A Graphical User Interface Testing Methodology

Ellis Horowitz and Zafar Singhera

Department of Computer Science
University of Southern California
Los Angeles, California 90089-0781

USC-CS-93-550

Abstract

Software testing in general and graphical user interface testing in particular is one of the major challenges in the software community today. Graphical user interface (GUI) testing is inherently more difficult than traditional, command line interface testing. A huge number of objects in a GUI, different look and feel of objects, a large number of parameters associated with each object, progressive disclosure, complex input from multiple sources and graphical output are some of the factors which make GUI testing different from traditional software testing. Existing techniques for creation and organization of test suites need to be adapted/enhanced for GUIs, and new techniques are desired to make the creation and management of test suites more efficient and effective.

This document provides a methodology to create test suites for a graphical user interface. The methodology organizes the testing activity into various levels. The tests created at a particular level can be reused at higher levels. The methodology extends the notion of modularity and reusability to the testing phase also. The organization of the created test suites resembles closely to the structure of the graphical user interface under test.

Graphical user interfaces (GUIs) have changed the use and perception of computers. They provide an easier way of using various functions of the application by organizing them in a hierarchy of options and presenting only the options which make sense in the current working domain. It helps users in concentrating more on the problem instead of putting efforts in remembering all the options provided by the software application being used to solve the problem, or searching for the right option from a huge list of options provided by the application, whether those options are valid for the current working domain or not. Graphical user interfaces divide the standard user actions and working paradigms into various categories which are presented graphically in a way that reflect their functionality. These features make graphical user interfaces more easy to use but putting all these features in an application makes its development, testing and maintenance much more difficult. Fifty to eighty percent of the code in average applications with a graphical user interface is related to the GUI features of the application. The nature of GUI applications, their asynchronous mode of operation, nontraditional input/output and hierarchical structure for user interaction make their testing significantly different from traditional software testing.

This paper presents a methodology for testing of graphical user interfaces. Section 1.0 provides some recommendations which help in organizing an effective and efficient activity for testing a GUI application. Section 3.0 describes the major steps of the methodology. Section 3.0 introduces a sample X application, called Xman, which will be used to demonstrate the effectiveness of the suggested strategy. Section 4.0 demonstrates the testing methodology when formal specification of the application is not available. Section 4.0 demonstrates the testing methodology when the formal specification of the application is available. This section also describes the way statistics are collected during a testing activity and how those can be used to improve the quality of the testing. Section 6.0 mentions the situations when a modification to the application under test might require tuning or recapturing of some of the test scripts. Section 7.0 provides the concluding remarks.

1.0 Some Recommendations for GUI Testing

This section provides some recommendations for planning a testing activity for a graphical user interface.

- Every element of the graphical user should be considered as an object, which can be addressed by a particular name. The objects should have a well defined set of parameters and its response to the outside events should also be well defined.
- The testing activity should be planned carefully around a formal model of the application under test. This model should be powerful enough to provide automatic test generation and coverage analysis.
- Testing of a graphical user interfaces should be performed in a layered fashion. The list of objects which will be tested at a particular level, are either built dynamically while testing at lower levels or from the specification of the application under test. Each level tests all the objects in its associated list. The list of objects for the lowest level are the basic widget, supported by the underlying toolkit. While the highest level considers the entire application as a single object. The decision about the number of testing levels and the qualifying criteria for a particular testing level must be made before creating any tests.
- The tests should be organized as a hierarchy of scripts, i.e. files containing commands to simulate user actions and verify results. This hierarchy should closely correspond to the object hierarchy of the application under test. Each directory in the hierarchy holds scripts which are related to a particular object and its descendents. The individual scripts should be as small as possible and should test one particular feature of an object. However if the features are related and simple, they can be grouped together in the same script.
- Each script should begin with a clear and precise description of the intended purpose of the script and the state of the application required for its proper execution. A script should be divided into three sections. The first section of the script builds the environment required to test the particular feature of the application, the script is intended for. The second section of the script tests the intended feature of the application being tested by the script. The third section restores the state of the AUT and the operating environment, to a point which existed before entering the script.
- A script should be created in such a way that some or all the sections of the script can be executed by calling the script from another script. It will provide reusability feature

in testing also.

- Instead of manually capturing or replaying the test scripts, a tool should be used which can capture and/or replay the test scripts automatically and verify the behavior of the application under test. The tool should be capable of addressing an object in the GUI by its symbolic name, instead of its location, dimensions or any other contextual information.
- The data for result verification should be captured in terms of object attributes when possible and only those attributes should be captured which are critical to verify the function of the application, being tested by the current script. When image comparison is unavoidable, the images should be captured with reference to the smallest enclosing object and area of the captured images should not be more than absolutely required. The number of verifications should also be kept to an absolute minimum especially when image comparisons are involved.
- The script commands to simulate user actions during replay and the data for verification of the AUT behavior should be kept separately. This separation is required because the verification data might change depending on the environment while the script commands should be independent of the environment and should be valid across multiple platforms. If script commands and verification data are stored separately then it will be easier to port a test suite across multiple platforms. In fact a good tool should automatically perform the porting from one hardware platform to the other.

2.0 A Methodology for GUI Testing

This section provides a methodology for testing of a Graphical User Interface. The methodology is particularly useful when one has a tool similar to XTester. It follows the recommendations provided in the previous section. The methodology provides two scenarios, i.e. Testing without formal specification of the application under test and Testing with a formal model of the application. Both the scenarios are presented in the following subsections.

2.1 Testing without a Formal Model

Specification of a graphical user interface for testing purposes is a difficult task and requires a significant amount of effort. Sometimes it is not feasible to invest resources in creating the formal specification of the application, hence testing has to be performed without it. The best thing which can be done in such a situation is to incrementally build a test hierarchy for the application, by capturing user sessions in an organized way. Automatic test generation or coverage analysis is not possible without a formal specification of the application under test. The major steps of the methodology to test an application without a specification, are given below:

1. **Initialization:** Make basic decisions about the testing activity. Some of the most important decisions, which must be taken at this point, are:

   - The number of testing levels and criteria to build list of objects for a particular level.
   - Initialize a list of objects for each level which holds the names and some information about the objects of the user interface. The information includes the way the object can be mapped on the screen, the mechanism to unmap it from the screen, and if it has been tested or not.
   - The location of the test suite and its organization.
   - The application resources which will be used during this testing activity.

2. **Building the Initial Object List:**

   - Go through the documentation of the application under test and find all the top level windows, which might appear as starting windows of the application. These windows and their related information is added to the list of objects for the first testing level and marked as tested.

3. **Building Test Suite:** Take the first object from the top of the object list, which has not been tested, and create test scripts for it. The procedure for creating test scripts for a particular object is given in Figure 1. The sub-objects of the object under test are added to the list of objects by scanning the object from left to right and top to bottom. Keep on taking objects from the top of the object list and testing them until all

```
Display the object on the screen and verify its appearance;
If the object has sub-objects
   Add its immediate sub-objects at the top of the list;
If the object qualifies for a higher testing level
   Add it to the list of objects for the higher level;
Send expected events to the object and verify its response;
If a new object appears in response to the event
   if the new object is not listed as a tested object
      Add it to the end of the list;
Pop down the object from the screen;
Mark the object in the list as tested;
```

FIGURE 1. Strategy for testing without specifications

the objects in the list are marked as tested. When the list associated with a particular level has no untested object, start testing objects from the list, associated with the next higher level. This process continues until all the levels are tested.

4. **Creating Script Drivers:** Write higher level scripts for all the top level windows or any other complex objects, each of the testing levels and for the entire test suite. These scripts will replay all the scripts related to the object and its descendents. The highest level script driver should replay each and every script in the suite.

5. **Testing the Test Suite:** Make sure that all the scripts and script drivers work properly and cover all the features of the application, which needs to be tested. Although one cannot do much automatically to determine the quality of a test suite, in the absence of a formal model of the application under test. However after the creation of the test suite, run the highest level script driver to verify that all the scripts in the suite are capable of proper replay and try to match the features covered by the test suite with those included in test requirements or application documentation.

2.2 Testing with a Formal Model

The strategy to build test scripts without a formal specification, discussed in the previous subsection, puts a lot of responsibility on the person creating those scripts. The strategy also require that the application under test should be running reliably before the capturing of script is even started. The scripts created without formal specification are also vulnerable to any modification in the application, which affects its window hierarchy. It also requires that after making any changes to the application under test, the affected scripts should be located manually and recaptured or tuned to offset the modification in the application. It is also not possible to create test scripts automatically or to get a coverage measure after running a set of test suites. To overcome these drawback and get access to advanced features like automatic test generation and coverage analysis, one has to invest some effort to formally specify the application under test. This sections provides a methodology to test an application when its formal specification is provided or resources are available to build such a specification. The following are the major steps of the testing methodology when a formal specification of the application under test is available.

1. **Building the Model:** Build the User Interface Graph of the application under test. When resources permit, the very first step in testing an application should be to build a formal model of the application under test. XTester provides such a formal model, called User Interface Graph. The User Interface Graph provides information about the object hierarchy in the application. It also provides information about the nature of a particular object and the effects of an event in an object to the other objects in the user interface. The User Interface Graph can be built manually by creating a User Interface Description Language (UIDL) file, or it can be created semi-automatically by steering through the application under test and filling in the missing information about objects. See [2] for more information on UIDL syntax and User Interface Graph.

2. **Initialization:** Make basic decisions about the testing activity. Some of the most important decisions, which must be taken at this point, are:

   - The number of testing levels and qualifying criteria for each testing level.
   - The location of the Test suite and its organization.

- The application resources which will be used during this testing activity.

3. **Build Object Lists:** Build a list of objects for each testing level. After building the formal model, it is possible to build object lists for all testing levels. The procedure for building those lists is to start from the root of the User Interface Graph and perform a post-order walk of the object hierarchy. For each visited node, add it to the object lists associated with levels, for which it qualifies.
4. **Building Test Suite:** Build a test suite. The strategy for capturing scripts without any formal specification, discussed in the previous section, can also be used for capturing scripts when the application has been specified formally. However capturing scripts with formal specifications provides us some additional advantages over the scripts which have been captured without any specification. These advantages include an overall picture of the application under test and hence a more efficient test suite, a test suite which is less affected by the changes in the application, automatic test generation, and coverage analysis.
5. **Creating Script Drivers:** Write higher level scripts for all the top level windows or any other complex objects, each of the testing levels and for the entire test suite. These scripts will replay all the scripts related to the object and its descendents. The highest level script driver should replay each and every script in the suite. These scripts can also be created automatically.
6. **Coverage Analysis:** Once a test suite has been created, it should be replayed in its entirety to determine the coverage provided by the test suite. This coverage should be performed at each level, i.e. the coverage criteria for level 1 will be that it should verify that all the objects in the application has been created, mapped, unmapped and destroyed, at least once and every event expected by an object has been exercised on it, at least once. The coverage criteria for higher levels will be to make sure that all the interactions and side effects among objects which make a composite object at the corresponding level, has been verified.

3.0 An Introduction to Xman

This section introduces a sample application, called Xman, which is used to demonstrate the effectiveness of the methodology, presented in the previous section. Xman is a small application which is distributed with the standard X release. It provides a graphical user interface to the UNIX man utility. It has been developed using the Athena widget set and some of its windows are shown in Figure 1. The following paragraphs briefly describe the functionality of Xman.

When started, Xman displays its main window, called Xman, by default. The main window contains three buttons, Help, Manual Page and Quit. Clicking on the Manual Page button displays a window, called Manual Page. A Help window is displayed when the Help button is clicked. The Quit button is used to exit from Xman.

The Manual Page window is organized into various sections. A bar at the top of the window contains two menu buttons, Options and Sections, in the left half and a Message area in the right. The rest of the area below the bar, called Text area, is used to display the names of available manual pages in the currently selected section, and the contents of the currently selected manual page. Both the names and contents portions of the Text area are resizable and have vertical scrollbars on the left. The Options menu contains the following entries:

Display Directory to display names of manual pages in entire Text area

Display Manual Page to display contents of the currently selected manual page in the entire Text area

Help to display a Help window

Search to display a dialog box to enter the name of a manual page to search for

Show Both Screens to vertically divide the area into two halves, with the upper half showing the directory contents of currently selected man page section and the lower half showing the contents of the currently selected manual page. This option toggles to Show One Screen and also disables menu entries Display Directory and Display Manual Page.

Remove This Manpage removes the Manual Page window from the screen

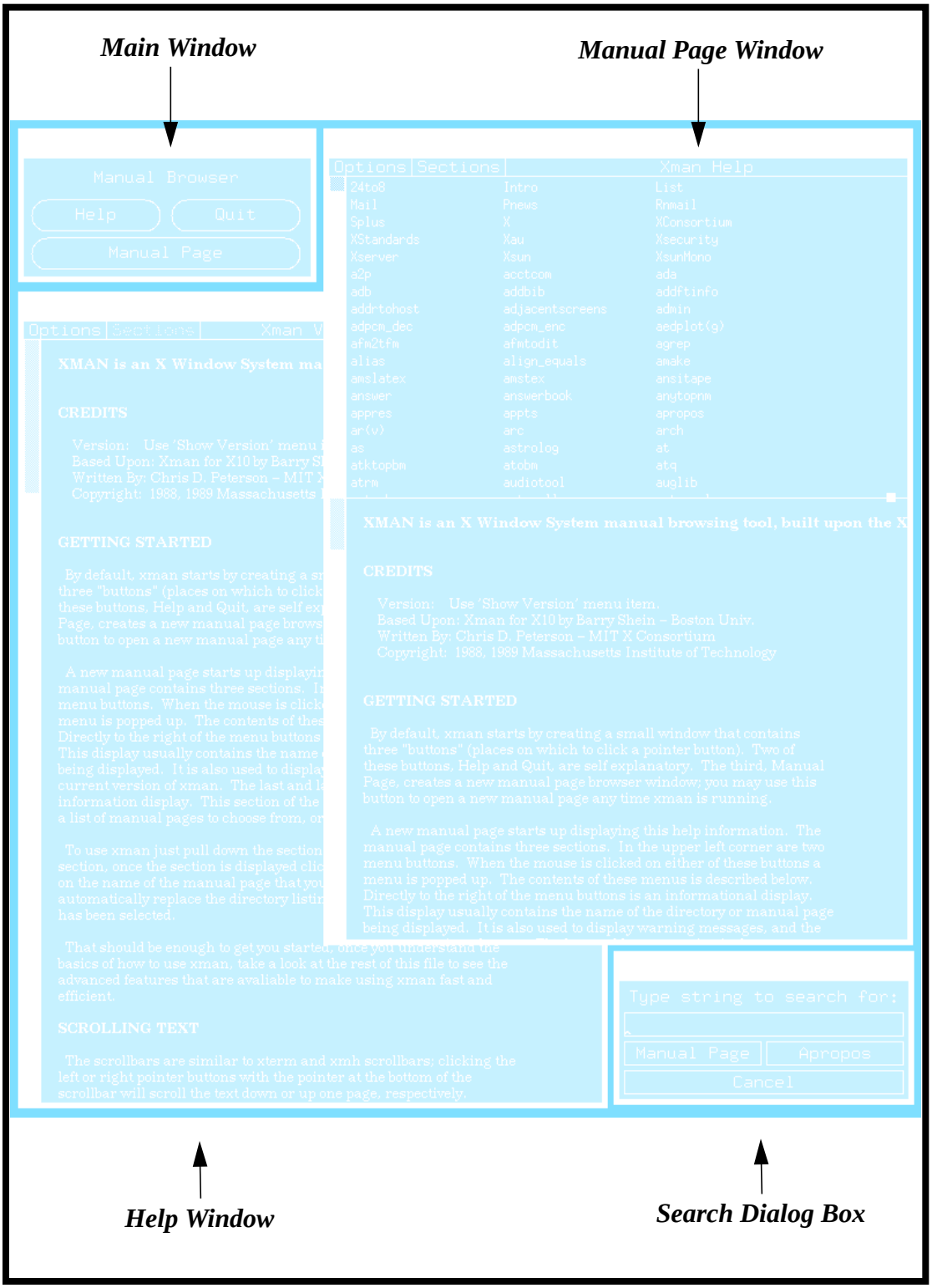

FIGURE 2. Main Windows of Xman

`Open New Manpage` creates another `Manual Page` window

`Show Version` displays the current version of `Xman` in the `Message` area

`Quit` exits from the `Xman` application.

The `Sections` menu contains one option for each manual page section available on the system. The standard options are `User Commands`, `System Calls`, `Subroutines`, `Devices`, `File Format`, `Games`, `Miscellaneous`, and `System Administration`.

The `Help` window displays a limited version of the `Manual Page` window. The window has exactly the same structure as the `Manual Page` window but the `Sections` menu button and the first five options in the `Options` menu are disabled. It displays man page for `Xman` itself. No more than one `Help` window can exist at a time while an arbitrary number of `Manual Page` windows can be created.

The testing of `Xman` has been organized in three layers. The first layer verifies the behavior of individual GUI objects. The second layer verifies the inter-object effects among objects belonging to the same top level window. The third layer verifies the inter-object effects among objects belonging to different top level windows. The following sections provide details of this testing activity.

4.0 Testing without Formal Specifications

This section provides a demonstration for the testing of Xman, when no formal model is available for it. The following subsections demonstrate each step of the methodology, described in Section 2.1.

4.1 Initialization

Number of Testing Levels: As Xman is a fairly small and simple application, so the testing activity is organized in three levels. The very first level tests the individual objects in Xman. The second level verifies the interactions and side effects of objects which belong to the same top level window. The third level verifies the interactions and side effects of objects which belong to different top level windows.

Location of the Test Suite: Let us assume that the root directory for the test suite being captured is `XmanTest`. This directory contains resource file(s) used during testing and the result files are created in the directory, by default. It also has three subdirectories, `Level-1`, `Level-2` and `Level-3`, one for each testing level.

Object List: Let us initialize a list of objects, called `ObjList`. This list will contain information about the objects and is initialized to be empty.

Application Resources: `~/XmanTest/Xman.Defaults` is the file which contains the default application resources of Xman for this testing activity and these resources always remains the same.

4.2 Building the Initial Object List

By default, Xman starts with a top box, called Main Window in Figure 2. However a command line option, `-notopbox`, is available which can be used to bypass the Main Window and display the Manual Page window directly. As Main Window and Manual Page window are the only windows which can appear when Xman is started, so the initial object list contains only these two objects.

4.3 Building the Test Suite

This section provides details on building all the three testing levels for Xman. To make things simpler, we only discuss the creation of test suites related to the `Main Window` of Xman.We shall also ignore any keyboard accelerators to which Xman responds. The test suites for other windows can be created in a similar way. Figure 3 provides the hierarchy of scripts related to the Main Window for all the three testing levels.

4.3.1 First Level

Let us assume that the initial object list is built so that Xman `Main Window` is on the top of the list. We select it as the object under test and create a new directory, called `~/XmanTest/Level-1/Xman` to build test scripts related to its first level testing. The scripts related to XMan Main window object itself display the window on the screen, exercise all the window manager operations on it, and then finally popdown the window. `DspXmanWin.scr` script popups up the `Xman` window and verifies that is looks

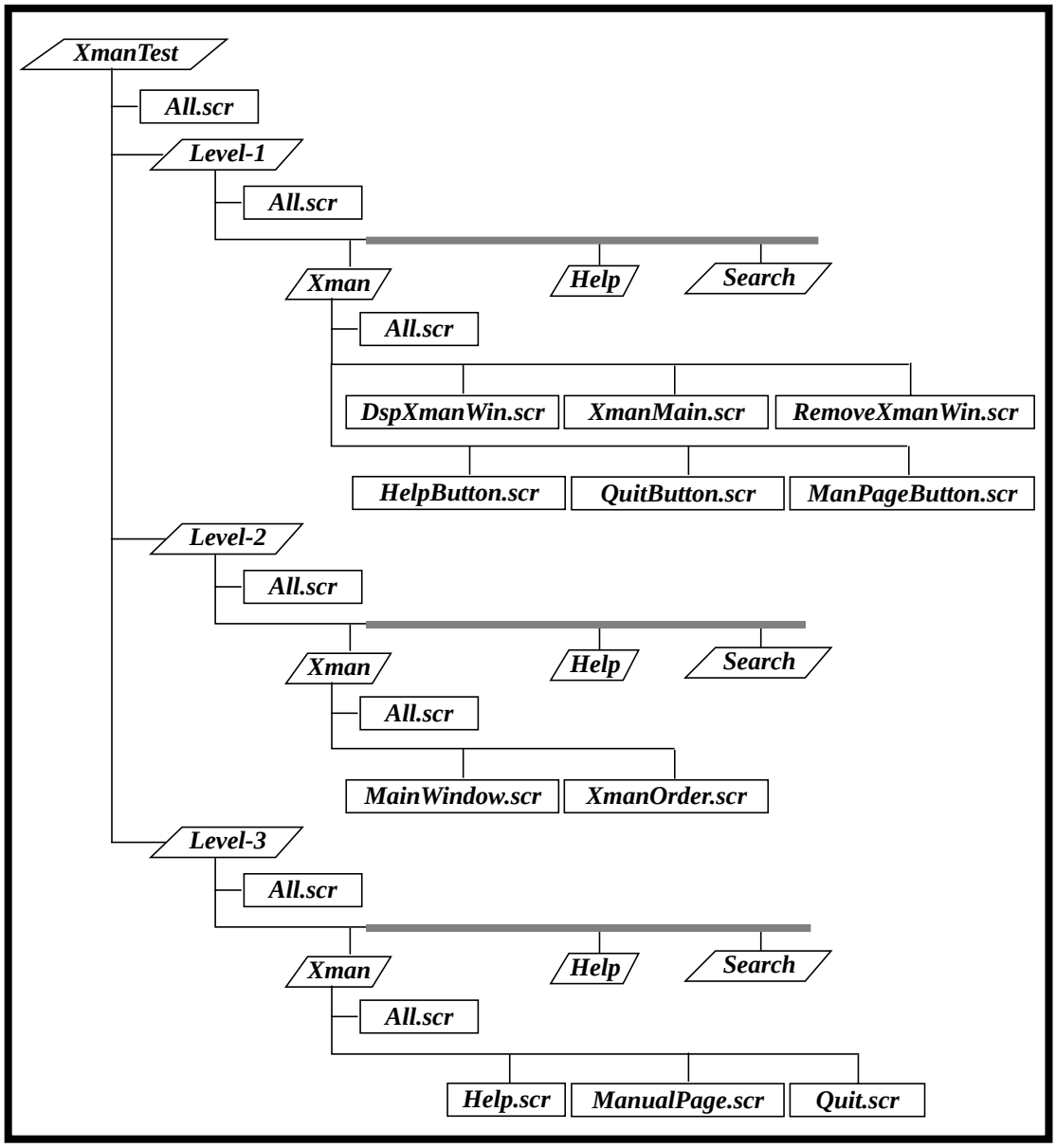

FIGURE 3. Test Suite Hierarchy of Xman

right.`RemoveXmanWin.scr` pops down the `Xman` window. The script `XmanMain.scr` uses `DspXmanWin.scr` in its entering section to display the `Xman` window. It verifies the window manager operations in its core section and then uses `RemoveHelpWin.scr` script in its leaving section to pop down the `Xman` Window. As soon as the `Xman` window pops up on the screen, we see that it contains four objects, i.e. `Manual Browser` label, `Help`

button, Quit button and Manual Page button. Manual Browser label does is a static piece of text and does not respond to any user actions, so we do not need a script for it. The other three objects a active objects and respond to user events so we create one script for each of them.The entering section of each one of these scrips call DspXmanWin.scr to display the Xman Main Window on the screen. The ending section of each of these scripts call RemoveXmanWin.scr to remove the Xman Main Window from the screen. The core section of HelpButton.scr script verifies the behavior of Help button in Xman Main Window when it is clicked on by a mouse button. The core sections of QuitButton.scr and ManPageButton.scr scripts verify the same thing for Quit and Manual Page buttons in Xman Main Window.

4.3.2 Second Level

The object list of second level contains all the top level windows of Xman. As we are considering the Main Window only in this discussion so we assume that it is at the top of the list and is selected for testing. There is not much interaction going on in the objects which belong to the Xman Main Window. The only interaction is the disappearance of the Main Window, in response to a click on the Quit button. So there will be only one script related to the Main Window which will verify that a click on the Quit button actually destroys the Main Window of Xman. This script is called MainWindow.scr and is located in ~/XmanTest/Level-2/. This script also used DspXmanWin.scr and RemoveXman.scr script to display and remove the Main Window from the screen. Another potential script, let us call it XmanOrder.scr, related to the Main Window verifies that the order in which Help or Manual Page buttons are pressed is not significant. No matter the Help button is pressed before or after the Manual Page button, it will display the Help window properly. The same is true for the Manual Page button also.

4.3.3 Third Level

The object list of the third level includes the Root object only and tests any interactions among the top level windows of Xman. Such interactions which involve the Main Window of Xman include display of the Help window and the Manual Page window in response to mouse clicks on the Help and the Manual Page buttons respectively. Similarly it also includes disappearance of all the windows related to Xman in response to a click on the Quit

button. The three scripts provided at this level, i.e. `Help.scr`, `ManualPage.scr` and `Quit.scr`, verify the behavior, related to the corresponding button, mentioned above. This level also might include scripts which verify application behavior, like multiple clicks on the `Help` button do not create more than one `Help` windows while each click on the `Manual Page` button create a new `Manual Page` window.

4.4 Creating Script Drivers

Once all the scripts for Xman has been captured, we need driver scripts so that all the script in the entire suite, all the scripts in a particular testing level or all the scripts related to a particular object can be executed automatically in the desired sequence. For example, we create a script driver, at each testing level, which executes all the scripts created for testing Xman `Main Window` and its descendents, at that particular level. These scripts are `~/XmanTest/Level-1/Xman/All.scr`,`~/XmanTest/Level-2/Xman/All.scr`, and `~/XmanTest/Level-3/Xman/All.scr`, respectively. The script `~/XmanTest/Level-1/All.scr` drive all the scripts created for the first testing level and similarly the other two drivers will execute scripts related to the other two levels. The script `~/XmanTest/All.scr` will drive all the scripts in all the three levels of the test suite.

4.5 Testing the Test Suite

After the creation of the test suite, it is necessary to replay all the scripts in the suite and verify if they work properly and also to make sure that they cover all the features which needs to be tested. Although without a formal specification, it is not possible perform any reasonable automatic coverage analysis but at least the replayed events and the objects which appear during the replay can be matched against application documentation to determine if any object or event has not been covered by the generated test suite.

5.0 Testing with Formal Specifications

This section demonstrates the testing of Xman when we have enough resources to build a formal model for Xman. The following subsections illustrates each step of the methodology, described in Section 2.2.

5.1 Building the Model

When the resources permit, the very step for testing is building a formal model for the application under test. Figure 4 displays a User Interface Graph built for Xman. The `Root` node of the graph represents the root window of the screen. The children of the `Root` node represent the six top level windows of Xman. The nodes at lower levels in the graph represent the descendents of the top level windows. Let us take the main window of Xman as an example. It is represented as `MainWin` node in the User Interface Graph. The child of the `MainWin` node is the `Form` node which acts as a container widget for the buttons and the label in the main window. The `ManPage`, `Quit` and `Help` nodes represent the `Manual Page`, `Quit` and `Help` command buttons in the main window, respectively. The `label` node represents the `Manual Browser` label in the main window. The dark black arc from the `ManPage` node to the `Manual Page` node represents the fact that clicking a mouse button over the `ManPage` button in the main window affects the top level window, called `Manual Page`. The arc from the `Quit` node to the `Root` node represents that a click on the `Quit` button affects all the top level windows of `Xman`. The type of event represented by an arc is reflected by the drawing pattern of the arc. The two arcs mentioned above have the same pattern and represent button clicks. An arc with a different pattern is the arc from the `Text` node to the `search` node. This pattern represents text entry and the arc represents that entering text in the `Text` node affects the `search` dialog box.

5.2 Initialization

All the decision and actions taken at the initialization step, i.e. the number of testing levels and their organization, the location of the test suite and the application default resources, is kept the same as for testing without a formal specification, described in Section 4.1.

5.3 Building Object Lists:

After building the User Interface Graph for Xman, it is possible to build object lists for all levels of testing. This can be done either manually or automatically by scanning each and every node in the User Interface Graph and verifying if it qualifies to be tested on a particular testing level. All the objects in Xman qualify for the first testing level and hence are placed on the list associated with it. The qualifying criteria for the second level is that the object must

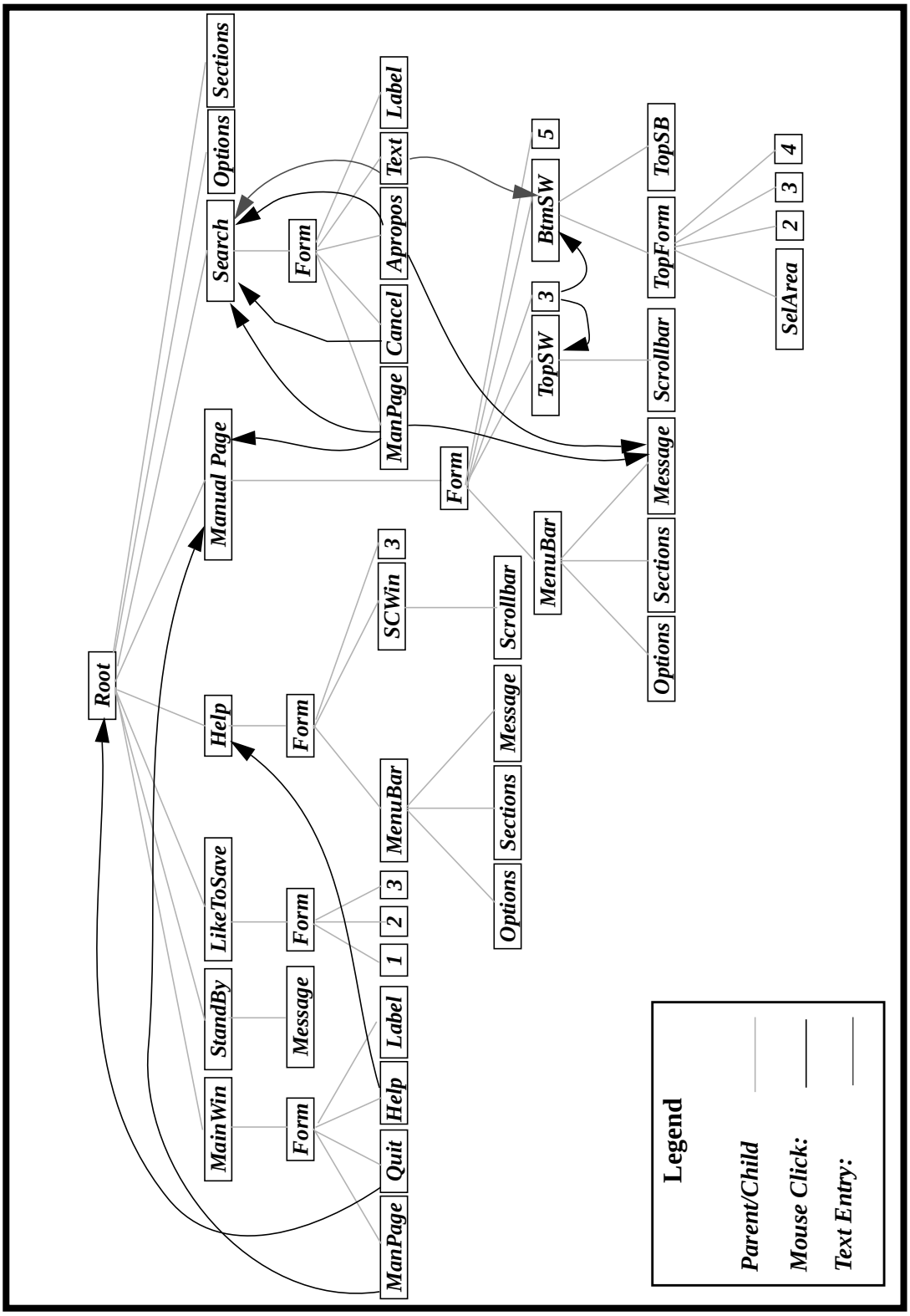

FIGURE 4. Combined Graph for Xman

be a top level window, i.e. its corresponding node must be a child of the root of the User Interface Graph. Some of the objects which qualify for the second testing level are Xman Main Window, Manual Page window, Help window, Search dialog box, etc. The third level treats the entire Xman application as a single object, and its corresponding node, `Root`, is the only object which qualifies for the third level of testing.

5.4 Building Test Suite

The strategy for capturing scripts without any formal specification, discussed in Section 4.3, can also be used for capturing scripts when the application has been specified formally. However capturing scripts with formal specifications also provides us the capability to write the test scripts manually or generate test scripts automatically by using the formal model and the test requirements. All of these three techniques for building a test suite are explained in the following section.

5.4.1 Capturing Scripts

A tool can be used for capturing user sessions for building a test suite, in exactly the same way as for capturing without a formal model, as mentioned in Section 4.3. However the presence of a formal model makes the generated scripts more robust and easy to follow and debug.A formal model also provides the flexibility that the captured scripts can be executed in any order without any conflicts in window names. It becomes easier to modify the test suite in case of a modification to the application under test. In fact in most cases the modification can be reflected in the specification file and the test suite remains valid without making any changes.

5.4.2 Writing Scripts Manually

The formal specification of the application under test also allows the user to write scripts manually, even for the first level of testing. This feature is particularly helpful when the test scripts need to be created in parallel with the application development in order to reduce the total development time. The way it can be organized is to formally specify the graphical user interface of the application once the design is complete. The developers and testers agree on this formal specification and any future changes are properly communicated between the two groups. Having the formal specification of the graphical user interface at hand, the testers can develop the scripts manually in parallel with the application development. Once the object

hierarchy and the behavior of the objects is known, the manual writing of test scripts is as easy as writing a UNIX shell script.

5.4.3 Automatic Test Generation

The formal specification of the application under test also provides capabilities to generate test scripts automatically. A tool can be written which will read the specification for an application and create test scripts for a particular level of testing or the entire test suite. Similar tools can be used to generate test scripts to test all the objects of a particular type or all the user actions of a particular type. For example, such a tool can generate test scripts to exercise clicks on each command button in an entire application. A similar tool can generate a suite of test scripts to verify that all windows in the application under test are displayed on the screen at least once. Another tool might generate scripts to individually select all the options of each menu in the application and verify that the application responds in the expected manner. Another tool might create scripts for selection of multiple menu options and/or command buttons in different sequences to verify the application response. All the above mentioned tools will only create scripts and validation data has to captured by replaying these scripts by using `caprep` [2] or a similar tool, to replay these scripts in `Update` mode.

The logic behind these automatic test generation tools is the same as used in Section 5.0 for manual creation of test suites. The tool starts at the root of the User Interface Graph and builds a list of GUI elements by performing a pre-order walk of the User Interface Graph. During this walk only the arcs which represent parent/child relationship, are considered. After building the list, its entries are taken one by one to create test scripts for them. If the current GUI element, taken from the list, is a top level window then a separate directory is created for it and the scripts for the element and all of its descendents are created in that directory. If the currently selected element belongs to the category for which a script is required, then one is created by following the arcs which represent user actions on the element. Figure 4 presents pseudo code for such an automatic test generator.

5.5 Coverage Analysis

No testing activity is useful unless it provides some coverage measures. These coverage measures reflect the quality of the testing activity. The User Interface Graph provides us a

```
Build the User Interface Graph of the application under test;
Build an object list by a pre-order traversal of the User
Interface Graph.
for each element on the list
do
   If the element is a top level window
      Create a new directory and change to the directory.
      Display the element on the screen.
   fi
   if the element accepts any user events
      Create a script for the element
      for each kind of user event accepted by the element
      do
         Add commands in script to
               Generate the event on the element;
               Verify the effect of that event;
               Undo the effect of the event;
      done
   fi
done
```

FIGURE 5. Pseudo Code for Automatic Test Generator

frame work to determine such a measure of coverage. During capture or replay of scripts, XTester keeps track of the user actions and their effects on individual objects. This information is stored in a `.stt` file upon the completion of the testing activity. Currently the information captured in `.stt` about a particular object includes the number of times it was created, mapped, unmapped and destroyed. It also accumulates the number of times a mouse button or keyboard key was pressed or released over the object. This information helps the user locate any particular objects in the application which have not been created, destroyed or received an expected user event. These figures can also be used for improving the efficiency of the test suite by removing the repetitive testing of the same characteristics, when possible. Figure 5 shows a file created by XTester after replaying a certain test suite for

```
Legend:
CW=Create Window DW=Destroy Window MW=Map Window
UMW=Unmap Window BP=Button Press BR=Button Release
KP=Key Press KR=Key Release

 Object Name                    CW  DW  MW  UMW BP  BR  KP  KR

Xman :                          1   0   2   1   0   0   0   0
Xman*Form :                     1   0   2   1   0   0   0   0
Xman*ManualPage :               1   0   2   1   50  50  0   0
Xman*Quit :                     1   0   2   1   0   0   0   0
Xman*Help :                     1   0   2   1   10  10  0   0
Xman*Label :                    1   0   2   1   0   0   0   0
StandBy :                       28  23  2   1   0   0   0   0
StandBy*Message :               28  23  2   1   0   0   0   0
LikeToSave :                    1   0   0   0   0   0   0   0
LikeToSave*Form :               1   0   0   0   0   0   0   0
LikeToSave*Message :            1   0   0   0   0   0   0   0
LikeToSave*Yes :                1   0   0   0   0   0   0   0
LikeToSave*No :                 1   0   0   0   0   0   0   0
Help :                          1   0   7   7   0   0   0   0
Help*Form :                     1   0   7   7   0   0   0   0
Help*MenuBar :                  1   0   7   7   0   0   0   0
Help*Options :                  1   0   7   7   20  2   0   0
Help*Sections :                 1   0   7   7   0   0   0   0
Help*Message :                  1   0   7   7   0   0   0   0
Help*TextArea :                 1   0   7   7   0   0   0   0
Help*Scrollbar :                1   0   7   7   10  10  0   0
Help.Form.3 :                   1   0   7   7   0   0   0   0
ManPage :                       27  23  27  1   0   0   0   0
ManPage*Form :                  27  23  27  1   0   0   0   0
ManPage*MenuBar :               27  23  27  1   0   0   0   0
ManPage*Options :               27  23  27  1   92  6   0   0
ManPage*Sections :              27  23  27  1   18  2   0   0
ManPage*Message :               27  23  27  1   0   0   0   0
ManPage*TextArea :              27  23  27  1   0   0   0   0
ManPage*Scrollbar :             27  23  27  1   8   8   0   0
ManPage.Form.3 :                27  23  27  1   0   0   0   0
ManPage*DirArea :               27  23  27  1   0   0   0   0
ManPage*DirList :               27  23  27  1   0   0   0   0
ManPage*List :                  27  23  27  1   0   0   0   0
```

FIGURE 6. Statistics file created by XTester

`Xman`. The legend is displayed at the top of the file, to explain symbols used to represent various actions. The next line after the legend provides the heading for the table. Each line in the table provides statistics about a particular object. For example the very first line of the table provide statistics about the Main Window of Xman, named `Xman`. This particular line shows that the object named `Xman` was created once, never destroyed, mapped twice on the screen and unmapped once, by the corresponding test script. It also shows that the corresponding test suite never exercised a button press/release or key press/release events on the `Xman` object.

Tools can be developed to extract user desired information from a `.stt` files. One such tool might read a `.uidl` file to build object hierarchy of the application under test, read a `.stt` file for statistics about a particular testing activity and map those statistics on the object hierarchy so that the user can visually see how many nodes and arcs in the object hierarchy have not been exercised by this particular test suite. Another tool might provide only the objects which did not receive an expected user action. Similarly tools can be written to display information about particular objects or particular user actions. Figure 7 provides the general logic for a coverage analysis tool. The tool will build the User Interface Graph of the application under test by reading in the specification file. After building the User Interface Graph, the information from the statistics file created by a particular test suite will be read, analyzed and displayed, according to the given criteria.

```
Read in the required criteria for analysis;
Build the User Interface Graph of the application under test;
Read in the specified statistics file(s);
for each element in the User Interface Graph
   do
      If the element qualifies for the given criteria
         Display the required information in proper format;
   done
```

FIGURE 7. Pseudo Code for Quality Analyzer

6.0 Invalidation of Test Data

XTester captures information as two entities, script commands and verification data and saves them in different files, `.scr` and `.img`, respectively. This section describes the scenario in which created test script(s) might fail and has to be re-captured or re-written. The following scenarios will invalidate some of the captured scripts.

- The application is modified in such a way that `WM_NAME` property of a top level window is changed. This modification will only effect the scripts which have been captured without a specification and are related to the window whose `WM_NAME` property was changed. The scripts captured with a formal specification remain unaffected, provide that the specifications are also modified to reflect the change in the application.
- The application is changed so that the order or number of children of a particular node is changed. The scripts which were captured without a specification and address objects in the modified hierarchy, will be affected and has to be recaptured or tuned. However the scripts captured with a formal specification remain unaffected, provided the specification is also modified accordingly.
- XTester provides option to build either full or short object names. If the application is modified so that the depth of a hierarchy is changed, then all the full names belonging to that hierarchy will no longer be valid names and has to be modified to reflect the change. However short names will still remain valid.
- If object information has been captured in pixmap mode, then trying to replay the scripts on another workstation, which is not compatible with the workstation on which pixmaps was captured, will give false alarms. The scripts work fine across multiple platforms, however verification data is platform specific in case of pixmaps. An easier way of creating verification data for a new hardware platform will be to replay all the scripts in `Update` mode which will replace the current verification data with the newly available one.

7.0 Conclusion

In this paper, we have provided some guide lines which are useful in planning a testing activity for a graphical user interface. We have presented a methodology for testing a graphical user interface, both when no formal specification of the application under test is not available and when such a specification is provided or resources are available to build such a specification. The paper also demonstrates the use of the suggested methodology to test a sample X application, Xman, with or without a formal specification. It also illustrates how the model is helpful in automatic test generation and coverage analysis. In the end, the paper describes the situations in which the scripts captured by XTester will become invalid.

References:

[1] Ellis Horowitz and Zafar Singhera, “Graphical User Interface Testing”, Proceedings of the Eleventh Annual Pacific Northwest Software Quality Conference, October, 1993.

[2] Ellis Horowitz and Zafar Singhera, “XTester Reference Manual”, 1993.