This Paper was originally presented at the Software Quality Week Conference in May 1998.

Reducing the Time to Thoroughly Test a GUI

Larry Apfelbaum
General Manager
Teradyne Software & Systems Test
44 Simon St.
Nashua, NH 03060
603 879-3555
Fax : 603 879-3075
E-Mail: larry@sst.teradyne.com

John Schroeder
System Engineer
Teradyne Software & Systems Test
44 Simon St.
Nashua, NH 03060
603 879-3697
Fax : 603 879-3075
E-Mail: johns@sst.teradyne.com

Abstract

The testing of the GUI interface is a critical part of the plan for the application's release. Many modern software applications include a Graphical User Interface (GUI), these interfaces can be extremely simple to agonizingly complex depending on the application. Using a graphical model to describe the operational behavior of a GUI in combination with an Automated Test Execution system can dramatically reduce the time required in testing this type of application. This paper describes a straightforward process for GUI testing using a modeling tool which then generates suites of tests verifying the application's operation. The result of this automated approach is reduced testing time and an increase in testing thoroughness.

Outline

1. Review of GUI components and constructs
2. The concept of a stimulus/response pair
3. Test design and generation using models
4. Modeling example of a user interface
5. Summary of benefits

Introduction

To display information and control the software applications through GUIs is a complex subject. A GUI can be extremely simple to agonizingly complex depending on the application. Testing a GUI can be almost as difficult as writing the software. A behavioral modeling approach in conjunction with an automated test and verification tool can simplify and expedite GUI testing.

The GUI operational paradigm has standardized over time. The roots of this standardization are complex, involving user training concerns, cross-platform issues, vendor market dominance, and the software used in GUI development. Teradyne has identified a reoccurring set of basic constructs that have found common usage in GUIs. Examples of these constructs include: Menus, Pushbuttons, and Scrollbars. These constructs are similar on the two major software platforms hosting GUIs: UNIX and MS Windows[1]. In an X/Motif programming environment these constructs are called widgets. In a Wintel windowing environment they are known as window resource objects.

This paper illustrates and explains the modeling strategy for each of these constructs. Given this finite set of models to use as an example, testers can create models of their GUI System Under Test by combining together models of these common constructs. The paper provides a step-by-step method for analyzing the components of a GUI and creating behavioral models.

[1] UNIX is a registered trademark of Unix System Laboratories.
MS Windows is a registered trademark of the Microsoft Corporation.

This paper assumes a familiarity with GUIs, state machine-based behavioral modeling, and automated test execution systems used within the context of GUI testing. *Model Based Testing*, presented at the Software Quality Week 1997 conference or *Black Box Testing* by Boris Beizer (John Wily & Sons, '95) can serve to provide this introduction.

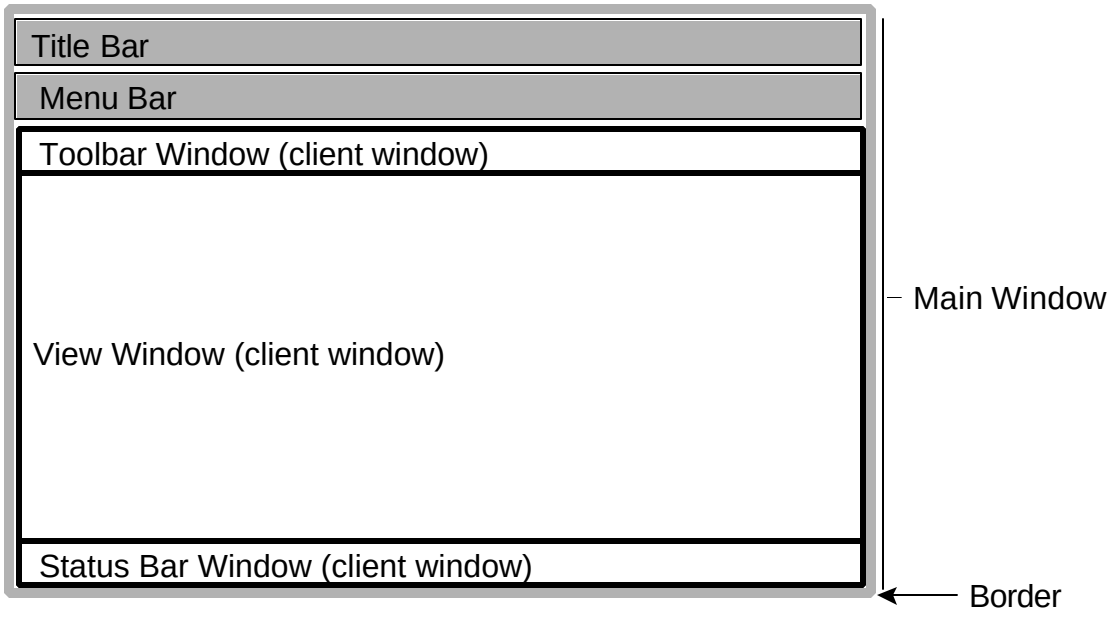
Basic Categories

This section presents the basic GUI components and templates describing how to create and describe models.

There are three basic categories of GUI components: Windows, Menus, and Controls. These three components used in conjunction with each other are assembled into GUIs. Before developing a modeling strategy a brief description of each of these categories is needed to establish a vocabulary.

Windows

A Window is a rectangular area of the user's display device under the control of the System Under Test. A display device is typically a computer monitor or flat panel display. Windows display information to the user as either text or graphics and solicit information from the user through Controls. Every GUI includes a Main Window which may include subordinate client windows within its area. These subordinate windows are arrayed in a hierarchy below the Main Window and inherit its operational policy. The diagram below shows the Main Window in relationship to client windows.



• Figure 1, Typical Window Organization

Menus

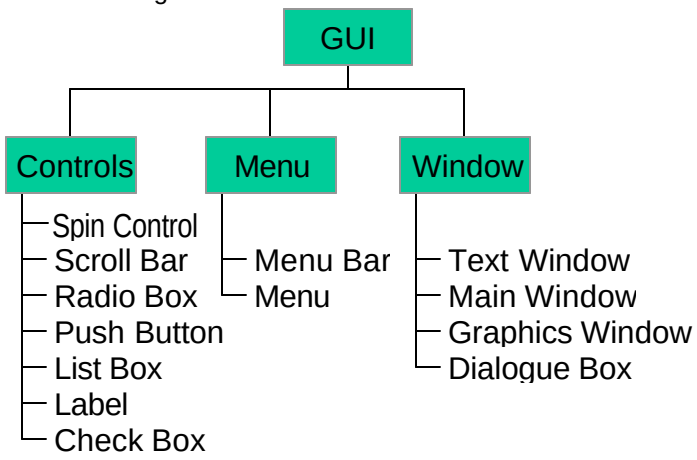
Menus are GUI constructs that can cause the display of new client windows. Menus are a special type of control. Using a menu typically causes a major state change in the GUI System Under Test. Menus are typically a horizontal list of items with subordinate associated pop-up menus. These subordinate menus appear when the user selects an item from the initial menu's list. Menus appear within the context of a window. Testers should note that, on occasion menus may have the appearance of controls (discussed below).

Controls

Controls are GUI constructs that allow the user to transmit commands or information to the GUI System Under Test. Controls appear within the context of a window. Sometimes they are displayed as “real-world” mechanical devices (Pushbuttons, Dials, etc.) to provide the user with operational context. Actuating a control typically causes an event to occur within the System Under Test which may change its state.

GUI Components

There are 13 primary GUI components used in conventional GUI design. The ***Figure 2*** shows the relationship of these components to the basic GUI categories.



- **Figure 2, Relationship of GUI Components to Categories**

The components (listed in alphabetical order) are as follows:

Check Box	A row or column containing one or more toggle buttons, any of which may be in either of two states. A Check Box is a type of Control.

Dialogue Box	A GUI component popped up by the System Under Test to inform or solicit information from the application's user. A Dialogue Box is a type of Window.
Graphics Window	A client window, multiple pixels in length, width, and color depth containing information in a graphical format (bitmap or otherwise). A Graphics Box is a type of Window.
Label	Text or graphic icon in a small window changed in response to an event within the System Under Test. A Label is a type of Control.
List Box	A text list of limited choices allowing a selection based on a controlling policy. A List Box is a type of Control.
Main Window	The application's primary window. The Main Window is a type of Window.
Menu	Either a popup or pull down listing of application processing selections. A Menu is a type of Menu.
Menu Bar	A row typically found across the top of an application's **Main Window** supporting pull down menus. A Menu Bar is a type of Menu.
Push Button	A GUI component labeled with text or graphics generating an event in the System Under Test when actuated. A Push Button is a type of Control.
Radio Box	A row or column containing multiple toggle buttons, allowing only one button selection at a given time. A Radio Button is a type of Control.
Scroll Bar	A GUI component used to horizontally or vertically reposition text or graphics in client windows for viewing. A Scroll Bar is a type of Control.
Spin Control	A numeric list of limited choices allowing selection based on a controlling policy. A Spin Control is a type of Control.
Text Window	A client window of one or more rows and columns containing information in text format supported by an editor allowing the text to be inserted, modified and deleted according to a policy. A Text Window is a type of Window.

Stimulus/Response Pairs

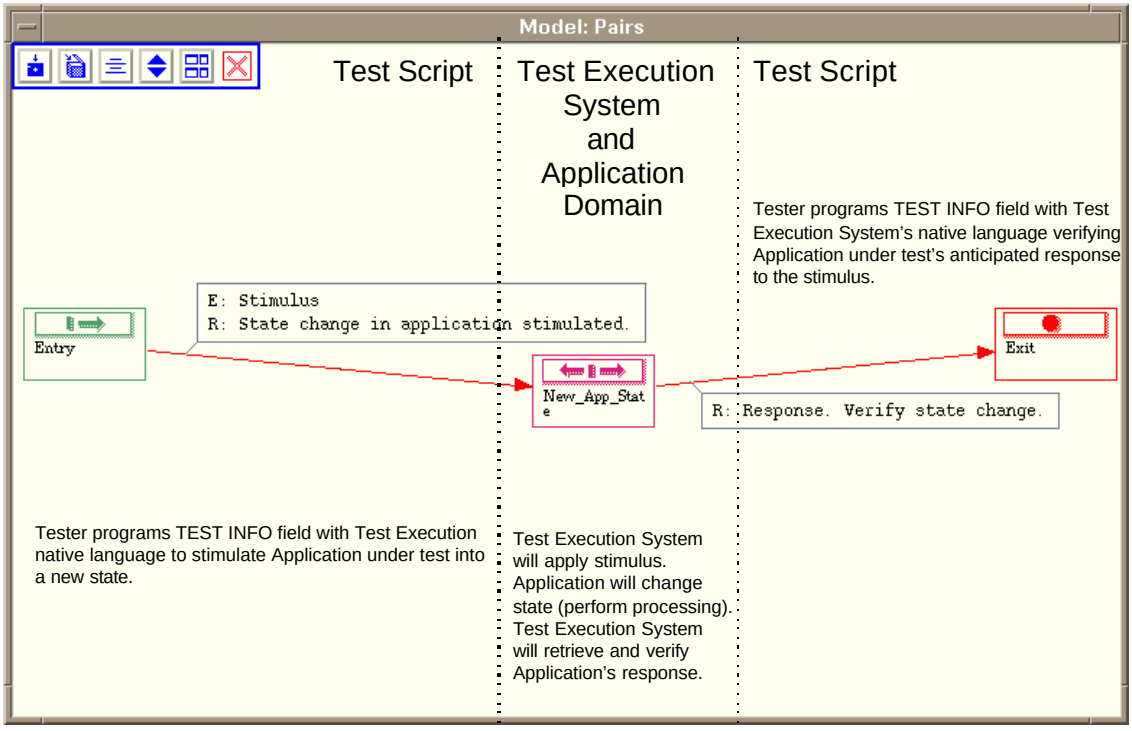
In ***Figure 3***, it is important to understand the concept of stimulus/response pairs. Stimulus/Response pairs are used to describe the construction of state machine-based behavioral models throughout this paper.

When testing, it is desirable that stimulus applied to a system has a verifiable response. When you create models of GUI's you need to know the events needed to stimulate the GUI System Under Test to change its state and the corresponding characteristics of the GUI's new state. The characteristics of the new state are used to verify the response. The stimulus, and response verification must be programmable in the test scripting language of your automated Test Execution System.

The models follow the general state machine format of Inputs, Processing, Outputs. Where an input is a stimulus, Processing is shown as a (presumably new) state, and Outputs are a response.

In our modeling approach we model a stimulus as an Event. Events are shown in models as transition objects (arrows in ***Figure 3***). The modeling tool allows Testers to embed scripting code into model transition objects. Responses are also modeled as transitions. Generally, there is a one-to-one correspondence between stimulus transitions and their corresponding verifiable response. There can also be a many-to-one correspondence between stimulus transitions and a response transition. This occurs when multiple stimuli have the same response. Your testing will be seriously complicated (or your System Under Test has serious design problems), if a single stimulus can result in any of several responses.

Figure 3 describes the general structure of the examples in this paper. It shows a transition dedicated to stimulating the System Under Test, the System Under Test's change of state (a state object), and a transition dedicated to verifying the response of the System Under Test. Each test step in your model needs to include these three elements.



• **Figure 3, Stimulus/Response Pairs**

Window Component Models

This section presents examples of state machine-based, behavioral models for selected GUI components within each of the major categories. Example models were created using Teradyne's TestMaster ™ automated test generation tool. Certain details of the models have been omitted to simplify the examples.

Windows constructs consist of the following basic components:

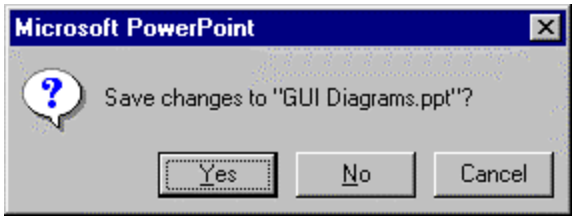
- Dialogue Box
- Graphics Window
- Main Window
- Text Window

A (new) window is generally the response of the underlying system to an event stimulus from the operator. The appearance of a window verifies a response to the stimulus. We suggest that a user models a window as a state or submodel object.

A submodel is subordinate model following the policy of its predecessor, but having its own behavior.

Dialogue Box

A Dialogue Box is "popped-up" into the current window by the underlying System Under Test to inform or solicit information from the application's user. The example shown in ***Figure 4***, is a typical Dialogue Box. This particular example is a Dialogue Box with three Push Button controls.



• **Figure 4, Dialogue Box (Wintel Example)**[2]

Graphics Window

A Graphics Window is a rectangular region of multiple pixels in length, width, and color depth containing information in graphical format (bitmap or otherwise).

Main Window

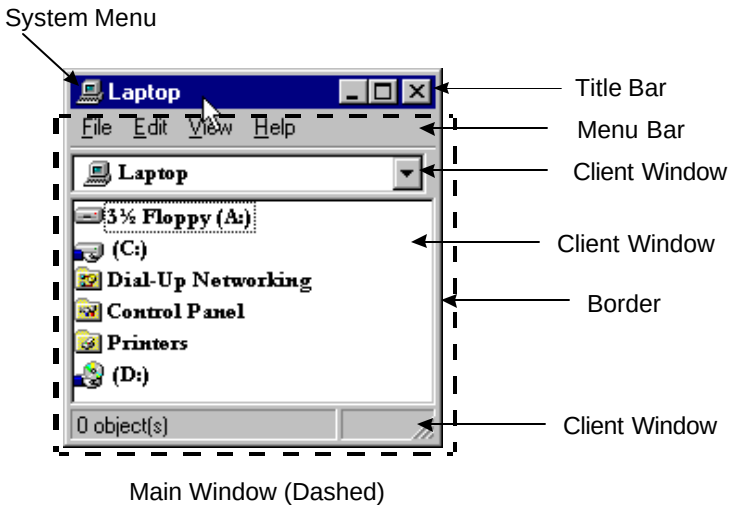
The Main Window is the application's primary window. Typically, the Main Window consists of four parts: Title Bar, Menu Bar, Client Windows and Border. Figure 1 illustrates a Main Window. The Title Bar contains the window's caption and optionally a System Menu or Push Buttons connected to the System Menu. The Title Bar and border are not part of the application. They are typically hosted by the underlying operating system's window management software. The Menu Bar is a row of selections leading to Pull down Menus accessing the System Under Test's major functions. The user and the System Under Test exchange information and commands through the Main Window and one or more Client Windows. Occasionally, the Main Window consists of only a single Client Window. The Border delineates the edges of the Main Window and optionally provides window sizing control.

The Main Window may be movable or static. Users may reposition a movable Main Window within the display device. Typically repositioning the window follows a resizing operation using the System Menu or the Border.

[2] Microsoft PowerPoint is a registered trademark of Microsoft Corporation

Likewise, a user's Main Window may be static and cannot be repositioned. A "full screen" Main Window is the most common example of the static type.

Figure 5 shows a Main Window that includes a Main Window with a Title Bar, Menu Bar, and three client windows.

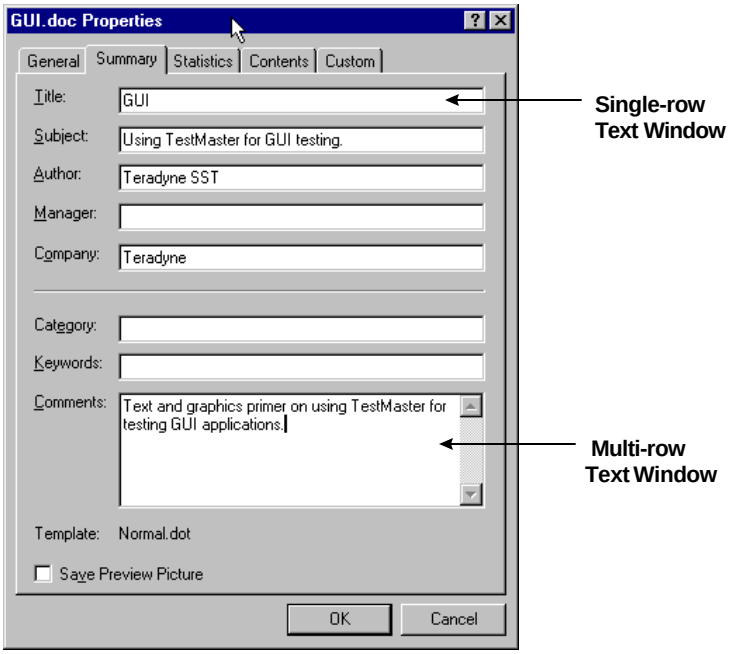


- **Figure 5, Main Window (Wintel Example)**

Text Window

A Text Window is one or more rows and columns containing information in text format supported by an editor allowing the text to be inserted, modified and deleted. A Text Window may have a policy applied by the GUI to control the type and syntax of the text being entered. The policy may include complex behavior.

Figure 6 shows a typical Text Window. Sometimes Text Windows are called Forms. This particular example includes a Text Window with: Title Bar, Tab Menu, two Push Button controls, a Check Box control, eight text windows, and a Slide Bar control.

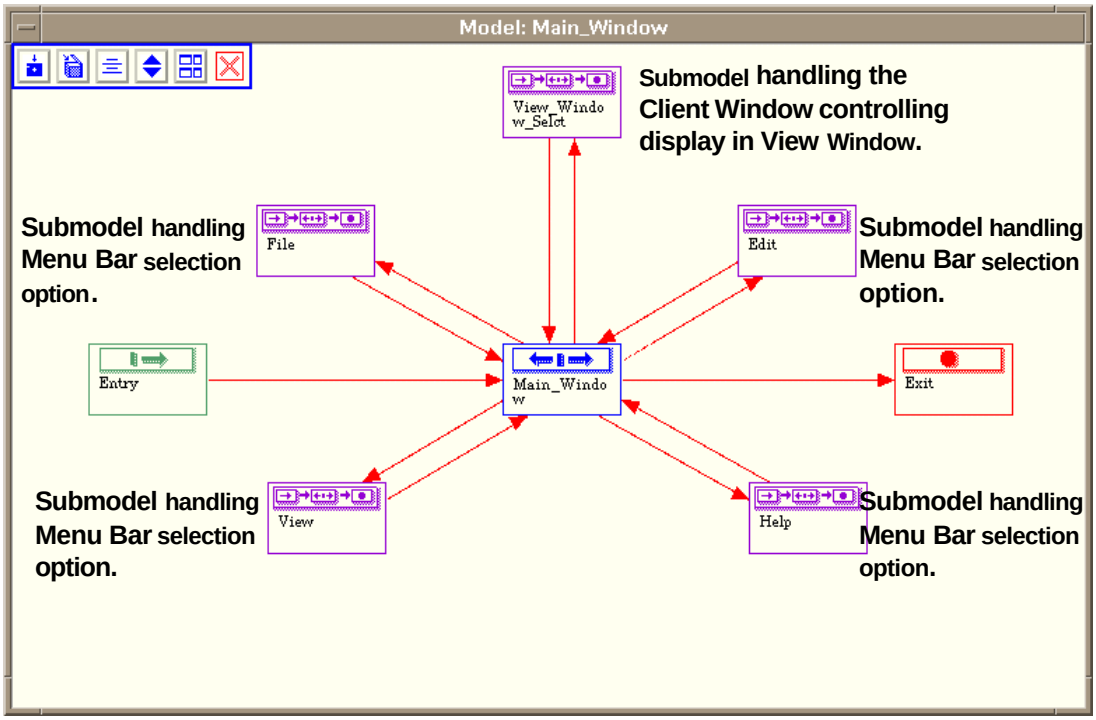


- **Figure 6, Text Windows (Wintel Example)**

Window Example Model

A window represents the processing mode of the System Under Test being presented to the application's user by the GUI. It can also be a response to a Menu selection or a Control actuation. *Model Windows as individual states or submodels*.

Model a window as a submodel, when it has complex behavior that needs to decomposed into several instances of simpler behavior. ***Figure 7*** gives an example of how to model a Window. The example is the Main Window example shown in ***Figure 5***, Main Window.



- **Figure 7, Main Window Submodel (Wintel Example)**

Figure 7 gives an example of how to model the Main Window shown in Figure 5, Main Window. The Main Window is a processing state of the System Under Test. The Main Window is rendered as a single state object in the model (central box icon labeled Main_Window). The Menu Bar selection options available from the Main Window are shown as submodels surrounding the central state. Another submodel (View_Window_Select) handles the control of the Main Window's client View Window.

Note the most complex behavior in the Main Window model results from the Menu Bar. This reflects the Main Window's essentially static behavior. Its true function is to provide a starting point for the underlying application's processing modes available through the Menu Bar.

Menu Component Models

Menu components consist of the following basic types:

- Menu
- Menu Bar

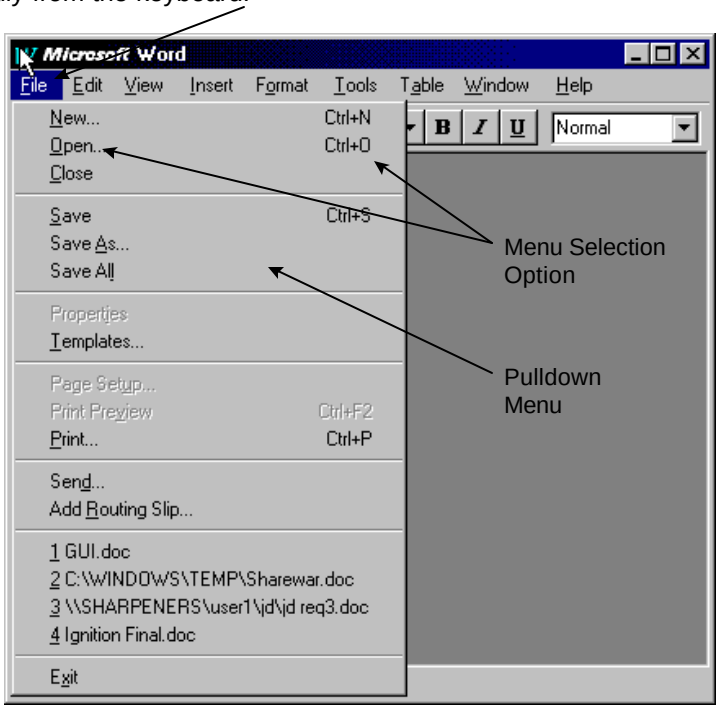
Menu selection options populate menus. Typically the selections are textual, although they may also be graphical. Users exercise menu selection options with a mouse button click on the menu selection option or from their keyboard. Typically, menu selections result in an event stimulating the System Under Test into a new processing state. Usually, a new state results in a new window being displayed.

Menu

A Menu is a fixed vertical or horizontal text listing of application processing selections. Typically, only valid processing options appear for user selection. It is assumed that any menu selection is as likely as any other menu selection in the GUI's behavior.

Menus can either be popup or pull down. Popup Menus appear in the display as a result of a user generated event. Typically, they can appear as the result of a key action or mouse button click. A Pull down Menu requires placing the mouse cursor onto a Pull down menu selection and clicking the mouse button or typing a controlling key to popup the menu.

Figure 8 shows an example of a pull down-type menu (File Menu) generated by a popular word processor. Users access the Menu from the Main Window's Menu Bar. This particular example groups the menu selections according to type (File Saving, File Printing, etc.). Users address the Menu with either a mouse button click or optionally from the keyboard.



• Figure 8, Pull down Menu (Wintel Example)[3]

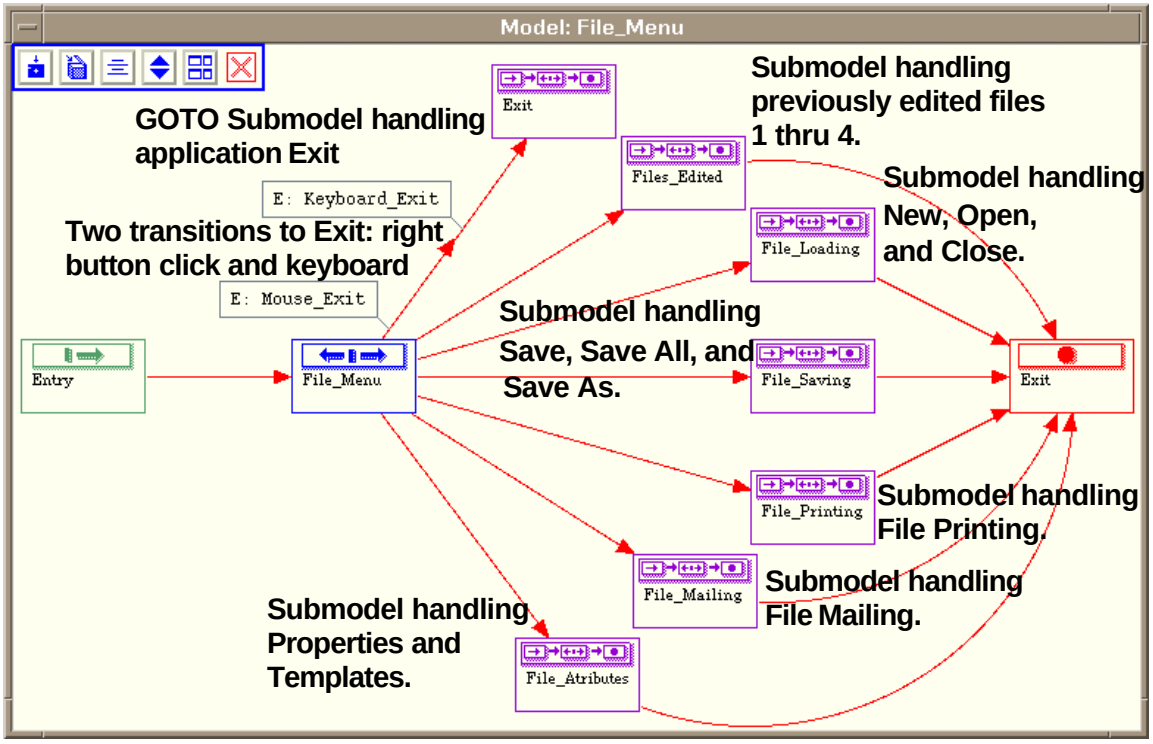
[3] Microsoft Word is a registered trademark of Microsoft Corporation.

Tab Windows are also Menu-type constructs. The Tabs appear in a file folder fashion displaying an underlying information window.

Modeling Menus

Model Menu Bars as diamonds. A Diamond models a Menu Bar, with each menu selection appearing as an object between the Entry and Exit objects.

Figure 9 gives an example of how to model the Pull down File Menu shown in ***Figure 8***. Submodels group the related functions together to reduce the complexity of the Diamond. This grouping reduces the Diamond from a potential 18 model objects to seven submodels. The grouping reflects the implied grouping shown in the menu figure.



• Figure 9, File Menu Example Submodel

The grouping of menu options comprises all the options except for Exit. Exit was explicitly omitted because it terminates the application and ends testing. The model does not include “hot-keys” (Ctrl+N, Ctrl+O, Ctrl+S, Ctrl+P). Within the example, a user can type a hot-key at any time. Omitting them simplifies the example.

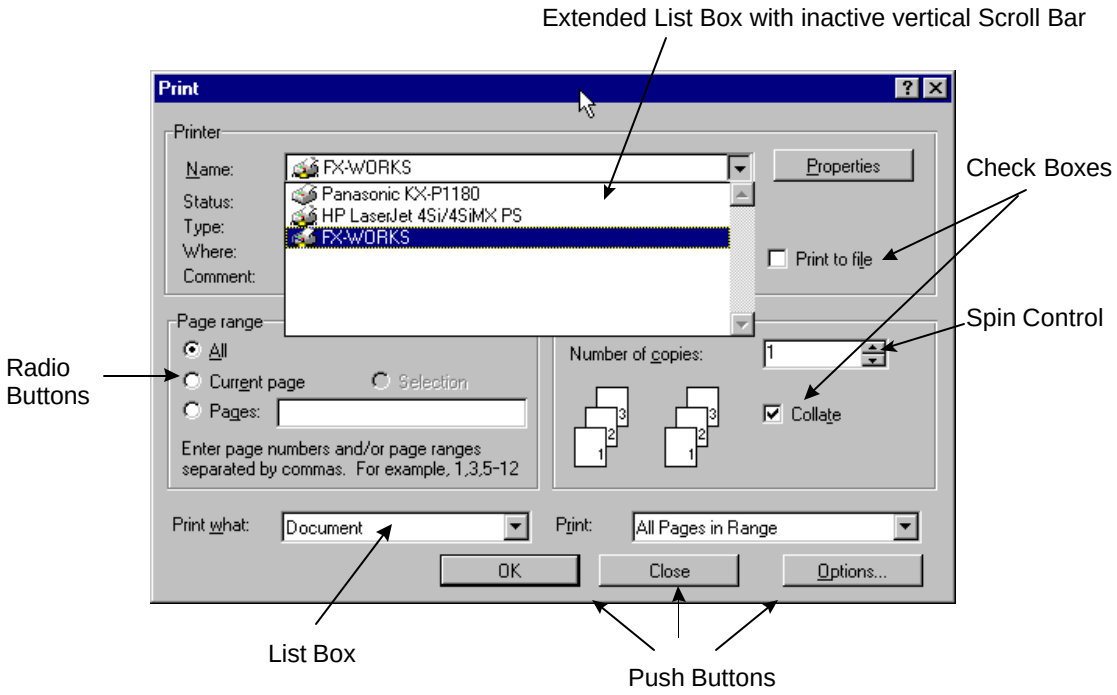
The stimulus to the menu shown above (except for Exit) is from the transitions shown entering the central submodel objects. The stimulus will be either a mouse button click or typed keyboard characters. The System Under Test’s typical response to a menu selection is immediately verifiable as a major state change in the application. The popup of a new window or dialogue box associated with the new state typically verifies the System Under Test’s response. This verification would appear in the first transition of the submodels shown in the example.

Control Constructs

Control constructs consist of the following basic components:

- Check Box
- Label
- List Box
- Push Button
- Radio Box
- Scroll Bar
- Spin Control

Control actuation stimulates the System Under Test into a new processing state. Control actions (press a button, check a box, select a list option) appear in models as transitions. Careful inspection of the control's operation is needed before modeling. It is not uncommon for GUI controls to have the appearance of a control and the functionality of a menu. In addition, control constructs may have the appearance of one type of construct, but function like another type of control construct. For example, Push Buttons may trigger popup menus and Spin Controls may function like List Boxes. The example window shown in ***Figure 10*** includes all the control constructs with the exception of a Label.



• Figure 10, GUI Controls (Wintel Example)

Check Box

A Check Box can be one or more toggle buttons arranged in a row or column, any of which may be in either of two states (checked or unchecked). The states of related Check Boxes are not mutually exclusive. The state of a Check Box is persistent. Once checked, the box remains checked until unchecked.

Modeling Check Boxes

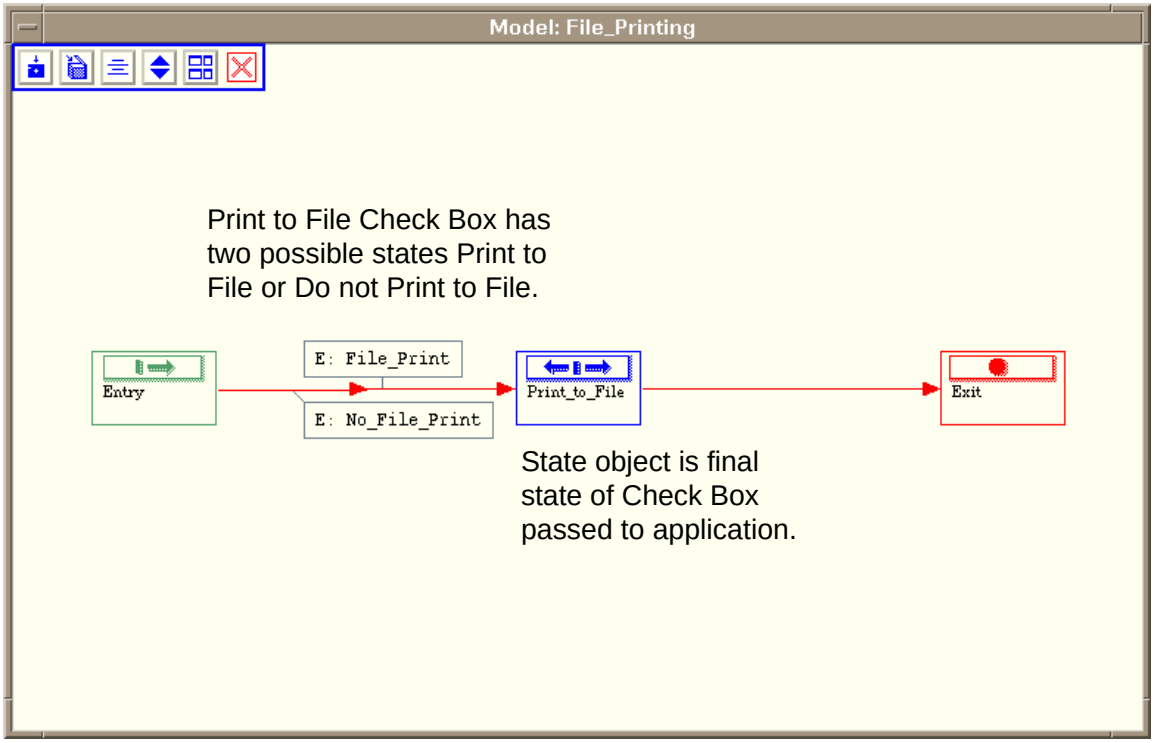
Model Check Boxes as two transitions with the same source terminating in the same destination state. Each transition is a possible state of the Check Box. The destination state represents the final setting of the Check Box passed to the System Under Test. Multiple Check Boxes are modeled as a chain using the same basic construct. Check Boxes generate events changing the state of the System Under Test.

Table 1 shows the possible states of a check box taken from the Printer section in ***Figure 10***, GUI Controls. The events Print to File and No Print to File are the user checking the Check Box using a mouse click.

Table 1, Print to File State Table

	Event	
Current State	Print to File	No Print to File
Print to File	No Change	No Print to File
No Print to File	Print to File	No Change

Figure 11 gives an example of how to model a Check Box. The information is taken from the ***Table 1***.



• **Figure 11, Check Box Example submodel**

The example models a single checkbox in the Print Window Example. The two transitions to the left of the Print_to_File state are the events stimulating the System Under Test. These events are the two possible states of the Check Box. The example model does not show the verification of the underlying system's response. The response can either be immediately verifiable or contingent on another event, for example the Push Button "OK." Program an immediately verifiable response into the transition to the right of the model's state. If the response were not immediately verifiable, create a variable to preserve the context of this model. In the variable, save the Check Box state (No_File_Print or File_Print) to verify the response in another submodel.

Push Button

A Push Button is a component labeled with text or graphics generating an event in the System Under Test when actuated. Push Buttons can be "pushed" with a mouse button click, or optionally actuated through the keyboard. The state of a Push Button is not persistent. A pushed button immediately returns to its default state. A pushed Push Button immediately stimulates the System Under Test with an event. Push Buttons generate events changing the state of the System Under Test. They have only two states: pushed or not pushed.

Modeling Push Buttons

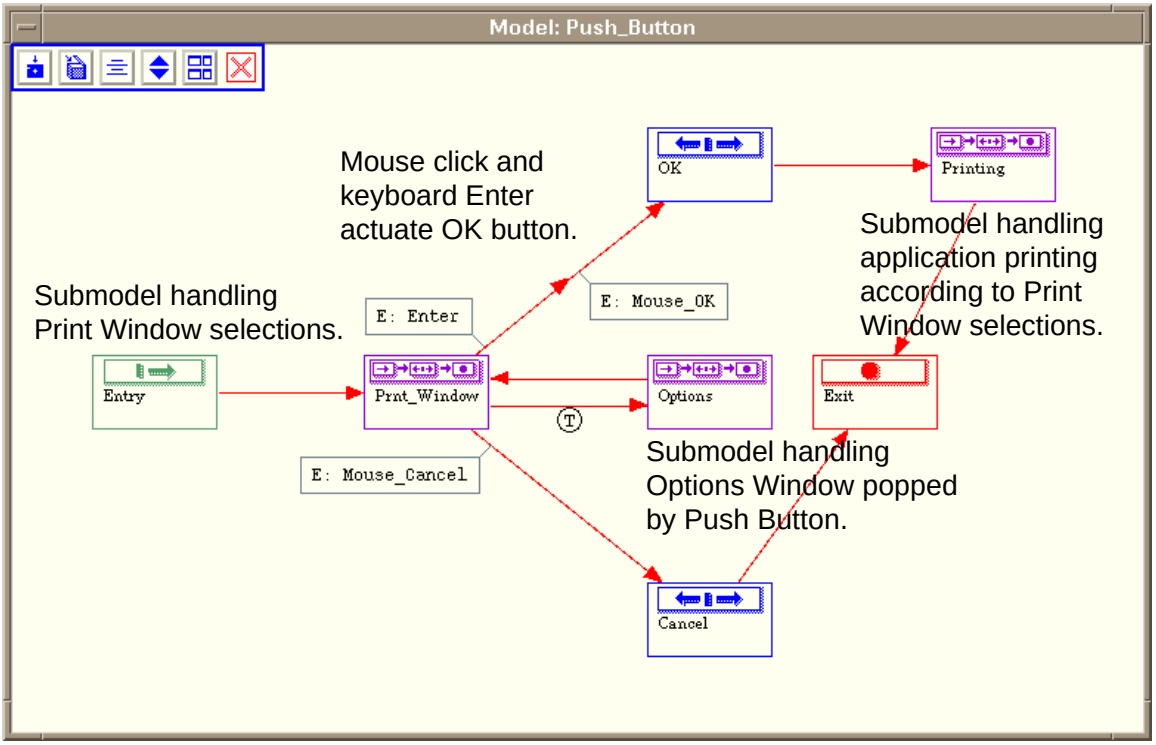
Model Push Buttons as a single transition terminating in a state. The transition is the actuation of the Push Button. The destination state represents the setting of the Push Button passed to the System Under Test. The transition leading out of the state contains the code to verify the Push Button's setting.

Table 2 shows the possible states of the three Push Buttons in the Print Window example shown in *Figure 10*, GUI Controls. The events OK, Cancel, and Options are the user actuating a Push Button using a mouse click.

Table 2, Print Window Push Buttons State Table

	Event		
Current State	OK	Cancel	Options
OK	OK	N/A	N/A
Cancel	N/A	Cancel	N/A
Options	N/A	N/A	Options

Figure 12 gives an example of how to model a Push Button. The example is from the table above created from ***Figure 10***.



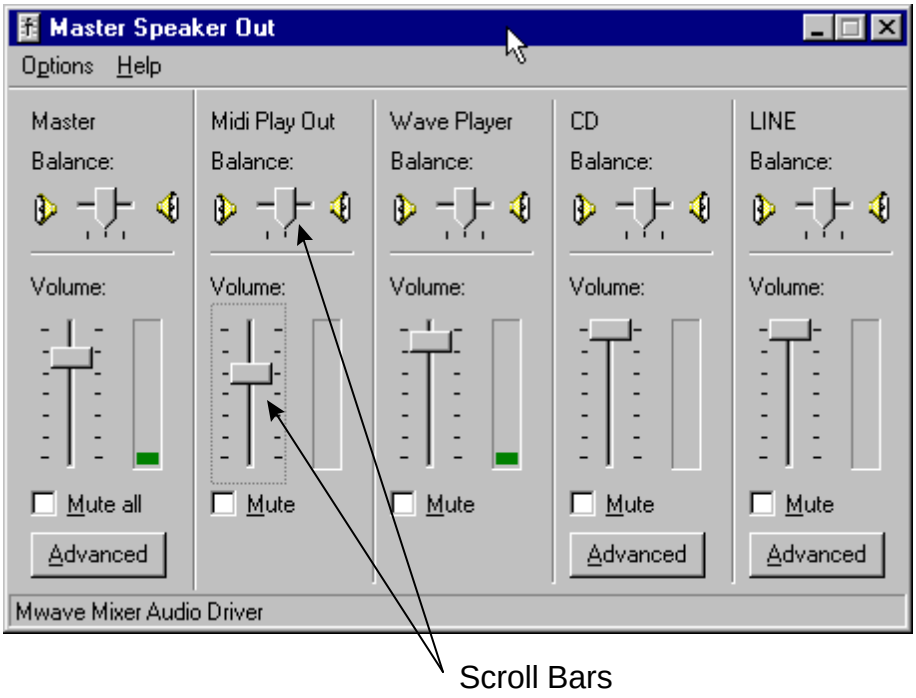
• **Figure 12, Push Button Example Submodel**

The model shown in ***Figure 12*** hides the complexity of selecting print options from within the Print Window in the Prnt_Window submodel. The transitions leading from the Print_Window submodel to the Options submodel and the OK and Cancel states are the events stimulating the System Under Test. These events occur when the GUI's user clicks on the Print Window's Push Buttons. Either mouse clicking on the Push Button or typing the keyboard Enter key actuates the OK Push Button. Two transitions entering the OK state model this Push Button. The example ignores the actuation of Tab and Arrow keys. Actuating OK exports print options selected in the Prnt_Window submodel into the Printing submodel. The Cancel Push Button is more straightforward. A single transition represents a user's mouse click on this GUI's Push Button. Cancel does not export Print options. Mouse clicking on the Options Push Button pops a window that returns to the Print Window. The Options Window is a Tabbed window. Tabbed windows are a type of Menu. This tabbed menu selects further print options. The Options submodel encompasses this complex behavior. The Print_Window submodel could logically include the Options submodel. The model does not show the underlying system's response. The response to the Cancel Push Button is the immediate collapse of the Print Window and no print job is created. Program an immediate verifying response into the transition to the right of the Cancel state object. The transition to the right of the OK state object is programmed to verify the immediate collapse of the Print Window only. The Printing submodel hides the complexity of verifying the correct print selections being used by the application when OK is chosen. Assume the response to the Options Push Button is in the Options submodel.

Scroll Bar

A Scroll Bar is a sliding bar arranged horizontally or vertically to increment a value, or reposition text and graphics in client windows for viewing. A Scroll bar allows selection based on a controlling policy. Numeric Scroll Bars typically control the selection of real (decimal-type) values. Graphic or Text Window Scroll Bars control panning within a window. A Scroll Bar can be in one and only one state. The state of a Scroll Bar is persistent during the time a window is being displayed, and may be longer. A selected value remains selected until it is deselected. Scroll Bars transmit data to the System Under Test.

Figure 13 gives an example of both horizontal and vertical scroll bars. In addition, it contains Check Boxes, Labels, and Push Buttons.



• Figure 13, Scroll Bar (Wintel Example)[4]

Modeling Scroll Bars

Model Scroll Bars as multiple transitions entering a state. Each of the transitions entering the state is a Scroll Bar increment. The state represents the final setting of the Scroll Bar presented to the System Under Test. The transition leading out of the state contains the code to verify the Scroll Bar's setting.

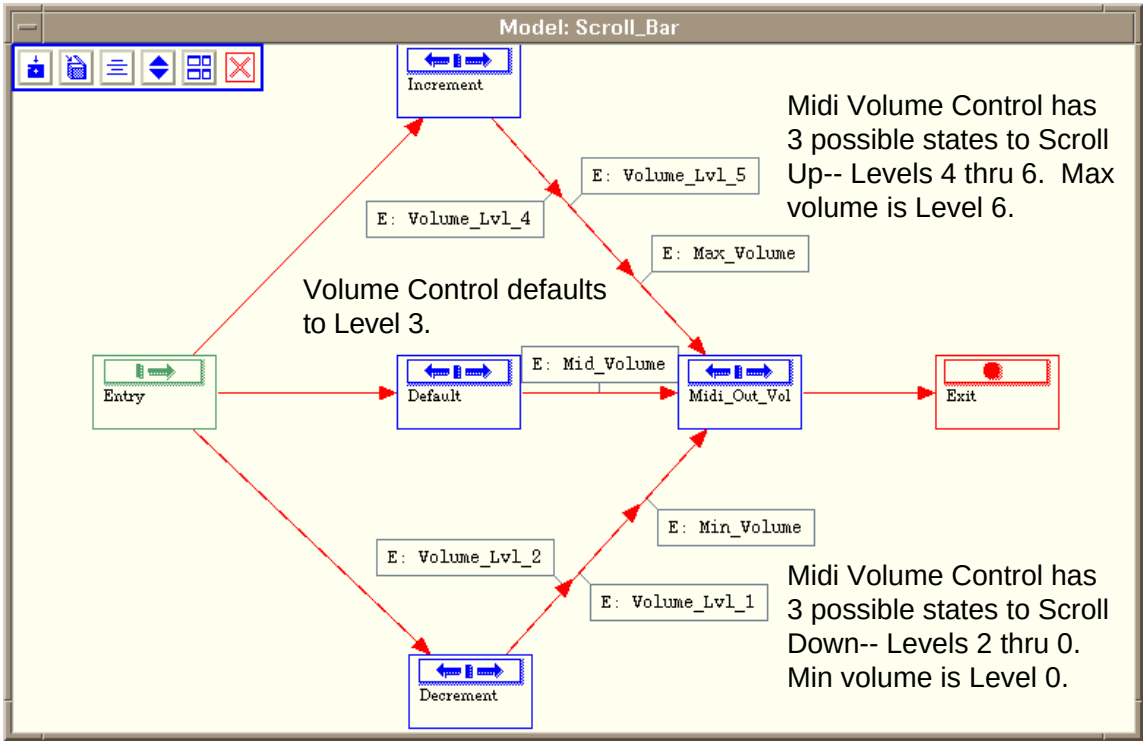
Table 3 shows the possible states of a vertical, Volume Scroll Bar taken from the Midi Play Out section of ***Figure 13***, Scroll Bar. There is no unit scale for the Scroll Bar. Therefor all Scroll Bar positions are relative. This particular Scroll Bar's minimum is zero (bottom position). Its maximum is six (top position). The Scroll Bar's default position is in the center or at three. The events are the user selection of decrement or increment the volume by mouse clicking on the bar and dragging it up or down.

Table 3, Midi Play Out Scroll Bar State Table

Event		Comment
Increment	Decrement	
Max.	N/A	Maximum Scroll Bar value.
N/A	N/A	Default Scroll Bar value.
Default thru Max.-1	Default thru Min.+1	Intermediate Scroll Bar values.
N/A	Min.	Minimum list value.

[4] Mwave is a registered trademark of General Signal Corporation.

Figure 14 gives an example of how to model a Scroll Bar. The example uses the information from ***Table 3***.



• Figure 14, Scroll Bar Example Submodel

This particular Scroll Bar segments its range into discrete increments. In the model above the states in the model are the possible test states of the Scroll Bar. Because the possible range of the Scroll Bar totals only seven increments, all volume levels appear as individual transitions. The model does not show the verification of the Scroll Bar's state. Code an immediate verification of the Scroll Bar's state into the transition to the right of the Midi_Out_Vol state object. Otherwise, create a variable to preserve and export the volume level for use in verifying the System Under Test's response in this or another submodel. For example, tests coming from the Decrement state's Min_Vol event might set a variable to 0 to indicate no volume. The verification in the transition to the right of the Midi_Out_Vol state would verify the volume produced compared to the volume indicated by the variable.

GUI Modeling Process

The following is the recommended process for modeling a GUI and producing a test script to test your application.

1. Set test objectives.
2. Analyze the windowing system.
3. Determine usage patterns.
4. Model windowing system's infrastructure.
5. Test infrastructure model.
6. Model windowing system's secondary functions and details.
7. Test model of primary and secondary functions.
8. Execute test script on test execution system.
9. Test GUI.

1-Set Test Objectives

Set goals for testing the GUI. It is necessary to understand how much testing of the system you can perform, how long it can take, and when testing is complete. Test objectives can be as simple as:

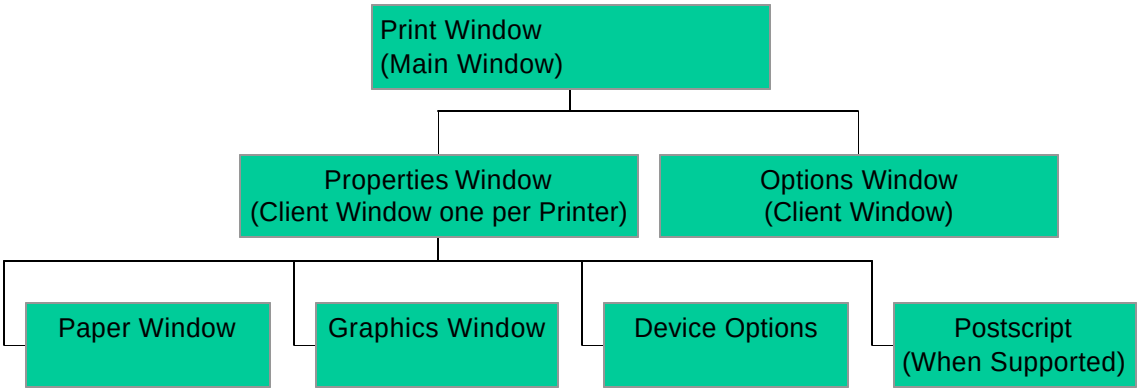
- Number of errors found, i.e. none remaining.
- Number of tests executed, i.e. 500 tests exercising all functions.
- Code coverage as measured by a software coverage tool, i.e. tests result in 90% coverage reported by tool.
- Number of hours spent testing, i.e. three calendar months of testing performed.

The product of this analysis is a schedule and criteria for completing the testing of the GUI system.

2-Analyze the Windowing System

Study the windowing system. It is necessary to understand: the hierarchy of the windows within the system, how the controls operate, how commands pass to the System Under Test, and how data passes back and forth between the GUI and the System Under Test. List the GUI components associated with each window.

Figure 15 gives a partial example of the windowing hierarchy of the Print Window used in previous figures.



• **Figure 15, Print Window Windowing Hierarchy**

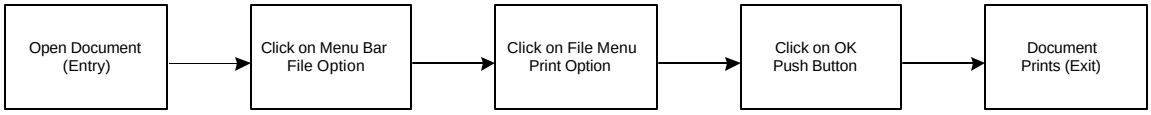
The product of this analysis is a list of the major GUI functions and their relationship to each other. This analysis will give you a list of your major submodels.

3-Determine Usage Patterns

Study the GUI's usage. It is necessary to understand the standard procedures for operating the GUI. To be successful in your testing, you will need to perform this study from three perspectivess:

- How is the system designed to be operated?
- How is the system being operated or can be operated (this may not be the same as the design usage)?
- How can the system be abused?

Figure 16 gives an example of a common usage sequence for the Print Window used in previous examples.



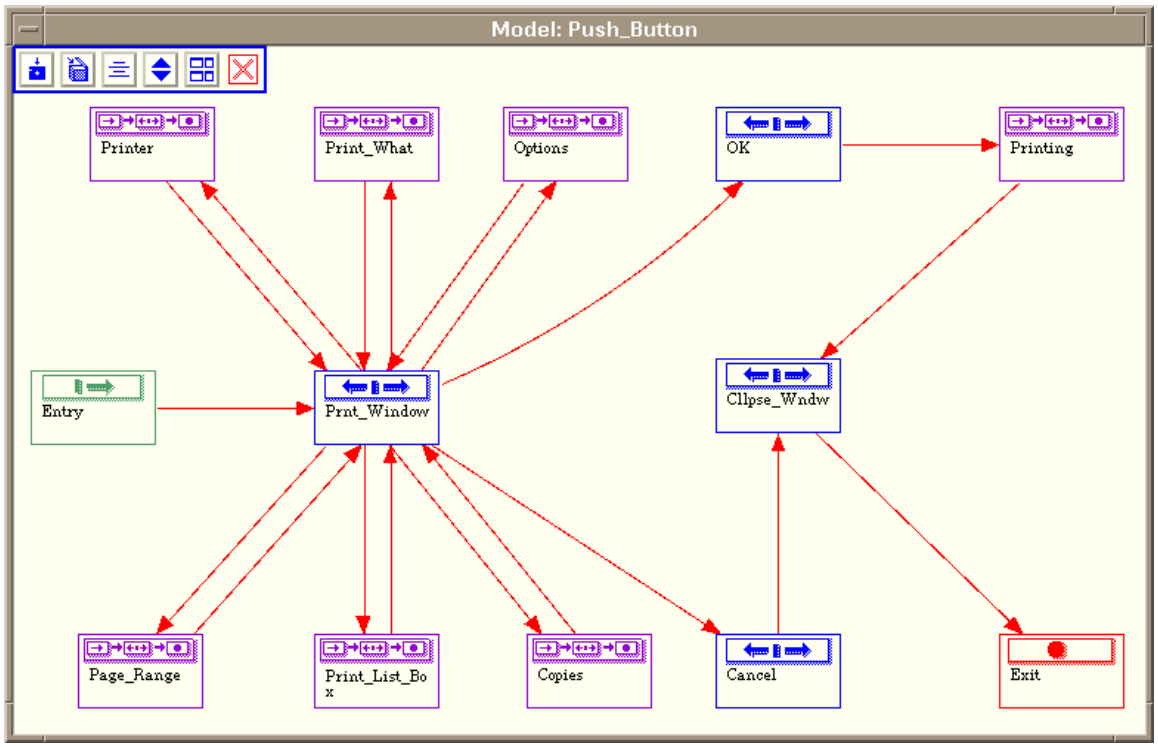
• **Figure 16, Print Window Usage Scenario**

The product of this analysis is a number of usage scenarios exercising each of the GUI's functions. This analysis will be used as a reference for validating your model.

4-Model Windowing System's Infrastructure

Create a preliminary TestMaster model based on the analysis of the windowing system. Infrastructure includes only enough model detail to reach the lowest-level window. Model the Main Window with its associated menus and controls as submodels. Iterate your modeling, extending the model submodel-by-submodel. Only model the GUI Control components needed to reach the next window. When modeling control components only model the default or most commonly used option. Use the component templates presented in this document to help you get a general idea on modeling windows, menus, and controls. *Do not attempt to model all the details shown in the templates at this step*. Every time you finish a submodel it is a good idea to execute it in a standalone context to check it's behavior. Once checked you can enter Test Execution System code or comments as you model.

Figure 17 gives an example of how a preliminary model of the Print Window used in previous examples might appear.



• **Figure 17, Preliminary Model**

The product of this task is a model including all the systems windows and the GUI's major functions.

5-Test Infrastructure Model

Execute the model verifying it behaves (produces tests) based on the analysis of the common usage patterns. Begin testing the model using a simple coverage scheme. A large amount of the model's behavior is structural—a result of placing the model objects in relationship to each other correctly. You may need to add additional information about the context or uses to ensure the model completely mimics the behavior of the System Under Test's specification. Iterate running tests and improving the fidelity of the model until it produces tests matching your usage profiles.

Test Execution System code should be complete in the Test Information fields of the model at this point. The product of this task is a test script testing all the default or most commonly used GUI functions. Depending on the test objectives, test scripting may be complete at this point.

About Coverage Schemes

One of the most powerful capabilities of a model based approach is the ability to create a test suite appropriate for each phase of a product's life-cycle by just changing the coverage scheme used. For example a Transition coverage produces tests that include every transition in a test at least once. At the other extreme, Full coverage produces tests that include every transition in all possible permutations with every other transition. Transition coverage can produce a modest number of tests that can satisfy most testing requirements. Full coverage generally produces a large amount of tests for even simple models. A higher degree of constraining needs to be applied to reduce the number of tests produced using Full coverage schemes. Other schemes enable a test engineer to create tests that can conform to the operational profile (see Operational Profiles in software reliability engineering from J.D. Musa) of users of the system, QuickCover[tm] provides a test suite similar to a transition coverage but based on user profile. ProfileCover can provide any number of tests whose distribution will mimic that of actual users, these suites can be used to determine expected system reliability or provide a excellent set useful for performance or load testing of a system.

To summarize, the key point is that once the basic model has been constructed all of these test suites can be created whenever they are needed and will reflect the current specification of the system [as represented in the model].

6-Model Windowing System's Secondary Functions and Details

Add the remaining functions and test paths for windowing system to the model. Secondary functions include the details within each window. Iterate your modeling, enhancing the model submodel-by-submodel. Model all GUI Control components for every window. When modeling control components add minimum and maximum Control options to every control. Use the component templates presented in this document to help you model windows, menus, and controls. Submodels should contain at least as much detail as shown in the templates. Every time you finish a submodel run a mini-test of the submodel standalone to verify your work.

Be prepared for the number of tests produced by your model to increase dramatically when using Full or Profile style coverage schemes.

Write Test Execution System code for the new functions into the Test Information fields of the model as completely as possible. The product of this task is a model including all the systems windows, the GUI's major and minor functions, and interesting test cases.

7-Test Model of Primary and Secondary Functions

Execute the model and verify that it produces all the scenarios called out in the analysis of the common usage patterns. The model may produce additional scenarios not anticipated by the analysis. Be prepared to evaluate them for inclusion in the test script. Begin testing using a Transition coverage scheme. If a large amount of tests are being generated disable unrelated or loosely related submodels and review the model's output in a piecemeal fashion.

Repeat this procedure using more robust coverage schemes adding constraints as needed. These constraints will eliminate trivial and uninteresting tests from the script resulting from the new functions. The product of this task is a fully constrained test script testing all the GUI's functions and interesting test cases. Initial test scripting is complete at this point.

8-Execute Test Script on Test Execution System

Execute the test script on the Test Execution System. You must debug the script before you can detect errors in the System Under Test. Verify the script drives the Test Execution System and through it the System Under Test. There are likely to be errors in the Test Execution System code. *Be prepared to fold the corrections to the script back into the model.* Typically, it takes several debug runs to achieve error-free Test Execution System code. The product of this task is a model producing syntactically correct test scripts. Test scripting is complete at this point.

9-Test GUI

Execute the test script on the Test Execution System evaluating the result for errors in the System Under Test.

Summary

This paper provides the basis for a toolkit of submodels and processes that Testers can use as examples for creating GUI models. It also offered advice and a general plan for testing a GUI.

A GUI contains an industry-standard set of components connected in a logical fashion. When modeling a GUI, Testers need to identify the components being used and how they relate to each other. Once they understand the structure of the GUI, they can implement all the components into a model. Testers should use the templates included in this paper as examples of how to create the building blocks of their own models. With a little experience you will quickly come to recognize the common GUI components and how to assemble them into working models.

Finally, Testers need a plan and need to know when testing is complete—before they begin. A step-by-step plan for testing a GUI is included with examples. Testers should review this plan within the context of their own plans and schedules for testing their System Under Test.

Creating a model of a GUI is not difficult, if it is approached in a systematic fashion. This paper has presented the solutions model based testing practitioners regularly use with success to create GUI models.

References

[1] Beizer, B., *Black Box Testing*, New York, John Wiley & Sons, 1995. ISBN 0-471-12094-4.

[2] Apfelbaum, L.,Doyle, J., "Model Based Testing", Proceedings of the Software Quality Week 1997 Conference, 1997.

[3] Musa, J.D., "Operational profiles in software reliability engineering," *IEEE Software*, 10(2), pp 14-32.

[4] Apfelbaum, L., "Automated Functional Test Generation", Proceedings of the Autotestcon '95 Conference, IEEE, 1995.

[5] Savage, P., Walters, S, and Stephenson, M., "Automated Test Methodology for Operational Flight Programs", Proceedings of the 1997 IEEE Aerospace Conference, 1997.

[6] Schroeder, J., "Using TestMaster in an Object Oriented Development Environment", Applications Note #6, Teradyne SST, 1997.