. This approach has a number of difficulties :

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

In our approach, the actual visual appearance of display objects are included in our specification to form a special kind of state transition diagram called *WinSTD* . The WinSTD is to be used by human testers for checking visual appearance of objects, as well as for identifying interaction functions for testing.

### 5.3 Input synthesis

Input synthesis is an approach to simulate keyboard and mouse inputs, so as to release human testers from having to execute tests by generating inputs physically by hand. The journal file of a JRR mechanism can provide the first step towards input synthesis. New or variations of the recorded interaction sequences can be produced by providing the facilities for the editing of the content of the journal file [22]. Another step forward would be to generate the contents of the journal file by means other than recording, such as derivation of test cases

from specifications. Release 4 of the X Window System, X11R4 [31] from MIT 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 the server actions in handling user inputs. Basically this proposal gives a programming interface for user inputs to be simulated. However there are synchronization problems with input synthesis that are non-trivial to resolve [19], [21], [8].

## 6 Survey of Formal Specifications for GUIs

In our discussion so far, the need of functional specification for GUI has become obvious for software validation purposes. There are a number of published works on the application of formal specification methods to user interfaces in general ( [16] , [14] ), graphics ([27], [9]), menu-based systems [3], and text processing ([43], [6]).

Confusion often arises concerning the languages and interfaces associated with interactive systems . A working group at the Seillac II workshop addressed this issue([27]) :

> 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 (e.g. ([27], [9]) mentioned above are on the specification of the *program interface*. There are also a number of published work on dialogue specification. [35] suggested using state transition diagrams (STD) to describe interactions. [20], [12], [1], [28] discussed the use of STD, BNF-like grammar, event languages, CSP with *me too*, STD with VDM. In [44] we have surveyed specification methods used in six user interface systems, and concluded that no one single method alone 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 important for testers to be able to see clearly the control flow of interactions.
- **Semantics** specification of operations or functions associated with each interaction step. It is helpful if 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 to satisfy all the above requirements is outlined in the following.

## 7 Developing a strategy for GUI validation

Considering the phases of software engineering life cycle, the proper source for deriving test oracles is the specification. The specification is the global reference point upon which communications and mutual understanding between designers, programmers, testers and users are based. It has been advocated [26] that specifications should be precise, unambiguous and should be reasonably easy to understand. In the case of user interface specification, comprehension is improved when the control flow is clearly presented. For the derivation of expected results for test cases, a formal specification is preferable .

### 7.1 Our approach to GUI specification

For graphical user interfaces, we see one additional requirement in the specification of presentation attributes of display objects. Being aware of the recent interest in visual languages [13], [40], we see no reason why visual information like the appearance of a menu or an icon should 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 [7]). Our contribution to GUI specification is the proposal of a method that includes :

**A) WinSTD**, a set of special State Transition Diagrams 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 (or 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 all functions (arcs) are enumerated with a unique name.

**B) WinSpec**, a language to formally specify all the interaction functions. It employs predicate calculus and set theory to minimize ambiguity and misinterpretation. WinSpec is a model based, formal and mathematical specification approach, similar to Z [42] and VDM [23]. Predicates of pre- and post- conditions are specified for each of the interaction functions, to allow the behaviour of a user interface implementation to be checked. Alternatively a WinSpec can also be used for program verification. WinSpec has special constructs for abstracting GUI interactions in a comprehensible manner.

**C) Algebraic notations** to help us to enumerate and to reason about display objects, interaction functions and sequences. Eventually we intend to apply path algebra to solve our problems of identifying all paths, nodes for test coverage.

Including all display objects in a state transition diagram is the most natural way to link display objects to the flow of interaction as shown in a WinSTD. The human tester can see clearly the expected visual appearance of objects together with their respective interaction functions to be tested. A WinSTD is useful for detecting any missing objects or functions in the design or implementation. Apart from testing, often pictures of display objects (e.g. menus) have to be made available in documents like user manuals. A WinSTD could also be useful for users to receive earlier training and evaluation of the interface. With a WinSTD, human testers will 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 JRR mechanism. Practically a WinSTD can be made easily as most window systems can produce screen dumps on paper.

Alternatively the user interface designer could produce design drawings of display objects using a drawing tool (e.g. MacDraw). The tester has to add the arcs joining objects, and then identify and enumerate all objects and functions for testing.

### 7.2 FIS and FES

In order to reduce the need for testing a large number of interaction sequences, we have introduced the concepts of *Functionally Equivalent Sequences* (FESs) and *Functionally Independent Sequences* (FISs). **FIS**s are two different interaction sequences that are independent of one another such that each can be carried out before or after the other with no effective difference in the final outcome as far as software testing is concerned.

For example the MacWrite menu bar has a number of options. The two options "Font" and "Style" both have a relative large number of sub-options. By our definition of FIS, the two sets of sub-options are independent. This means that we don't have to test all possible combinations of the two sets of sub-options. Let say if there are 20 different names of fonts and 10 different styles, we will be testing 20 + 10 = 30 cases, instead of 20 x 10 = 200 cases. (We need, of course, to check that our concept of applying FIS to reduce test cases is reliable in practice.)

**Functionally Equivalent Sequences** (FES) are interaction sequences that would produce the same outcome if any one of them is selected for execution. One main application of FES is to select the shortest equivalent sequence to reach to a certain object or function for testing.

An example of FES is the movement of the mouse pointer. We could, in most cases of interaction, move the mouse pointer to go over all different locations on the screen (e.g. 512 x 512 points) and then settle on one location where we click the mouse button. (Formally expressed as : Loc(x1,y1) o Move_to(1,1) o Move_to(1,2) o ... Move_to(512,511) o Move_to(512,512) o Move_to(x2,y2) o Mouse_click )

In effect we could have moved the mouse pointer straight onto its final location without going round everywhere (Formally : Loc(x1,y1) o Move_to(x2,y2) o Mouse_click ), the outcomes are the same. We say the two interaction sequences are equivalent :

Loc(x1,y1) o Move_to(1,1) o Move_to(1,2) o ... Move_to(512,511) o Move_to(512,512) o Move_to(x2,y2) o Mouse_click
= Loc(x1,y1) o Move_to(x2,y2) o Mouse_click

Another application of the concept of FES is in the testing of multiple instances of the same interaction component. For example we could have a number of terminal windows (different instances of the same terminal emulation program) on a workstation screen. We would only need to perform an exhaustive set of interaction sequences on one of the terminal windows. However it is necessary to have multiple terminal windows for testing window management functions such as the overlapping of windows.

### 7.3 An example of GUI specification

In the following we give an example of our formal visual specification approach for a simple "Logon" GUI. The following diagram is the WinSTD for the Logon interface :

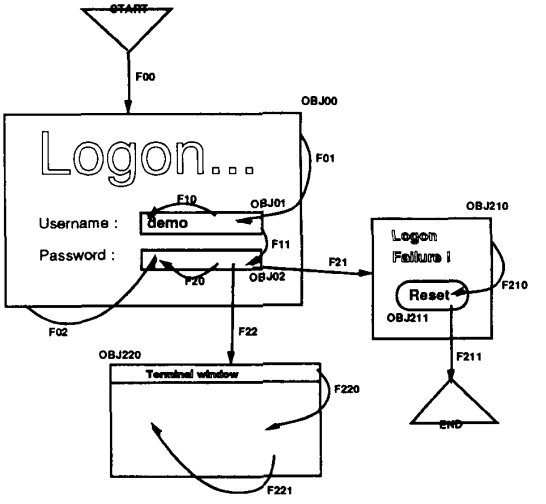


Figure 4 WinSTD for Logon interface

**WinSTD notation** . Each display object is assigned a name. On the top interaction level we have :
OBJ00 = {OBJ01, OBJ02}
OBJ00 - a composite object for the Logon dialogue box
OBJ01 - a text editing field, an object within OBJ00
OBJ02 - a non-echoing editing field, within OBJ00

If there are more display objects, the enumeration carries on as : OBJ03, ... OBJ09, OBJ0A, OBJ0B, ... OBJ0Y, OBJ0Z . Display objects appear at a later stage of interaction are denoted with a longer subscript (e.g. OBJ210 as from OBJ02). A subscript with a trailing "0" denotes a composite object (OBJx0 ).

The interaction functions associated with each of the objects are identified as :
F01 - Move mouse pointer into OBJ01
F02 - Move mouse pointer into OBJ02
F10 - Keyboard inputs in OBJ01
F20 - Keyboard inputs in OBJ02
F11 - Carriage-return input in OBJ01
F21 - Carriage-return input in OBJ02

Interaction sequences can be expressed in this algebraic notation, e.g. :
F01 o F10 o F11 (Function F01 then F10 then F11)

**WinSpec notation**. In addition to mathematical symbols, a number of special notations have been introduced in our WinSpec language to abstract useful I/O details of user interactions :

| | |
|---|---|
| kb? | Keyboard input, e.g. kb?=<CR> is a carriage-return input. |
| mb? | Mouse button input, |

e.g. mb?=Click is a mouse button click.

mp? Mouse pointer input,
e.g. mp?=[.] OBJxx means pointer is inside OBJxx
mp?=.[] OBJxx means pointer is outside OBJxx .

| | |
|---|---|
| OBJxx Visible | OBJxx is visible on screen (see WinSTD for visual appearance ). |
| OBJxx HiLit | The border of OBJxx is highlighted (thicken). |
| OBJxx DeHiLit | The border of OBJxx is normal. |
| -OBJxx | OBJxx is not visible on screen. |
| -Fxx | The reverse of function Fxx . |
| Vxx | Vxx is the global variable associated with the state for OBJxx . |

We have extended the Seeheim Model to show the user interface and the main body of the application communicating through messages. GUI validation is to check these application messages together with visual outputs (specified in WinSTDs).

App_Msg_Sent (MSGxx)
A message MSGxx has been sent to application. This is a predicate that will become true when a message buffer is filled and a ready flag is set :
(Msg_Buf=MSGxx) is true AND (Msg_ToApp_Ready=1)

App_Msg_Recvd (MSGyy)
A message MSGyy has been received from application, as :
(Msg_Buf=MSGyy) is true AND (Msg_FrApp_Ready=1)

Λ-> Temporal logical and,
e.g. mp?=[.]OBJ01 Λ-> mb?=Click
means mp?=[.]OBJ01 "and then" mb?=Click

! Comments, e.g. ! No visible objects

Specification for function F00 :
Pre-conditions : -OBJn , ∀ n ε N , the set of natural numbers ! No visible objects
Λ-> kb?=<CR>
Post-conditions : OBJ00 Visible

Specification for function F01 :
Pre-conditions : mp?=[.] OBJ01
Post-conditions : OBJ01 HiLit

Specification for function F10 :
Pre-conditions : F01 Λ-> kb?
Post-conditions : OBJ01=kb? Λ V01=kb?
! V01 holds username i/p from kb?
! OBJ01 echoes i/p from kb?

Specification for function F11:
Pre-conditions : (F01 Λ-> kb?=<CR>)
V mp?=[.] OBJ02
Post-conditions : OBJ01 DeHiLit Λ OBJ02 HiLit

Specification for function F02 :
Pre-conditions : mp?=[.] OBJ02
Post-conditions : OBJ02 HiLit

Specification for function F20 :
Pre-conditions : F02 Λ-> kb?
Post-conditions : OBJ01="" ! No password echo
Λ V02=kb? ! V02 holds password i/p from kb?

Specification for function F21 :
Pre-conditions : F02 Λ-> kb?=<CR>
Post-conditions : App_Msg_Sent (V01, V02) Λ->
IF App_Msg_Recvd ("Logon failure")
THEN OBJ210 Visible

Specification for function F22 :
Pre-conditions : F02 Λ-> kb?=<CR>
Post-conditions : App_Msg_Sent ( V01, V02) Λ->
IF App_Msg_Recvd ("Logon OK") THEN
OBJ220 Visible

Specification for function F210:
Pre-conditions : mp?=[.] OBJ211
Post-conditions : OBJ211 HiLit

Specification for function F211 :
Pre-conditions : F210 Λ-> mb?=Click
Post-conditions : -OBJn , ∀ n ε N , the set of natural numbers ! No visible objects

### 7.4 Interaction sequence and path algebra

For the Logon user-interface specified above, imagine that the "Logon failure" dialogue box has been changed to include the display of an additional information about the reason for logon failure : "Invalid username" or "Invalid password". We have to retest this dialogue box, which is precisely identified in our notation as OBJ210. Path algebra is used to list the possible paths to reach to OBJ210 as follows. (See [46] for details about *path algebra for GUI.*)

Normal path:
F00 o F01 o F10 o F02 o F20 o **F21 --> OBJ210**

Skip username entry, F10 :
F00 o F01 o F02 o F20 o **F21 --> OBJ210**

Skip password entry, F20 :
F00 o F01 o F10 o F02 o **F21 --> OBJ210**

Skip username field, F01 and F10 :
F00 o F02 o F20 o **F21 --> OBJ210**

Skip F01,F10 & password entry,F20 :
F00 o F02 o **F21 --> OBJ210**

The last path in the above list is the shortest path to reach to OBJ210 for the display of the "Logon failure" dialogue box. This interaction sequence should test the "Invalid username", according to the revised specification for F21. To test the "Invalid password" display, the 3rd path in the above list should

be followed, where F10 should be the entry of a valid username. The revised formal specification for F21 is given here :

```
Pre-conditions :   F02 ∧-> kb?=<CR>
Post-conditions :  App_Msg_Sent (V01, V02) ∧->
                   IF App_Msg_Recvd ("Invalid
                   username") THEN (OBJ210 Visible
                   ∧ OBJ212="Invalid username")
                   ELSE
                   IF App_Msg_Recvd(
                       "Invalid password")
                   THEN (OBJ210 Visible ∧
                       OBJ212="Invalid password")
```

We have added a new object OBJ212 within the dialogue box object OBJ210. The formal specification for F211 remains the same as before.

### 7.5 Test case selection

In the following three test cases for validating the "Logon" interaction are given. We begin by following the WinSTD in enumerating and selecting objects, functions and messages for testing. Then the use of WinSpec pre-conditions is demonstrated, for designing the required interaction steps for test cases. The post-conditions are used as oracles to validate the Logon interface implementation. Our strategy is to select the minimum number of interaction sequences for :

- 100% display object coverage.
- 100% function coverage.
- 100% application message coverage.

In these test cases, we have not selected all keyboard keys for input testing, because the ability to input all keyboard keys belong to the underlying window system and device driver. We are interested in the GUI's ability to pass keyboard inputs to other parts of the application program.

It is also important to note that we are not testing the other part of the application that actually undertakes the authorization check of the username and password against the authorization database. This is the reason why we have only selected one instance for each of the three possibilities :

- Invalid username
- Invalid password
- Logon ok

---

Test case (A)

| Required input sequences | Function tested | Expected Outcome (to be checked) |
|---|---|---|
| kb?=<CR> | F00 | OBJ00 Visible |
| mp?=[.] OBJ01 | F01 | OBJ01 HiLit |
| kb?=<CR> | F11 | OBJ01 DeHiLit<br>OBJ02 HiLit |
| mp?=<CR> | F21 | App_Msg_Sent ( V01="",<br>V02="")<br>OBJ210 Visible<br>OBJ212="Invalid<br>username" |
| mp?=[.] OBJ211 | F210 | OBJ211 HiLit |
| mp?=Click | F211 | -OBJn , ∀ n ε N<br>! No visible objects |

---

Test case (B)

| Required input sequences | Function tested | Expected Outcome (to be checked) |
|---|---|---|
| kb?=<CR> | F00 | OBJ00 Visible |
| mp?=[.] OBJ01 | F01 | OBJ01 HiLit |
| kb?="DemoUser" | F10 | OBJ01="DemoUser" |
| kb?=<CR> | F11 | OBJ01 DeHiLit<br>OBJ02 HiLit |
| kb?=<CR> | F21 | App_Msg_Sent (<br>V01="DemoUser",<br>V02="")<br>OBJ210 Visible<br>OBJ212="Invalid<br>password" |
| mp?=[.] OBJ211 | F210 | OBJ211 HiLit |
| mb?=Click | F211 | -OBJn , ∀ n ε N |

---

Test case (C)

| Required input sequences | Function tested | Expected Outcome (to be checked) |
|---|---|---|
| kb?=<CR> | F00 | OBJ00 Visible |
| mp?=[.] OBJ01 | F01 | OBJ01 HiLit |
| kb?="DemoUser" | F10 | OBJ01="DemoUser" |
| kb?=<CR> | F11 | OBJ02 HiLit |
| mp?=.[] OBJ02 | -F02 | OBJ02 DeHiLit |
| mp?=[.] OBJ02 | F02 | OBJ02 HiLit |
| kb?="DemoPass" | F20 | OBJ02="" ! No echo |
| kb?=<CR> | F22 | App_Msg_Sent(<br>V01="DemoUser",<br>V02="DemoPass")<br>OBJ220 Visible |

The Logon user interface is small. We have applied the same validation approach to a larger user interface (Xmail) developed under the X Window System [46] and find it useful for systematic testing of user interface functions and uncovering errors.

## 8 Conclusion

In this paper we have undertaken an analysis of the problems concerning the validation of GUI software. Functionally GUI software is highly interactive, modeless, it handles position dependent and window based I/Os, graphical information and direct manipulation. Structurally GUIs are closely coupled and dependent on underlying software with large numbers of external routine calls, and it handles asynchronous events by call back routines.

The theoretical concept of a Finite State Machine (FSM) forms the basis of State Transition Diagrams (STDs) which is very suitable for describing the flow of interactions in a GUI. An algebraic notation for enumerating display objects and functions

can also benefit from mathematical concepts such as path algebra for identifying test coverage. Concepts of functional decomposition, FESs and FISs can help the design and selection of test cases.

We have argued that a functional testing approach is suitable for GUI software and conclude that the derivation of test cases from formal specification is an important step towards the automation of GUI validation. Our research direction is to pursue this strategy of GUI validation by tool implementation and to extend our specification to properly introduce temporal logic and concurrence.

The tool implementation consists of two parts : generate test inputs from WinSTDs, and extract expected outputs from WinSpecs. So far we are able to produce an internal representation of a WinSTD that extracts information for invoking interaction functions on objects. We have also started work on a WinSpec interpreter. Eventually we aim to generate test cases (scripts of mouse positions, keyboard and mouse button inputs, and expected outputs) automatically from specifications and feed them into a JRR tool to automate testing.

**Acknowledgement**
S.W.L.Yip is funded by grants from the UK Science and Engineering Research Council (SERC) and the British Telecom Research Laboratories (BTRL). We gratefully acknowledge their support.

**References**

[1] Alexander H., "Formally-Based Tools and Techniques for Human-Computer Dialogues", PhD. Thesis, Stirling University 1986.

[2] Special Report on "Major Vendors Agree on Window Standard", The Anderson Report, February 1987, Page 5-6, Anderson Publishing Company .

[3] Arthur J.D., "Towards a Formal Specification of Menu-based systems", The Journal of System and Software 1987.

[4] Bobrow D.G., "Expert Systems: Perils and Promise", Communications of the ACM, Sept. 1986, p880-894.

[5] Casey B.E., Dasarathy B., "Modelling and Validating the Man-Machine Interface", GTE Labs.,Software Practice and Experience 12(6) p558-569, 1982.

[6] Chi U.I., "Formal Specification of User Interface: A Comparison and Evaluation of 4 Axiomatic Approaches", IEEE Trans. Software Eng., 11(8), p671-685, 1985.

[7] "Serpent Overview", SEI Carnegie Mellon University, August 1989.

[8] Coutu D., "Automating X Window System testing by User Synthesis", Digital Equipment Corp., X Technical Conference, Jan 1990. (Abstract only).

[9] Duce D.A., Fielding E.V.C., "Towards a formal specification of the GKS output primitives", Proc. Eurographics '86, p307-324, 1986.

[10] Ehrlich K., et al., "Incorporating usability studies & Interface design into Software development" , Sun Microsystems Inc. 1989

[11] Elverex, "Evaluator" - Sales Literature, in Personal Computer Magazine, August 1989 .

[12] Green M., "A Survey of Three Dialogue Models", ACM Trans. Graphics, July 1986, p244-275.

[13] Harel D., "On Visual Formalisms", Comms. of ACM, 31(5), p514-531, May 1988.

[14] Harrison M., Thimbleby H. (eds), "Formal Methods in Human-Computer Interaction", Cambridge Univ. Press 1990.

[15] Hartson R.,"User-Interface Management Control and Communication", IEEE software, Jan 1989, p62-70.

[16] Hekmatpour S., Ince D., "Software Prototyping, Formal Methods and VDM", Addison-Wesley 1988.

[17] Howden W.E., "Functional Program Testing & Analysis", McGraw-Hill 1987.

[18] Ince D., Hekmatpour S., "An evaluation of some black-box testing methods", Technical Report No 84/7, Computing Discipline, Faculty of Mathematics, Open University.

[19] Islam N., Ingoglia J.P., "Testing Window Systems", Proc. 28th Annual Technical Symposium "Interfaces : System and People Working together", Washington D.C. ACM Chapter.

[20] Jacob R.J.K., "A Specification Language for Direct Manipulation User-interfaces", ACM Transactions on Graphics, Oct. 1986, p283-317.

[21] Jamison A., "Enhancing the Input Synthesis Extension with Xtrap", Digital Equipment Corp., X Technical Conference, Jan 1990. (Abstract only).

[22] Johnson M.A., "Automated Testing of User Interfaces", p285-293, Pacific North West Software Quality conference 1987.

[23] Jones C.B., "Systematic Software Development Using VDM", 2nd edition, Prentice-Hall 1990.

[24] Leach 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 August 1983, p143 - 147.

[25] Lewis 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.

[26] Liskov B., Guttag J., "Abstraction and Specification in Program Development", MIT Press, 1986.

[27] Mallgren W.R., "Formal specification of interactive graphics programming languages", PhD. dissertation, Univ. Washington, Seattle, 1981.

[28] Marshall S.L., "A Formal Description Method for User Interfaces", PhD. thesis,University of Manchester 1986.

[29] Maurer M.E., "Full-screen testing of interactive applications", IBM Systems Journal, 22(3), p246-261, 1983.

[30] Microsoft Corporation, "AUTO MAC III, Macro Recorder" Reference Manual, 1988.

[31] Various documents in X11R4 distribution tape, MIT 1989.

[32] Myers B.A., "A Taxonomy of Window Manager User Interfaces", IEEE Computer Graphics and Applications, Sept. 1988, Page 79-109 .

[33] Myers B.A., "User-Interface Tools: Introduction and Survey ", IEEE Software, Jan 1989.

[34] Myers G.J., "Art of Software Testing", John Wiley & Sons 1979 .

[35] Parnas 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.

[36] Pfaff G.E. (ed), "User Interface Management System" (Proceedings of Workshop on UIMS, Seeheim, Germany, Nov. 1983), Springer-Verlag 1985.

[37] Scheifler R.W. , Gettys J., "The X Window System", ACM Transactions on Graphics, April 1986, Vol. 5 No. 2.

[38] Shneiderman B., "Direct Manipulation: A Step Beyond Programming Languages", Computer, August 1983 p57-69.

[39] Shooman M.L., "Software Engineering", McGraw-Hill 1983.

[40] Shu N.C., "Visual programming: Perspectives and approaches", IBM System Journal, 28(4), p525-547, 1989.

[41] Software Research, "CAPBAK/X - Test Capture/Replay for X Windows, Technical Specifications", Software Research Inc., May 1990, San Francisco, USA.

[42] Spivey J.M., "An introduction to Z and formal specifications",Software Engineering Journal, Jan 1989 p40-50.

[43] Sufrin B., "Formal specification of a display-oriented text editor", Sci.Comput.Program., Vol. 1, p157-202, 1982.

[44] Yip S.W.L., "A survey of 6 user interface systems in search for a specification approach suitable for deriving test cases", unpublished manuscript.

[45] Yip S.W.L., Robson D.J., "User Interfaces and Software Maintenance", submitted to the Journal of Software Maintenance, Sept. 1990.

[46] Yip S.W.L., "Functional Testing for Graphical User Interfaces (Test Cases for Xmail)", Technical Report 5\90, Computer Science Dept., University of Durham.

## Appendix A Definition of terms

**A Graphical User Interface (GUI)** is the use of interactive display objects such as windows, icons, Popup (or pulldown) 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 graphical or window user interface to distinguish it from the traditional textual command line interface.

**Window systems** provide the underlying window graphics libraries and device drivers for the construction of window or graphical user interfaces (e.g. X [37]).

**Toolkits and UIMSs**, see section 1.

**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 [15]. Dialogue separation is crucial for easy modification and maintenance of user interfaces, and could also increase the portability of software packages.

**Journal Record and Replay (JRR)**, see section 5.

**Input synthesis, visual verification**, see section 5.

**Software testing** is the execution of a program with the intent of finding errors [34].

**Program verification**, use mathematical induction to prove an implementation is in accordance with its formal specification.

**Validation** involves checking the software against its requirements or specifications.

A **test case** is a set of tests designed by human testers, it consists of both a detailed description of the input data and a precise description of the expected (or correct) output .

A **test oracle** is someone who could give a precise and authoritative description of the expected (or correct) output and behaviour of the program when executed with a certain test case.

**Functional Testing, Structural Testing**, see section 4.2 .

**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 [17].

**Reachability** of a specification requires every state satisfying the state definition can be reached by some sequence of operations applied to the initial state. .