XRunner®

User's Guide

Version 6.0

MERCURY INTERACTIVE

XRunner User's Guide—Version 6.0

If you have any comments or suggestions regarding this document, please send them via e-mail to documentation@mercury.co.il.

Mercury Interactive Corporation
470 Potrero Avenue
Sunnyvale, CA 94086
Tel. (408) 523-9900
Fax. (408) 523-9911

XRUG6.0/01

Table of Contents

PART III: CREATING TESTS

PART IV: PROGRAMMING WITH TSL

PART V: RUNNING TESTS

PART VII: CONFIGURING XRUNNER

PART VIII: WORKING WITH LOADRUNNER

PART IX: APPENDIXES

Welcome to XRunner

Welcome to XRunner, Mercury Interactive's automated GUI testing tool for X Windows. XRunner gives you everything you need to quickly create and execute sophisticated automated tests on your application.

Using This Guide

This guide describes the main concepts behind automated software testing. It provides step-by-step instructions to help you create, debug, and execute tests, and to analyze test results.

This guide contains eight parts:

Part I Starting the Testing Process

Provides an overview of XRunner and the main stages of the testing process.

Part II Understanding the GUI Map

Describes Context Sensitive testing and the importance of the GUI map for creating adaptable and reusable test scripts.

Part III Creating Tests

Describes how to create test scripts and insert checkpoints that allow you to check data in your application.

Part IV Programming with TSL

Describes how to enhance your test scripts using variables, control-flow statements, arrays, user-defined functions, and XRunner's visual programming tools.

Part V Running Tests

Describes how to execute your automated tests and analyze test results.

Part VI Debugging Tests

Describes how you can control test execution in order to identify and isolate bugs in test scripts.

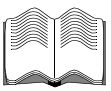
Part VII Configuring XRunner

Describes how to change system defaults in order to adapt XRunner to your testing environment.

Part VIII Working with LoadRunner

Describes how to use XRunner, in combination with LoadRunner to test client/server systems.

XRunner Documentation Set



In addition to this guide, XRunner comes with a complete set of documentation

XRunner Installation Guide explains how to install XRunner on a single computer, or on a network.

XRunner Tutorial teaches you basic XRunner skills and shows you how to start testing your application.

TSL Reference Guide describes Test Script Language (TSL) and the functions it contains, and examples of how to use these functions.

XRunner Customization Guide explains how to customize XRunner to meet the special testing requirements of your application.

Product Release Notes provides last-minute information that is not included in the XRunner User's Guide.

Online Resources

XRunner Online Help provides quick answers to questions that arise as you work with XRunner. It describes menu commands and forms, and TSL functions and guides you in tailoring XRunner's Context Sensitive testing features to meet the specific needs of your application.

TSL Online Help describes Test Script Language (TSL), the functions it contains, and examples of how to use the functions.

Typographical Conventions

This book uses the following typographical conventions:

Bold	**Bold** text indicates function names and the elements of the functions that are to be typed in literally.
Italics	*Italic* text indicates variable names.
Helvetica	The Helvetica font is used for examples and statements that are to be typed in literally.
[ ]	Square brackets enclose optional parameters.
{ }	Curly brackets indicate that one of the enclosed values must be assigned to the current parameter.
...	In a line of syntax, three dots indicate that more items of the same format may be included. In a program example, three dots are used to indicate lines of a program that have been intentionally omitted.
|	A vertical bar indicates that either of the two options separated by the bar should be selected.

Part I

Starting the Testing Process

1

Introduction

Welcome to XRunner, Mercury Interactive's automated testing tool for X Windows applications. This guide provides you with detailed descriptions of XRunner's features and automated testing procedures.

Recent advancements in client/server software tools enable developers to build applications quickly and with increased functionality. Quality Assurance departments must cope with software that is dramatically improved, but increasingly complex to test. Each code change, enhancement, bug fix, and platform port necessitates retesting the entire application to ensure a quality release. Manual testing can no longer keep pace in this rapid development environment.

XRunner helps you to automate the testing process, from test development to execution. You create adaptable and reusable test scripts which challenge the functionality of your application. Prior to a software release, you can execute these tests in a single overnight run—enabling you to detect bugs and ensure superior software quality.

XRunner Testing Modes

XRunner facilitates easy test creation by recording how you work on your application. As you point and click on GUI (Graphical User Interface) objects in your application, XRunner generates a test script in the C-like Test Script Language (TSL). You can further enhance your test scripts with manual programming. XRunner includes a Visual Programming tool which helps you to quickly and easily add functions to your recorded tests.

XRunner offers two modes for recording tests:

Context Sensitive

Context Sensitive mode records your actions on the application under test in terms of the GUI objects you select (such as windows, lists, and buttons), ignoring the physical location of an object on the screen. Each time you perform an operation on the application, a TSL statement is generated in the test script that describes the object selected and the action performed.

As you record, XRunner writes a unique description of each selected object to a GUI map. The GUI map consists of files that are maintained separately from your test scripts. If the user interface of your application changes, you only have to update the GUI map, and not hundreds of tests. This allows you to easily reuse your Context Sensitive test scripts on future versions of your application.

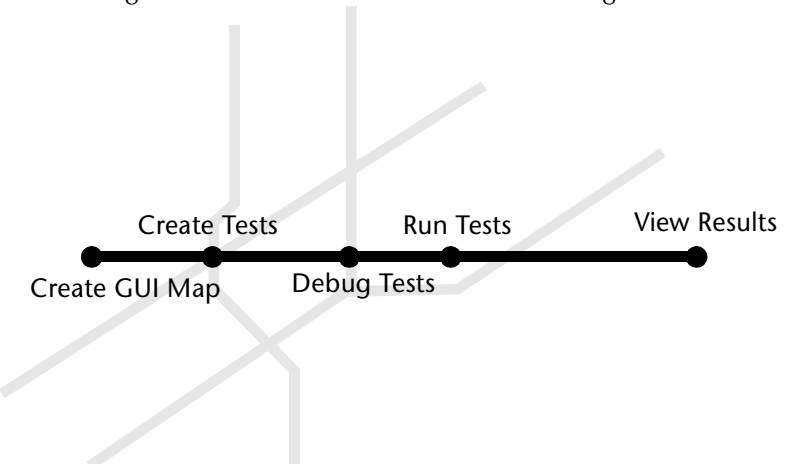
To run a test, you simply play back the test script. XRunner simulates a human user by moving the mouse cursor over your application, selecting objects, and entering keyboard input. XRunner reads the object descriptions in the GUI map and then searches for objects that match the description in the application under test (AUT). It can locate the objects within a window even if their placement has changed.

Analog

Analog mode records mouse clicks, keyboard input, and the exact coordinates traveled by the mouse. When the test is executed, XRunner retraces the mouse tracks. Use Analog mode when exact mouse coordinates are important to your test, such as when testing a drawing application.

The XRunner Testing Process

Testing with XRunner involves five main stages:



Create the GUI Map

The first stage is to create the GUI map so that XRunner can recognize the GUI objects in your application. Use the Test Wizard to review the user interface of your application and systematically add descriptions of every GUI object to the GUI map. Alternatively, you can add descriptions of individual objects to the GUI map by clicking on objects while recording a test.

Create Tests

Next, you create test scripts through recording, programming, or a combination of both. While recording tests, insert checkpoints where you want to check the response of the application under test (AUT). You can insert checkpoints that check GUI objects and bitmaps. During this process, XRunner captures data and saves it as *expected results*—the expected AUT response to the test.

Debug Tests

Run tests in Debug mode to make sure that the test runs smoothly. You can set breakpoints, monitor variables, and control test execution in order to

identify and isolate bugs. Test results are saved in the debug directory, which you can discard when you finish debugging the test.

Run Tests

Run tests in Verify mode. Each time XRunner encounters a checkpoint in the test script, it compares current data in the AUT to the expected data captured earlier. If any mismatches are found, it captures them as *actual results*.

Analyze Results

Determine the success or failure of the tests. Following each test run, results are displayed in a report. The report details all the major events that occurred during the run, such as checkpoints, error messages, system messages, or user messages.

If mismatches were detected at checkpoints, you can view the expected results and actual results. For bitmap mismatches, you can also view a bitmap that displays only the difference between the expected and actual results.

Working with LoadRunner

LoadRunner is Mercury Interactive's testing tool for client/server applications. Using LoadRunner, you can simulate an environment where many users are simultaneously engaged with a single server application. In place of human users, it creates virtual users that execute automated tests on the application under test. You can test an application's performance "under load" by simultaneously activating virtual users on multiple host computers.

LoadRunner is an independent tool and can be purchased separately from Mercury Interactive.

2

XRunner at a Glance

This chapter introduces the XRunner window and explains how to execute XRunner commands.

This chapter describes:

- Starting XRunner
- The Main XRunner Window
- Selecting XRunner Commands
- Configuring XRunner Softkeys

Starting XRunner

You start XRunner from your working directory by entering the command:

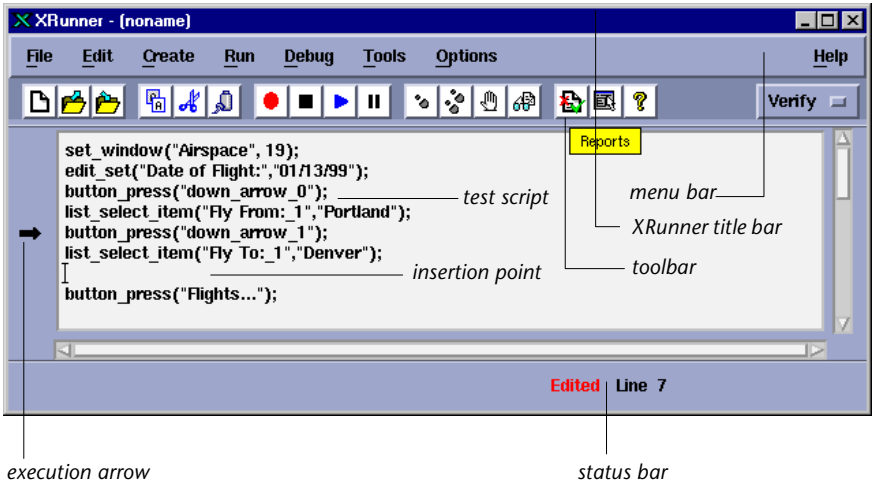
```
xrun &
```

After several seconds the main XRunner window is displayed on your desktop.

The Main XRunner Window

The main XRunner window contains the following key elements:

- the *XRunner title bar* displays the name of the test you are currently working on
- the *menu bar,* displaying menus with XRunner commands
- the *toolbar,* containing the commands you use most often
- the *status bar,* providing information on the line number of the insertion point, and the current mode in which XRunner is running
- the *test script*, consisting of statements generated by recording, programming, or both. These statements are in TSL, Mercury Interactive's Test Script Language
- the *execution arrow,* indicating the line of the test script being executed. To move the arrow to any line in the script, click the mouse in the left window margin next to the line
- the *insertion point,* indicating where text can be inserted or edited



Selecting XRunner Commands

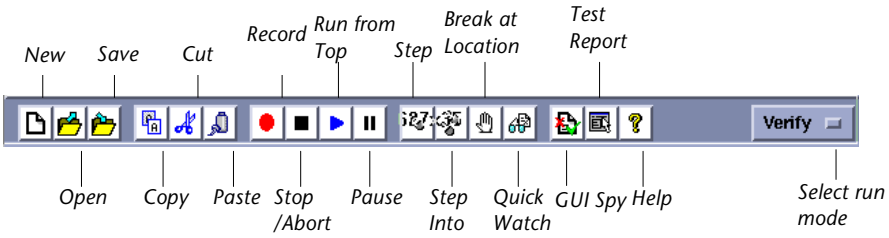
You can use the mouse to select XRunner commands from the menus and from the toolbar. Certain XRunner commands can also be activated using command softkeys.

Choosing Commands From a Menu

You can select XRunner menu commands using the mouse. In addition, some of these commands can be selected using standard X Windows accelerator key conventions. Other menu commands are selected using softkeys.

Choosing Commands From the Toolbar

You can execute some XRunner commands by clicking on an icon in the toolbar.



Choosing Commands Using Softkeys

Several XRunner commands can be activated using command softkeys. XRunner reads input from softkeys, even when the XRunner window is not the active window on your screen, or when it is minimized. The following table lists the default softkey configurations and their functions.

Command	Default Softkey Combination	Function
run from arrow	F8	Executes the test from the line in the script indicated by the execution arrow.
STEP	F7	Executes and indicates only the current line of the test script.
STEP INTO	Left Ctrl + F7	Like Step: however, if the current line calls another test, the called test is displayed in the XRunner window but not executed.
PAUSE	F8	Stops test execution after all previously interpreted TSL statements have been executed. Execution may be resumed from this point.
STOP/ABORT	F6	Stops test recording or aborts test execution.
GET TEXT	Left Ctrl + F1	Captures the text defined by the locator.
MARK LOCATOR	Left Ctrl + F5	Records a move_locator_abs statement with the current position (in pixels) of the screen pointer.
RECORD	F5	Starts test recording. During recording, the Record softkey toggles between Context Sensitive and Analog modes.
CHECK WINDOW	F2	Captures an entire window bitmap.

Command	Default Softkey Combination	Function
CHECK PARTIAL WINDOW	Left Ctrl + F2	Captures the part of the window defined by the locator.
WAIT WINDOW	F4	Instructs XRunner to wait for a specific window bitmap to be redrawn.
WAIT PARTIAL WINDOW	Left Ctrl + F4	Instructs XRunner to wait for a specific window area to be redrawn.
CHECK GUI (CHECKLIST)	Left Alt + F2	Opens the Check GUI form.
WAIT REDRAW WINDOW	F3	Instructs XRunner to wait for the window to be redrawn, without evaluating its contents.
WAIT REDRAW PARTIAL WINDOW	Left Ctrl + F3	Instructs XRunner to wait for a specific window area to be redrawn, without evaluating its contents.
INSERT FUNCTION (OBJECT/WINDOW)	Left Ctrl + F9	Insert a function for a GUI object, using the Function Generator.
INSERT FUNCTION (FROM LIST)	F9	Choose a function from the list in the Function Generator.

Configuring XRunner Softkeys

Softkey assignments are configurable. If the application you are testing uses one of the default softkeys preconfigured for XRunner, you can redefine softkey bindings using the appropriate XRunner system parameter.

You use the Configuration form to change XRunner softkey definitions.

To change XRunner softkey definitions:

1. Display the Configuration form by selecting Configure from the Options menu. The Configuration form opens.
2. Choose Initialization, then Softkeys. The current softkey definitions are displayed.

3 Modify the softkey definitions as needed.

4 To save all changes press Save.

5 For modified softkey settings to apply, you must restart XRunner.

Note: Any changes you make to the softkey configuration are automatically inserted into the .xrunner configuration file in your home directory.

Part II

Understanding the GUI Map

3

Introducing Context Sensitive Testing

This chapter introduces Context Sensitive testing and explains how XRunner identifies the Graphical User Interface (GUI) objects in your application.

This chapter describes:

- How a Test Identifies GUI Objects
- Physical Descriptions
- Logical Names
- The GUI Map
- Setting the Window Context

About Context Sensitive Testing

Context Sensitive testing lets you test your application the way that you see it: in terms of GUI objects, such as windows, menus, buttons, and lists. Each object has a defined set of properties that determine its behavior and appearance. XRunner learns these properties and uses them to identify and locate GUI objects during a test run. The physical location of the GUI objects on the screen is not important.

Before you can begin Context Sensitive testing, XRunner must learn the properties of each GUI object in your application. Use the Test Wizard to guide you through the learning process. It systematically opens each window in your application and learns the properties of the GUI objects it contains. Additional methods are available for learning the properties of individual GUI objects. For more information on the learning process, refer to Chapter 4, “Creating the GUI Map.”

GUI object properties are saved in the GUI map. XRunner uses the GUI map to help it locate objects during a test run. It reads an object's description in the GUI map and then looks for an object with the same properties in the application under test (AUT). You can open and view the GUI map in order to gain a complete picture of the objects in your application.

The GUI map protects your investment in test development. As the user interface of your application changes, you can continue to use previously created tests. You simply add or delete object descriptions in the GUI map, or edit existing descriptions so that XRunner can continue to find the objects in your application.

How a Test Identifies GUI Objects

You create tests by recording or programming *test scripts*. A test script consists of statements in Mercury Interactive's test script language (TSL). Each TSL statement represents mouse and keyboard input to the application under test. For more information, refer to Chapter 7, "Creating Tests."

XRunner uses an intuitive *logical name* to identify an object: for example "Open" for an Open form, or "OK" for an OK button. This short name connects XRunner to the object's longer *physical description*. XRunner uses this detailed description to ensure that each GUI object has a unique identification. The physical description contains a list of the object's physical properties, or *attributes*: the Open form, for example, is identified as a window with the label "Open".

Together, the logical name and the physical description form the *GUI map*. The following example illustrates the connection between the logical name and the physical description. Assume that you record a test in which you open a readme file using the Open form, and then press the OK button. The test script looks something like this:

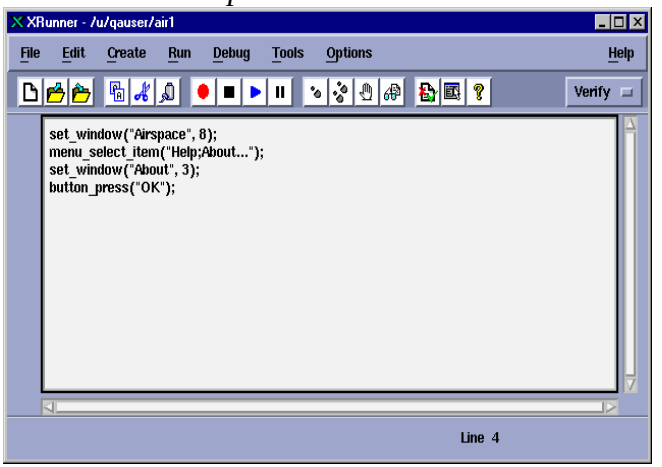
```
set_window ("Open");
list_select_item ("File Name"; "README");
button_press ("OK");
```

XRunner learns the actual description—the list of attributes and their values—for each of the three objects that are involved and writes this description in the GUI Map:

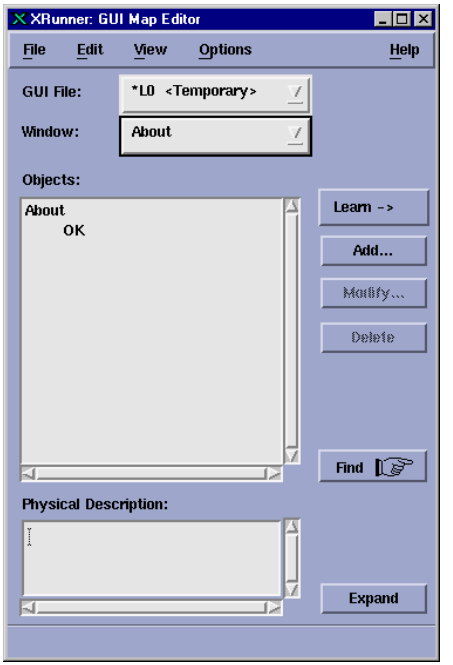
Open window: {class:window, label:Open}
File Name list box: {class:list, attached_text:File Name}
OK button: {class:push_button, label:OK}

XRunner also assigns a logical name to each object. As it runs the test, it reads the logical name of each object in the test script and refers to its physical description in the GUI map. XRunner then uses this description to find the object in the application under test.

Test Script



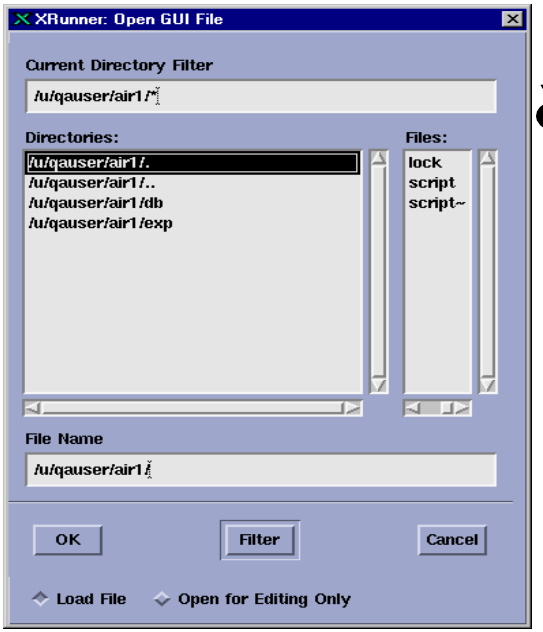
GUI Map



❶ *XRunner reads logical name in test script and refers to GUI map file*

❷ *Matches logical name with physical description*

AUT



❸ *Using physical description, locates object in AUT*

Physical Descriptions

XRunner identifies each GUI object in the application under test by its *physical description*: a list of physical properties, called *attributes*, and their assigned values. These attribute–value pairs appear in the following format in the GUI map:

{*attribute1:value1, attribute2:value2, attribute3:value3, ...*}

For example, the description of the "Open" window contains two attributes: class and label. In this case the class attribute has the value *window*, while the label attribute has the value *Open*:

```
{class:window, label:Open}
```

The class attribute indicates the type of the object. Every object belongs to a different class, according to its functionality: window, pushbutton, list, radio button, menu, etc. XRunner always identifies an object by learning at least its class attribute.

For each class, there is a set of default attributes: those attributes that XRunner always learns. For a detailed description of all attributes, refer to Chapter 6, "Configuring the GUI Map."

Note that XRunner always learns the physical description of an object in the context of the window in which it appears. This creates a unique physical description for each object. For more information, see "Setting the Window Context" in this chapter.

Logical Names

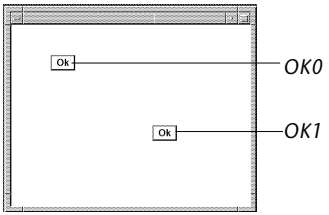
In the test script, XRunner does not use the full physical description for an object. Instead, it assigns a short, intuitive name to each object: the *logical name*.

An object's logical name depends on its class. In most cases, the logical name is the label that appears on an object: for a button, the logical name is its label, such as OK or Cancel; for a window, the logical name is the window title; for a list, the logical name is the text that appears next to or above the list.

For a static text object, the logical name is a concatenation of the text and the string "(static)". For example, the logical name of the static text "File Name" is: "File Name (static)".

In certain cases, several GUI objects in the same window are assigned the same logical name, using a location selector (for example, LogicalName1, LogicalName2.) The purpose of the selector attribute is to create a unique name for the object.

For example, if a window contains two buttons with the label OK, XRunner uses a location selector to distinguish between the buttons. The buttons are assigned numbers according to their positions within the window (from the upper left corner of the window to its lower right corner): *OK0,* and *OK1*.

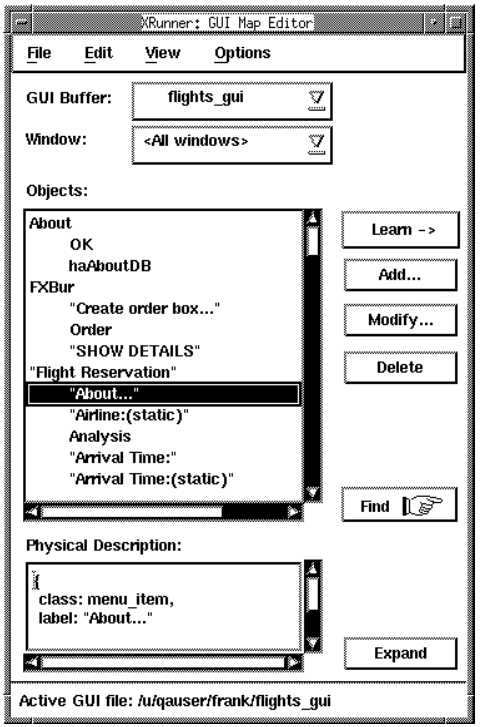


The GUI Map

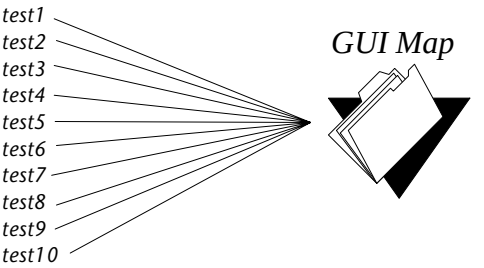
The GUI map is the sum of one or more GUI map files. These files contain the logical names and physical descriptions of GUI objects. In most cases, you store all the GUI object information for your application in a single GUI map file.

You can view the contents of the GUI map at any time by selecting GUI Map Edit from the Tools menu to open the GUI Map Editor. You can view either the contents of the entire GUI map, or the contents of individual GUI map

files. GUI objects are grouped according to the window in which they appear in the application.



The GUI map lets you easily keep up with changes made to the user interface of the application under test. Instead of editing your entire suite of tests, you only have to update the object descriptions in the GUI map.



For example, suppose the OK button in the Open form is changed to a Save button. You do not have to edit every test script which uses this OK button.

Instead, you can modify the OK button's physical description in the GUI map as in the example. The value of the label attribute for the button is changed from OK to Save:

OK button: {class:push_button, label:Save}

During a test run, when XRunner encounters in the test script the logical name "OK" in the Open form, it searches for a pushbutton with the label "Save".

You can use the GUI Map Editor to modify the logical names and physical descriptions of GUI objects at any time during the testing process. In addition, you can use the Run Wizard to update the GUI map during a test run. The Run Wizard opens automatically if XRunner cannot locate an object in the application under test. See Chapter 5, "Editing the GUI Map" for more information.

Setting the Window Context

XRunner learns and performs operations on objects in the context of the window in which they appear. When you record a test, XRunner automatically inserts a **set_window** statement into the test script each time the active window changes and an operation is performed on a GUI object. All objects are then identified in the context of that window. For example:

```
set_window ("Open");
list_select_item ("Filename"; "README");
button_press ("OK");
```

The **set_window** function indicates that the Open window is the active window. The file name and the OK button are learned within the context of this window.

When programming a test, you need to enter the **set_window** function manually when the active window changes. When editing a script, take care not to delete necessary **set_window** statements.

4

Creating the GUI Map

This chapter describes how to teach XRunner the Graphical User Interface (GUI) of the application under test and save the information for use during testing.

This chapter describes:

- Learning the GUI with Test Wizard
- Learning the GUI by Recording
- Learning the GUI Using the GUI Map Editor
- Saving the GUI Map
- Loading the GUI Map File

About Creating the GUI Map

XRunner can learn the GUI of your application in several ways. Usually, you use the Test Wizard before you start to test in order to learn all the GUI objects in your application at once. This ensures that XRunner has a complete, well-structured basis for all your Context Sensitive tests. The descriptions of GUI objects are saved in GUI map files. Since all testers can share these files, there is no need for each user to learn the GUI individually.

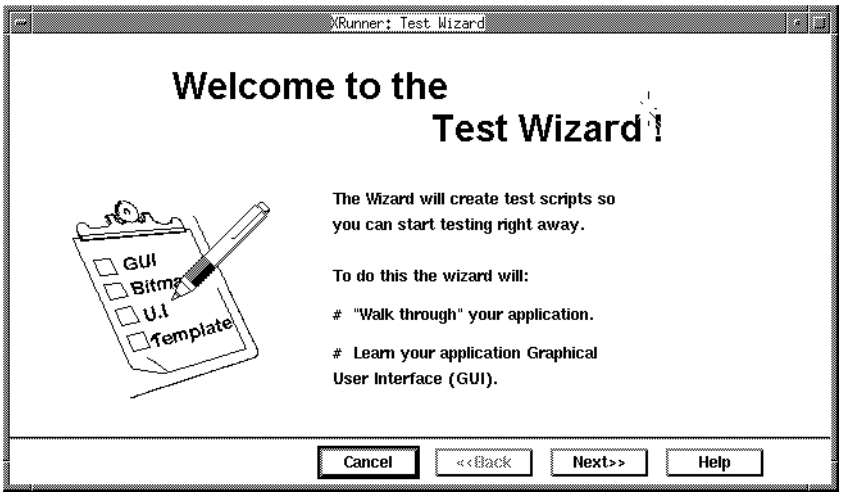
If the GUI of your application changes during the software development process, you can use the GUI Map Editor to learn individual windows and objects in order to update the GUI map. You can also learn objects while recording: you simply start to record a test and XRunner learns the properties of each GUI object you manipulate in your application. This approach is fast and lets a beginning user create test scripts immediately.

However, it is unsystematic, and should not be used instead of the Test Wizard if you are planning to develop comprehensive test suites.

You must load the appropriate GUI map files before you run tests. XRunner uses these files to help locate the objects in the application under test. The recommended method is to insert a **GUI_load** command into a startup test. When you start XRunner, it automatically runs the startup test and loads the specified GUI map files. For more information on startup tests, refer to Chapter 34, "Initializing Special Configurations." Alternatively, you can insert a **GUI_load** command into individual tests, or use the GUI Map Editor to load GUI map files manually.

Learning the GUI with Test Wizard

The Test Wizard allows XRunner to learn all windows and objects in your application at once. It systematically opens every window in the application and learns the GUI objects it contains. XRunner then instructs you to save the information in a GUI map file. A **GUI_load** command that loads this file is added to a startup test.



To start the Test Wizard, select the Test Wizard command from the Create menu at any time.

Learning the GUI by Recording

When you record a test, XRunner first checks whether the objects you select appear in the GUI map. If they do not, XRunner learns the objects and inserts them into the temporary GUI map file.

In general, you should use recording as a learning tool for small, temporary tests only. Use the Test Wizard to learn the entire GUI of your application.

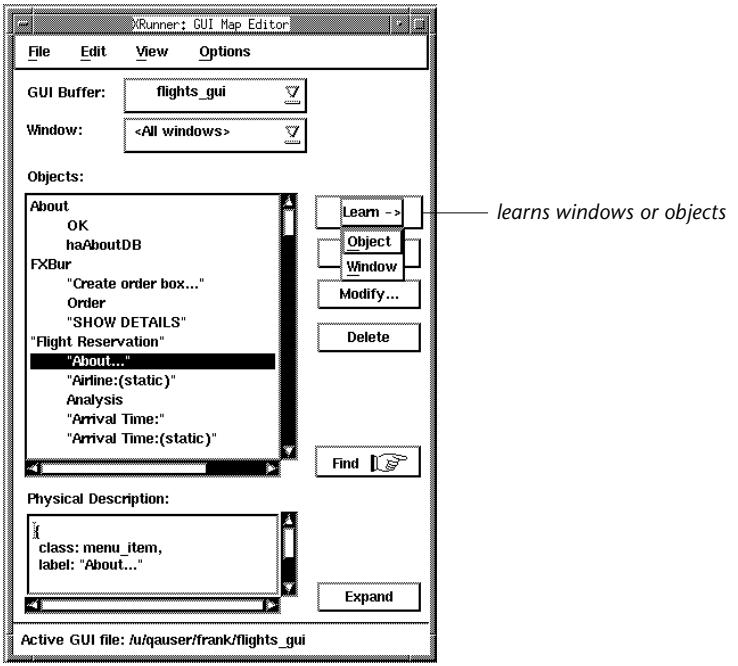
Learning the GUI Using the GUI Map Editor

You can use the GUI Map Editor to learn an individual object or window, or all objects in a window.

To learn GUI objects using the GUI Map Editor:

1 Select GUI Map Edit from the Tools menu to open the GUI Map Editor.

2 Press the Learn button. Select Window to learn a window or Object to learn an object. The mouse pointer turns into a pointing hand. (To cancel the operation, press the right mouse button.)



3 Place the hand on the desired object and click the left mouse button. If you point at a window, you are asked if you want to learn all objects within the window. Selecting No instructs XRunner to learn the window only.

GUI information about the learned objects is placed in the active GUI map file. See "Loading the GUI Map File" in this chapter for more information.

Note: XRunner learns objects according to its current default record configuration. In cases where you want XRunner to learn different attributes, you should modify the default record configuration before learning objects. For details on modifying the default configuration, see Chapter 6, "Configuring the GUI Map."

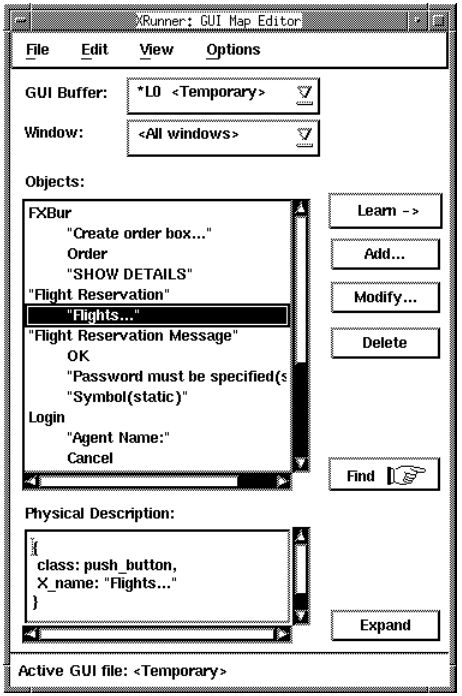
Saving the GUI Map

When you learn GUI objects using recording, the object descriptions are added to the temporary GUI map file. The temporary file is always open, so that any objects it contains are recognized by XRunner. When you start XRunner, the temporary file is loaded with the contents of the last testing session.

To prevent valuable GUI information from being overwritten during a new recording session, you should save the temporary GUI map files in a permanent GUI map file.

To save the contents of the temporary file in a permanent GUI map file:

1 Select GUI Map Edit from the Tools menu to open the GUI Map Editor.



2 Make sure that the <*Temporary*> file is displayed in the GUI File list. An asterisk (*) next to the file name indicates that the GUI map file was changed. The asterisk disappears when the file is saved.

3 Choose Save from the File menu of the GUI Map Editor to open the Save form.

4 Select a directory. Type in a new file name or select an existing file.

5 Press OK. The saved GUI map file is loaded and appears in the GUI Map Editor.

You can also move objects from the temporary file to an existing GUI map file. For details, refer to Chapter 5, "Editing the GUI Map."

Loading the GUI Map File

XRunner uses GUI map files to locate objects in the application under test. Before you run tests on your application, you must ensure that the appropriate GUI map files are loaded.

XRunner does not fully load a GUI map file that "shadows" objects contained in a previously loaded file. While loading the more recent file, XRunner deletes duplicate objects from the buffer.

You can load GUI map files in one of two ways:

- using the **GUI_load** command
- from the GUI Map Editor

You can view a loaded GUI map file in the GUI Map Editor. A loaded file is marked with the letter "L".

Loading GUI Map Files Using the GUI_load Command

The **GUI_load** statement loads any GUI map file you specify. To load several files, use a separate command for each one. You can insert the **GUI_load** statement at the beginning of any test, but it is preferable to place it in a startup test. This ensures that the GUI map files are loaded automatically each time you start XRunner. For more information, refer to Chapter 34, "Initializing Special Configurations."

To load a file using GUI_load:

1 Open the test from which you want to load the file.

2 Type the **GUI_load** statement as follows, or select the function from the Function Generator:

GUI_load ("*file_name_full_path*");

For example:

```
GUI_load ("/u/qa/flights.gui")
```

See Chapter 18, "Using Visual Programming" for more information on the Function Generator.

3 Run the test to load the file. See Chapter 24, "Running Tests" for more information.

Loading GUI Map Files Using the GUI Map Editor

You can load a GUI map file manually, using the GUI Map Editor.

To load a GUI map file from the GUI Map Editor:

1 Select GUI Map Edit from the Tool menu to open the GUI Map Editor.

2 Choose Open from the File menu and select a GUI map file.

Note that by default, the file is loaded into the GUI map. If you only want to edit the GUI map file, select the "Open for Editing Only" checkbox. See Chapter 5, "Editing the GUI Map" for more information.

3 Click OK. The GUI map file is added to the GUI file list. The letter "L" indicates that the file is loaded.

5

Editing the GUI Map

This chapter describes how you can extend the life of your tests by modifying descriptions of objects in the GUI map.

This chapter describes:

- The Run Wizard
- The GUI Map Editor
- Modifying Logical Names and Physical Descriptions
- Using a Single Description for Varying Labels
- Copying and Moving Objects between Files
- Finding an Object in a GUI Map File
- Finding an Object in Multiple GUI Map Files
- Manually Adding an Object to a GUI Map File
- Deleting an Object from a GUI Map File
- Clearing a GUI Map File
- Filtering Displayed Objects
- Saving Changes to the GUI Map

About Editing the GUI Map

XRunner uses the GUI map to identify and locate GUI objects in your application. If the GUI of your application changes, you must update object descriptions in the GUI map so that you can continue to use previously created tests.

You can update the GUI map in two ways:

- during a test run, using the Run Wizard
- at any time during the testing process, using the GUI Map Editor

The Run Wizard opens automatically during a test run if XRunner cannot locate an object in the application under test. It guides you through the process of identifying the object and updating its description in the GUI map. This ensures that XRunner will find the object in subsequent test runs.

You can also manually edit the GUI map using the GUI Map Editor. You can modify the logical names and physical descriptions of objects, add new descriptions and remove obsolete descriptions. You can also move or copy descriptions from one GUI map file to another.

Note that before you can update the GUI map, the appropriate GUI map files must be loaded. You can load files using the **GUI_load** statement in a test script or using the Open command in the GUI Map Editor. See Chapter 4, "Creating the GUI Map" for more information.

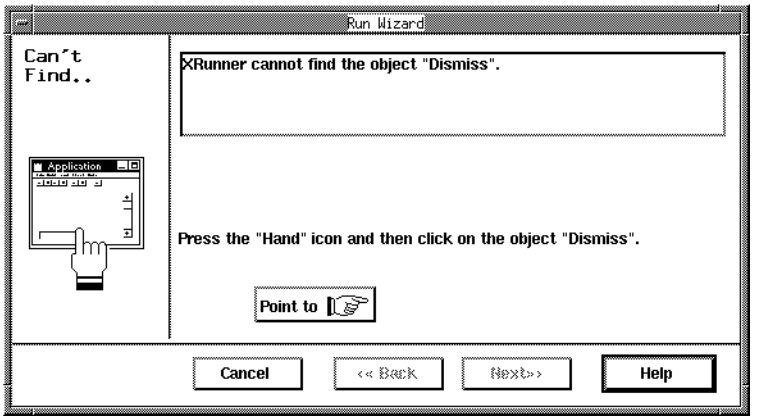
The Run Wizard

The Run Wizard detects changes in the GUI of your application that interfere with test execution. During a test run, it automatically opens when XRunner cannot locate an object. The Run Wizard asks you to point to the object in your application, determines why the object could not be found, and then offers a solution. the Run Wizard suggests loading an appropriate GUI map file; in most cases a new description is automatically added to the GUI map or an existing description is modified. When this process is completed, the test run continues. In future test runs, XRunner can successfully locate the object.

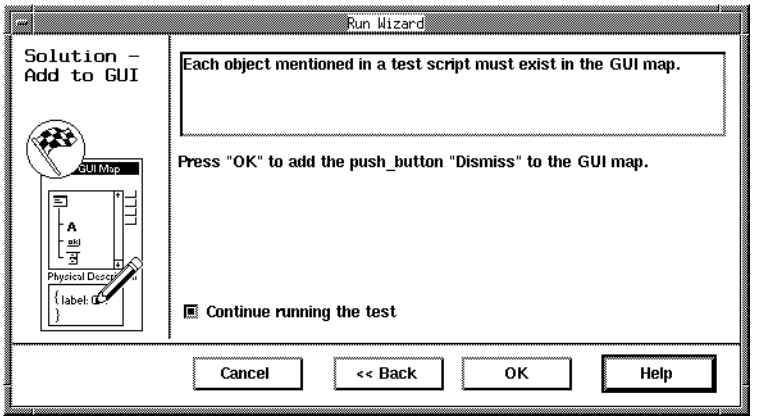
For example, assume you run a test in which you press the Dismiss button in an Open window.

```
set_window ("Open");
button_press ("Dismiss");
```

If the Dismiss button is not in the GUI map, the Run Wizard opens, and describes the problem.



Press the hand button in the wizard and point to the Dismiss button. The Run Wizard then offers a solution.



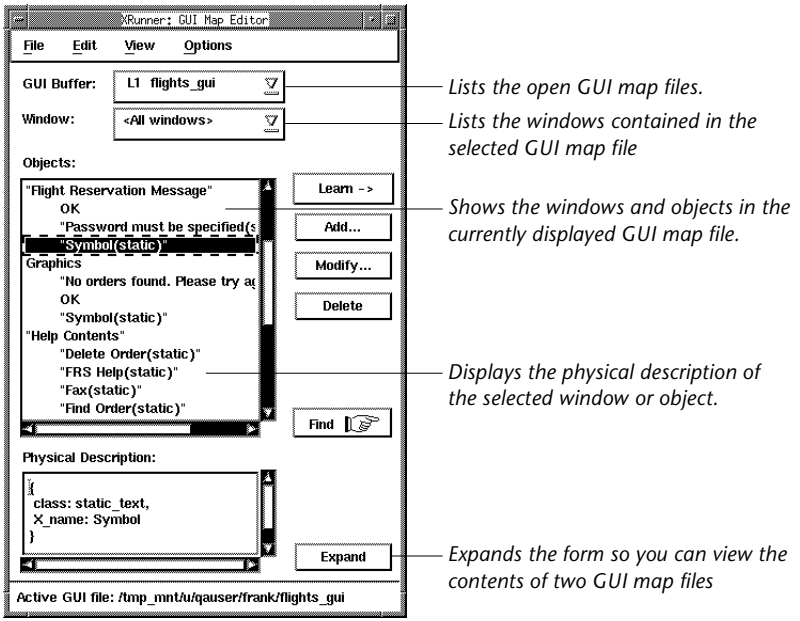
When you press OK, the Dismiss object description is automatically added to the GUI map and XRunner resumes test execution. The next time you run the test, XRunner is able to identify the Dismiss button.

Note that in some cases, the Run Wizard edits the test script instead of the GUI map. For example, if XRunner cannot locate an object because the appropriate window is not active, the Run Wizard inserts a **set_window** statement in the test script.

The GUI Map Editor

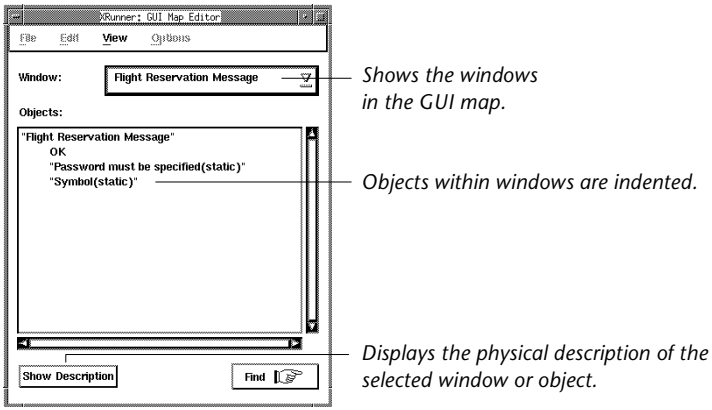
You can edit the GUI map using the GUI Map Editor. To open the GUI Map Editor, select GUI Map Edit from the Tools menu.

Two views are available in the GUI Map Editor. By default, the contents of individual GUI map files are displayed.



When viewing the contents of specific GUI map files, you can expand the GUI Map Editor to view two GUI map files simultaneously. This allows you to easily copy or move descriptions between files.In the GUI Map Editor, objects are presented in a tree according to the window in which they appear. A drop down list in the Windows field displays the windows contained in a particular GUI file. Click the name of a window to display the objects it contains, or click *<All Windows>* to display all the windows in the GUI map file, and the objects they contain.

To view the contents of the entire GUI map, select GUI Map from the View menu.



When you view the GUI map, click the Show Physical Description button to display the physical description of any object you select from the Windows list.

For example, if you click on the Show Description button and select the Flight Reservation Message window in the Windows list, the following physical description is displayed in the lower part of the GUI Map Editor:

```
{
class: window,
label: "!Flight Reservation Message.*"
}
```

Note that if the value of an attribute contains any spaces or special characters (such as the exclamation mark and asterisk in the above example), the value is surrounded by quotation marks.

Modifying Logical Names and Physical Descriptions

You can modify the logical name or the physical description of an object in a GUI map file using the GUI Map Editor.

Changing the logical name of an object is useful when the logical name that is assigned is not sufficiently descriptive, or is too long. For example, suppose XRunner assigns the logical name Employee Address (static) to a static text object. You can change the name to Address to make the scripts easier to read.

Changing the physical description is necessary when an attribute value of an object changes. For example, suppose that the label of a button is changed from Insert to Add. You can modify the value of the label attribute in the physical description of the Insert button as shown below:

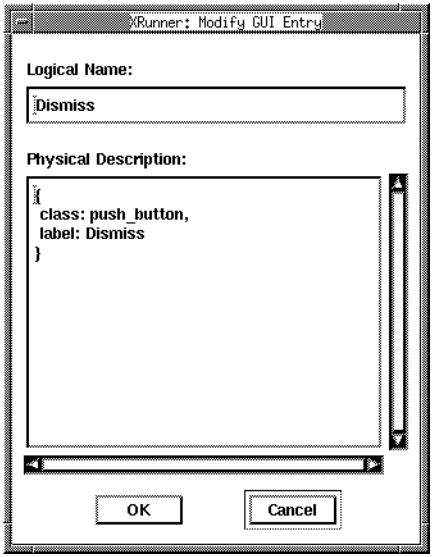
```
Insert button:{class:push_button, label:Add}
```

During a test run, when XRunner encounters the logical name Insert in a test script, it searches for the button with the label Add.

To modify an object's logical name or physical description in a GUI map file:

1. Choose GUI Map Edit from the Tools menu. The GUI Map Editor opens.
2. If the appropriate GUI map file is not loaded, select File > Open in the GUI Map Editor to open the file.
3. To see the objects in a window, display the drop down list of windows in the Windows field. Click on the window name you need.
4. Select the window or object to modify.

5 Click Modify. The Modify GUI Entry form is displayed.



6 Edit the logical name or physical description as desired and click OK. The change appears immediately in the GUI map file.

Using a Single Description for Varying Labels

For example, suppose your application contains a Start button whose label sometimes toggles to Stop. You can define a regular expression in the physical description of the button as follows:

```
Start button:{class:push_button, label: "!St.*"}Dismiss
```

The regular expression in the value of the label attribute enables XRunner to identify the button as long as the letters “St” are displayed and to automatically ignore any other variations.

To use a regular expression in the description of a single object:

1 Select GUI Map Edit from the Tools menu to open the GUI Map Editor.

2 In Windows/Objects list, select the object you want to modify.

3 Click Modify to open the Modify form.

4 In the Modify form, edit the attribute to conform to the following format:

{attribute: "*!regular_expression*"}

The exclamation point indicates that this is a regular expression. For more information, refer to Chapter 23, "Using Regular Expressions."

For example, to change the label of the "Print Out # 3" window, to a generic Print Out window, you enter:

{label: "!Print Out**.***"}

Copying and Moving Objects between Files

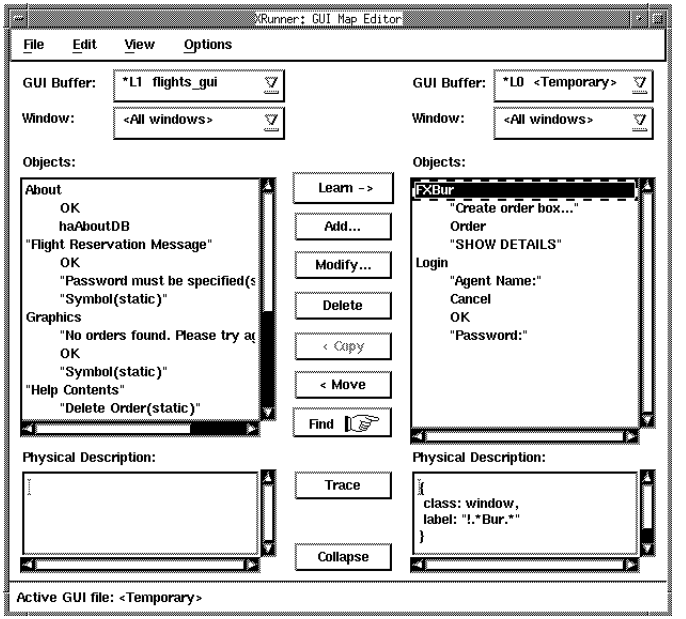
You can update GUI map files by copying or moving the description of GUI objects from one GUI map file to another. Note that you can only copy objects from a GUI file that you have opened for editing only.

To copy or move objects between two GUI map files:

1 Choose GUI Map Edit from the Tools menu to open the GUI Map Editor.

2 Select File > Open in the GUI Map Editor to open both GUI map files.

3 Press the Expand button in the GUI Map Editor. The form expands to display two GUI map files simultaneously.



4 Display a different GUI map file on each side of the form by selecting the file names in the GUI File lists.

5 Select in one file the object(s) that you want to copy or move. To select all objects in a window, choose Select All from the Edit menu.

6 Press the Copy or Move button.

7 To restore the GUI Map Editor form to its original size, press the Collapse button.

Finding an Object in a GUI Map File

You can easily find the description of a specific object in a GUI map file by pointing to the object in the application under test.

To find an object in a GUI map file:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Select File > Open to load the GUI map file.
3. Press Find. The mouse pointer turns into a pointing hand.
4. Click on the object in the application under test. The object is highlighted in the GUI map file.

Finding an Object in Multiple GUI Map Files

If an object is described in more than one GUI map file, you can quickly locate all the object descriptions using the Trace button in the GUI Map Editor. This is particularly useful if you learn a new description of an object and want to find and delete older descriptions in other GUI map files.

To find an object in multiple GUI map files:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Open the GUI Map files in which the object description might appear.

   For each file, select File > Open to open the Open GUI File form. Choose the GUI map file you want to open and deselect the Load File checkbutton. Click OK.
3. Display the contents of the file with the most recent description of the object.
4. Select the object in the Objects field.
5. Press the Expand button to expand the GUI Map Editor form.
6. Press the Trace button. The GUI map file in which the object is found is displayed on the other side of the form, and the object is highlighted.

Manually Adding an Object to a GUI Map File

You can manually add an object to a GUI map file by copying the description of another object, and then editing it as needed.

To manually add an object to a GUI map file:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Select File > Open in the GUI Map editor to open the appropriate GUI map file.
3. Select the object to be used as the basis for editing.
4. Press the Add button to open the Add form.
5. Edit the appropriate fields and press OK. The object is added to the GUI map file.

Deleting an Object from a GUI Map File

If an object description is no longer needed, you can delete it from the GUI map file.

To delete an object from a GUI map file:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Select File > Open in the GUI Map Editor to open the appropriate GUI map file.
3. Select the object to be deleted. If you want to delete more than one object, use the Shift key to make your selection.
4. Press the Delete button.
5. Select File > Save to save the changes to the GUI map file.

To delete all objects in a window:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Select File > Open in the GUI Map Editor to open the appropriate GUI map file.
3. Choose Clear All from the Edit menu.

Clearing a GUI Map File

You can quickly clear the entire contents of the temporary GUI map file, or any other GUI map file.

To delete the entire contents of a GUI map file:

1. Select GUI Map Edit from the Tools menu to open the GUI Map Editor.
2. Open the appropriate GUI map file.
3. Display the GUI map file at the top of the GUI File list.
4. Choose Clear All from the Edit menu.

Filtering Displayed Objects

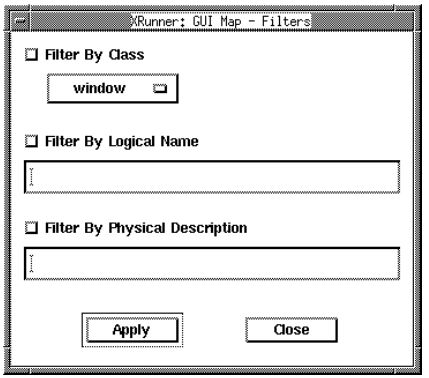
You can filter the list of objects displayed in the GUI Map Editor by using any of the following filters:

- *Logical name* displays only objects with the specified logical name or substring (for example, "Open" or "Op").
- *Physical description* displays only objects that match the specified physical description. Use any substring belonging to the physical description (for example, specifying "w" will filter out all objects that contain a "w" in their physical description).
- *Class* displays only the objects of the specified class, such as all the pushbuttons.

To apply a filter:

1 Select GUI Map Edit from the Tools menu to open the GUI Map Editor.

2 Select Filters from the Options menu to open the GUI Filters form.



3 Select the type of filter you want by clicking a checkbox and entering the appropriate information.

4 Press the Apply button. The GUI Map Editor displays objects according to the filter applied.

Saving Changes to the GUI Map

If you edit the logical names and physical descriptions of objects in the GUI map, you must save the changes in the GUI Map Editor before ending the testing session and exiting XRunner.

To save changes to the GUI map:

➤ Select File > Save in the GUI Map Editor to save changes in the appropriate GUI map file.

➤ Select File > Save As to save the changes in a new GUI map file.

6

Configuring the GUI Map

This chapter describes how you can configure the way XRunner identifies GUI objects during Context Sensitive testing.

This chapter describes:

- Viewing GUI Object Attributes
- Understanding the Default GUI Configuration
- Identifying Objects with the Same Name
- Configuring Record Attributes
- Configuring the Record Method
- Configuring the Selector
- Configuring the Record Method For a Specific Object
- The Class Attribute
- All Attributes
- Default Attributes Learned
- Working with Motif and Xt Resources

About Configuring the GUI Map

Each GUI object in the application under test (AUT) is defined by multiple attributes, such as class, label, x, y, width, and height. XRunner uses these attributes to identify GUI objects in the AUT during Context Sensitive testing.

When XRunner learns the description of a GUI object, it does not learn all of its attributes. Instead, it learns the minimum number of attributes that enables a unique identification of the object. For each object class (such as push_button, list, window, or menu), XRunner learns a default set of attributes: its GUI configuration.

For example, a standard push button is defined by 26 attributes (class, label, text, nchildren, x, y, height, class, focused, enabled, etc.). In most cases, however, XRunner needs only the *class* and *label* attributes to create a unique identification for the push button.

Many applications also contain custom objects. A custom object is any object whose class XRunner cannot identify. These objects are therefore learned under the general class "object". For more information on implementing support for custom objects, refer to Chapter "Testing Custom Widgets: Built-in Support" and both in the XRunner Customization Guide.

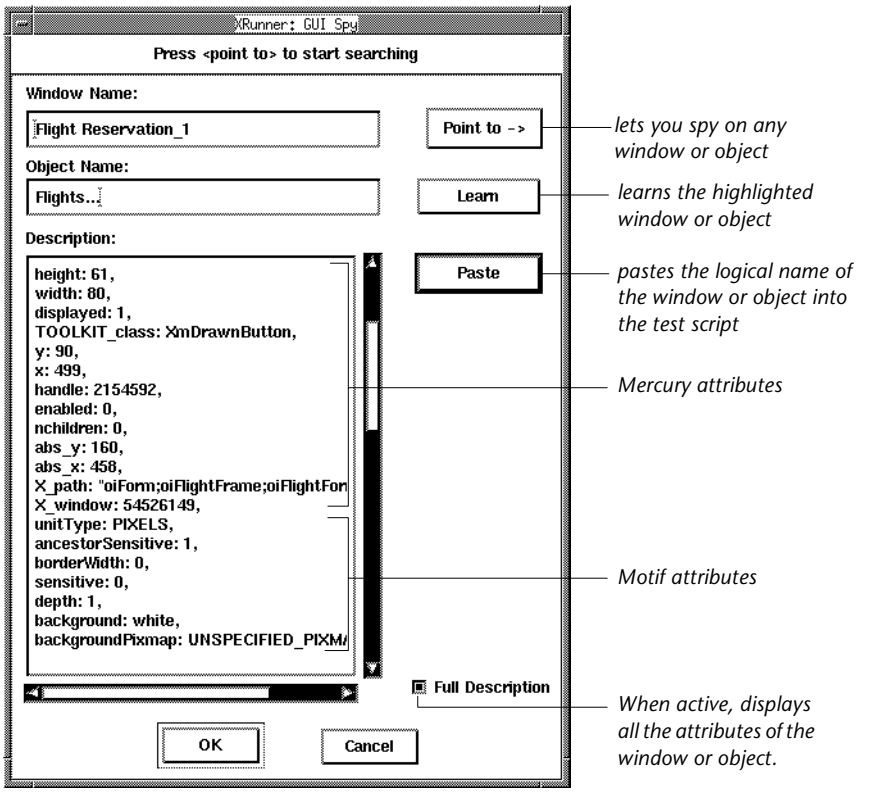
Attributes vary in their degree of portability. Some are unique to a specific platform (non-portable), such as X_name or X_path. Some are semi-portable (supported by multiple platforms, but their value is likely to change), such as handle, or TOOLKIT_class. Others are fully portable (such as label, attached_text, enabled, focused or parent).

Viewing GUI Object Attributes

Using the GUI Spy, you can view the attributes of any GUI object on your desktop. You simply point to an object and the GUI Spy displays the attributes and their values in the GUI Spy form. You can choose to view all the attributes of an object, or only the default set of attributes defined for the object class.

To spy on a GUI object:

1 Choose the GUI Spy command from the GUI menu. The GUI Spy form opens.



2 To view all the attributes defined for an object, click on the Full Description checkbox. XRunner displays all the attributes of the object, including Motif and Xt resources. To better view all the attributes displayed in the GUI Spy, expand the Description field by resizing the form.

If the Full Description checkbox is not activated, the GUI Spy displays only the default set of attributes for an object.

3 Press the "Point to" button. Click Window to spy on a window; click Object to spy on an object. Point to an object on the screen. The object is highlighted and the active window name, object name, and object description (attributes-values) are displayed in the appropriate fields.

Note that as you move the pointer over other objects, each one is highlighted in turn and its description is displayed in the form.

4. To capture an object description in the GUI Spy form, point to the desired object and press the STOP softkey.
5. Click Learn to learn the window or object, without closing the GUI Spy form.
6. Click OK to learn the window or object, and close the GUI Spy form.
7. Click Cancel to ignore changes and close the form.

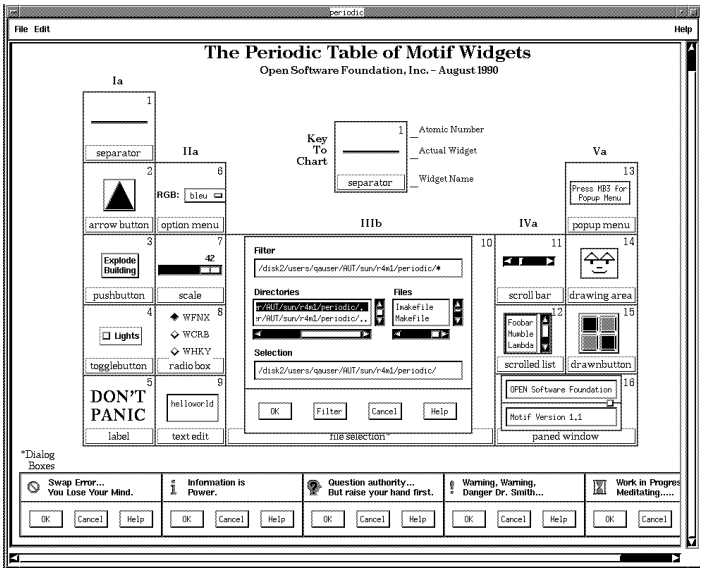
Understanding the Default GUI Configuration

For each class, XRunner learns a set of default attributes. Each default attribute is classified as either obligatory or optional. (For a list of the default attributes, see "All Attributes" in this chapter.)

- An *obligatory* attribute is one that is always learned (if it exists).
- An *optional* attribute is used only if the obligatory attributes do not provide a unique identification of an object. Optional attributes are stored in a list, from which XRunner selects the minimum number needed to identify the object. XRunner begins with the first attribute in the list, continuing if necessary to add attributes to the description until it obtains a unique identification for the object.

  For example, for the push_button class, the default obligatory attributes are class and label. The optional attribute list contains the X_name and class_index attributes.

Examine the buttons in the Periodic application.



If you use the GUI Spy utility (which you access by selecting GUI Spy from the Tools menu) to view the attributes of the Filter button, XRunner displays the attributes class and label. The physical description of this button is therefore:

```
{class:push_button, label:Filter}
```

For the OK buttons in the form, by contrast, XRunner learns the attributes X_name and class_index, as well. The description of the bottom left OK button, for example, is:

```
{class:push_button, label:OK, X_Name:OK, class_index:24}
```

This is because the values of the class and label obligatory attributes are the same for all six buttons. XRunner adds the X_name optional attribute, but its value is also the same for all buttons. By adding the class_index attribute which is different for each button, the buttons are uniquely identified.

Identifying Objects with the Same Name

When the obligatory and optional attributes do not achieve a unique identification of an object XRunner uses the following selectors to uniquely identify the objects.

- A *location* selector uses the spatial position of objects.
- An *index* selector uses the object's hierarchical position within the parent window.

For instance, suppose you configure the record attributes for push buttons in your application as follows:

Obligatory Attributes: class, label

Optional Atttributes: none

In such a case, XRunner uses the locator selector to uniquely identify two OK buttons appearing in the same window of the application.

The location selector uses the spatial order of objects within the window, from the top left to the bottom right corners, to differentiate between the buttons. In this example, the description of the upper OK button is:

```
{class:push_button, label:OK, location:0}
```

The lower OK button has a location of "1".

Note that the selector is applied only after all attributes currently set for the object class do not achieve a unique identification of a single object. The selector thus appears last in the physical description.

For more information on setting record attributes, See "Configuring Record Attributes," below.

Configuring Record Attributes

For each class of object, XRunner learns a set of default attributes. For a full description of the default attribute learned for each class, see "Default Attributes Learned", on page 64.

Sometimes, you need to configure XRunner to learn different attributes for a particular class. For example, let's suppose your application uses an Edit drop-down menu containing two Copy items. Using the regular configuration, XRunner would be unable to achieve a unique description for each menu item. In such a case, you could add the attribute *position* to the list of obligatory attributes for the menu_item class. XRunner would then be able to distinguish between the two menu items.

You can configure the attributes XRunner records and learns for a particular class using either of two methods:

- Modifying record attribute parameters from the Configuration form
- Using a **set_record_attr** command in a test script.

Before you begin, note that not all attributes apply to all classes. The following table lists each attribute and the classes to which it can be applied:

Attribute	Classes
class	all classes
label	push_button, radio_button, check_button, window, static, (object)
attached_text	edit, list, scroll, spin, notebook, (object)
x	all classes
y	all classes
height	all classes
width	all classes
image_label	pushbutton, check_button, radio_button
displayed	all classes

Attribute	Classes
count	list, menu_item, scroll
parent	menu_item
position	menu_item
sub_menu	menu_item
focused	all classes
handle	all classes
active	all classes
enabled	all classes
value	check_button, radio_button, list, scroll, static_text, edit, spin
nchildren	all classes
minimizable	window
maximizable	window
abs_x	all classes
abs_y	all classes
class_index	all classes
mic_if_handles_windows	Java
orientation	scroll

Configuring Record Attributes in the Configuration Form

You use the Configuration form to modify class record attribute parameters. Each parameter corresponds to a particular class, as the table below illustrates. For each class record attribute parameter, you specify the obligatory and optional attributes as well as the selector. For more information on using the Configuration form, see Chapter 33, "Changing System Defaults."

Parameter	Class
XR_WINDOW_REC_ATTR	window
XR_PBUTTON_REC_ATTR	push_button
XR_CBUTTON_REC_ATTR	check_button
XR_RBUTTON_REC_ATTR	radio_button
XR_LIST_REC_ATTR	list
XR_EDIT_REC_ATTR	edit
XR_SCROLL_REC_ATTR	scroll
XR_MENU_REC_ATTR	menu_item
XR_STATIC_REC_ATTR	static_text
XR_NOTEBOOK_REC_ATTR	notebook
XR_SPIN_REC_ATTR	spin
XR_OBJ_REC_ATTR	object

To configure the record attributes from the Configuration form:

1. Display the Configuration form by selecting Configure from the Options menu.
2. Click the Recording tab. Click the Attributes tab. XRunner displays the class record attribute parameters.
3. Scroll the form if necessary, until the class record attribute parameter you want is displayed. Make changes to the obligatory and optional attributes and selector as necessary.

4 Click Apply to apply the changes for the current XRunner session only. Click Save to save the changes for current and future sessions.

Configuring Record Attributes from a Test Script

You can use a **set_record_attr** TSL command to modify the attributes XRunner learns or records for the current session. The **set_record_attr** function has the following syntax:

set_record_attr (*class*, *oblig_attr*, *optional_attr*, *selector***);**

Below are two examples that illustrate how to configure a standard class or a TOOLKIT_class.

For instance, to configure the menu_item class so that the sub_menu attribute is in the obligatory list instead of in the optional list, enter the following:

```
set_record_attr ("menu_item", "class label sub_menu", "X_name class_index",
"location");
```

By default, a custom object is learned as the class object. Therefore, the attributes learned are the default attributes of the object class. By configuring a custom object, you can customize the learning and recording of attributes for that object.

To configure a custom object, use the **set_record_attr** function with the value of the TOOLKIT_class attribute of the object.

The example below shows how to configure a custom drawing object so that its X_name attribute is in the optional list, instead of in the obligatory attribute list.

Use the GUI Spy form (with Full Description selected) to find the value of the TOOLKIT_class attribute (here, MyClass). Then use **set_record_attr** as follows:

```
set_record_attr("MyClass", "class label", "X_name class_index", "location");.
```

For more information on the **set_record_attr** function, refer to the *TSL Reference Guide*.

Configuring the Record Method

By configuring the record method, you can determine the way XRunner records mouse clicks and drags and key strokes on objects of a particular class. You can configure a standard Mercury class, or a custom class.

For example, XRunner records all custom class objects as the class object, using the **obj_mouse_click** and **obj_mouse_drag** functions. Instead, you might want a particular custom object to be recorded using the **win_mouse_click** or **win_mouse_drag** functions. In such a case, you modify the record method for the TOOLKIT_class to MIC_MOUSE_WIN. Using this configuration, XRunner records mouse operations relative to the top level window of the custom object.

You configure the record method of a particular class using either of two methods:

- Modifying class record method parameters from the Configuration form
- Using a **set_record_method** command in a test script.

Before you begin, study the table, below, describing the ten different record methods available.

Method	Description
MIC_RECORD_CS	Records operations using Context Sensitive functions. This is the default method for all the standard classes, except the *object* class (for which the default is MIC_MOUSE).
MIC_IGNORE	Turns off recording.
MIC_KEYBOARD	Records *only* keyboard operations (when keyboard focus is set to the object of that class), using the **type** function.
MIC_MOUSE	Records *only* mouse operations relative to the upper left corner of the object/window on which the operation was performed.
MIC_ALL	Records mouse operations (relative to the object/window on which the operation was performed) *and* keyboard input.

Method	Description
MIC_MOUSE_PARENT	Records *only* mouse operations, relative to the parent (one level up) of the object on which the operation was performed.
MIC_ALL_PARENT	Records mouse operations (relative to the parent of the object on which the operation was performed) and keyboard input.
MIC_MOUSE_WIN	Records *only* mouse operations, relative to the top level window of the object on which the operation was performed.
MIC_ALL_WIN	Records mouse operations (relative to the parent window of the object on which the operation was performed) *and* keyboard input.
MIC_RECORD_ ANALOG	Records *all* mouse-clicks, keyboard input and the exact coordinates traveled by the mouse.

Configuring the Record Method in the Configuration Form

You use the Configuration form to modify class record method parameters. Each parameter corresponds to a particular class, as the table below illustrates. For more information on using the Configuration form, see Chapter 33, "Changing System Defaults."

Parameter	Class
XR_WINDOW_REC_METHOD	window
XR_PBUTTON_REC_METHOD	push_button
XR_CBUTTON_REC_METHOD	check_button
XR_RBUTTON_REC_METHOD	radio_button
XR_LIST_REC_METHOD	list_item
XR_EDIT_REC_METHOD	edit
XR_SCROLL_REC_METHOD	scroll
XR_MENU_REC_METHOD	menu_item
XR_STATIC_REC_METHOD	static_text
XR_NOTEBOOK_REC_METHOD	notebook
XR_SPIN_REC_METHOD	spin
XR_OBJ_REC_METHOD	object

To configure the record method using the Configuration form:

1. Display the Configuration form by selecting Configure from the Options menu.
2. Click the Recording tab. Click the Methods tab. XRunner displays the class record method parameters.
3. Display the drop down list of methods for the class record method parameter you want to modify.
4. Click on the desired method.
5. Click Apply to apply the changes for the current XRunner session only. Click Save to save the changes for current and future sessions.

Configuring the Record Method from a Test Script

You can use a **set_record_method** TSL command to modify the attributes XRunner learns or records for the current session. The **set_record_method** function has the following syntax:

set_record_method (*class, method***);**

For example, assume you want XRunner to ignore operations performed on a custom drawing object. First, you find out the value of the TOOLKIT_class attribute of the object using GUI Spy with Full Description selected. Assume it is MyClass.

Next, you use **set_record_method**, as follows:

```
set_record_method ("MyClass", MIC_IGNORE);
```

For more information on the **set_record_method** function, refer to the *TSL Reference Guide*.

Configuring the Selector

In cases where both obligatory and optional attributes cannot uniquely identify an object, one of two selectors is applied: *location* or *index*.

A location selector performs a selection according to the position of objects within the window: from top to bottom and from left to right. An index selector performs a selection according to the object's position in relation to the hierarchy of objects within the entire screen. For an example of how selectors are used, see "Understanding the Default GUI Configuration" in this chapter.

By default, XRunner uses a location selector for all classes. You can change the selector by modifying the settings for class record attribute parameters in the Configuration form (see "Configuring Record Attributes", on page 51).

Configuring the Record Method For a Specific Object

You can configure the record method for a specific standard or custom object.

Suppose while conducting Context Sensitive testing on a CAD/CAM application, you wish to record the exact lines drawn by the mouse-cursor inside a drawing area widget. You can configure the record method for the drawing area object to MIC_RECORD_ANALOG. XRunner now records all operations you perform in the drawing area in Analog mode.

To configure the record method for specific objects, you use a **set_obj_record_method** statement in a test script. The function has the following syntax:

set_obj_record_method (*win*, *obj*, *method* **);**

In the example above, you insert the following line into a test script:

```
set_obj_record_method (“DrawWindow”,”DrawArea”,
MIC_RECORD_ANALOG );
```

For more information about the **set_obj_record_method** function, refer to the *TSL Reference Guide*.

The Class Attribute

The *class* attribute is the prime attribute used by XRunner to identify the type of GUI object. XRunner categorizes GUI objects according to the following classes:

Classes	Description
push_button	A push (command) button.
check_button	A check button.
radio_button	A radio (option) button.
list	A list box. This can be a regular list, a combo box, an option menu or a container.

Classes	Description
edit	A single of multiple line edit field.
scroll	A scroll bar, scale, or scroll box.
menu_item	A menu item or cascade button (in Motif).
static_text	Display-only text which is not part of any GUI object.
notebook	A notebook.
spin	A spin object.
object	All objects not included in one of the classes described above.
window	Any application window, dialog box or form.

All Attributes

The following tables list all attributes used by XRunner in Context Sensitive testing. Attributes are listed according to their portability levels: portable, semi-portable, and non-portable.

Portable Attributes

Attribute	Description
abs_x	The x coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
abs_y	The y coordinate of the top left corner of an object, relative to the origin (upper left corner) of the screen display.
attached_text	The static text located near the object.
class	See "The Class Attribute," above.
class_index	The hierarchical position of the object among other objects of the same class within the parent window.
comment	Comment that user can include in the physical description.
count	List objects: indicates the number of items in the list. Menu_item objects: indicates the number of menu items a menu contains.
displayed	A Boolean value indicating whether the object is displayed: 1 if visible on screen, 0 if not.
enabled	A Boolean value indicating whether the object can be selected or activated: 1 if enabled, 0 if not.
focused	A Boolean value indicating whether keyboard input will be directed to this object: 1 if object has keyboard focus, 0 if not.
height	Height of object in pixels.
host	The name of the host on which the application under test is running.
label	The text that appears on the object, such as a button label or window title.
maximizable	A Boolean value indicating whether a window can be maximized. 1 if the window can be maximized, 0 if not.
mic_if_handles_windows	Returns 1 if the the item is a window and 0 if the item is an object. (For Java only.)
minimizable	A Boolean value indicating whether a window can be minimized: 1 if the window can be minimized, 0 if not.

Attribute	Description
nchildren	The number of children the object has: the total number of descendants of the object.
orientation	VSCROLL indicates a vertical scroll; HSCROLL indicates a horizontal scroll.
parent	The logical name of the parent of the object.
position	Specifies the position (top to bottom) of a menu item within the menu (the first item is at position 0).
submenu	A Boolean value indicating whether a menu item has a sub-menu: 1 if menu has submenu, 0 if not.
value	Different for each class: Radio and check buttons: 1 if the button is checked, 0 if not. Menu_items: 1 if the menu is checked, 0 if not. List objects: indicates the text string of the selected item. Edit/Static objects: indicates the text field contents. Scroll objects: indicates the scroll position. Spin objects: indicates the contents. All other classes: the value attribute is a null string.
width	Width of object in pixels.
x	The x coordinate of the top left corner of an object, relative to the window origin.
y	The y coordinate of the top left corner of an object, relative to the window origin.

Semi-Portable Attributes

Attribute	Description
handle	A run-time pointer to the object: the widget pointer.
image_label	Identifies labels that use pixmaps instead of strings. Format: image_<*checksum*>, where checksum is a unique number.
TOOLKIT_class	The value of the toolkit class. The value of this attribute is the same as the value of the X_class.

Non-Portable X Attributes

Attribute	Description
left_footer	(XView only) The Xview left window footer.
right_footer	(XView only) The Xview right window footer.
X_arrow	(Motif only) The direction of an arrow button ("up_arrow"; "down_arrow"; "right_arrow"; or "left_arrow").
X_attached_name	The X_name of a scrollbar work_window.
X_name	The name assigned to the widget upon creation.
X_path	The X_names of all ancestors concatenated with “;”.
X_window	(For *window* class only) For Motif: the window id of the Xt widget. For XView: the window id of the XView widget.

Default Attributes Learned

The following table lists the default attributes learned for each class. (The default attributes apply to all methods of learning: the Test Wizard, the GUI Map Editor, and recording.)

Class	Obligatory Attributes	Optional Attributes	Selector
All buttons	class, label	X_name, class_index	location
list, edit	class, attached_text	X_name, class_index	location
static_text	class, X_name	label, class_index	location
scroll	class, attached_text, orientation	X_attached_name, X_name, class_index	location
window	class, label	X_name, class_index	index
object	class, X_class, X_name	class_index	location
menu_item	class, label	X_name, class_index	location
notebook	class	X_name, class_index	location
spin	class, attached_text	X_name, class_index	location

Working with Motif and Xt Resources

The GUI Spy command in the Tools menu allows you to view all the resources for a specific object, including all Motif and Xt Resources. Any of these resources can also be used as attributes.

You can use the **obj_get_info** TSL function to return the value of any Motif or Xt resource.

For example, if you execute the command

```
obj_get_info ("OK", "labelType", out_val);
```

on the Flights application, (where *labelType* is a Motif resource of type XmNlabelType), XRunner returns the value "STRING".

For more information on using the GUI Spy, see "Viewing GUI Object Attributes", on page 46 in this chapter.

Part III

Creating Tests

7

Creating Tests

Using recording, programming, or a combination of both, you can quickly create automated tests.

This chapter describes:

- Context Sensitive Recording
- Analog Recording
- Checkpoints
- Synchronization Points
- Planning a Test
- Documenting Test Information
- Recording a Test
- Programming a Test
- Editing a Test
- Managing Test Files

About Creating Tests

You can create tests using both recording and programming. Usually, you start by recording a basic *test script*. As you record, each operation you perform generates a statement in Mercury Interactive's Test Script Language (TSL). These statements appear as a test script. You can then enhance your recorded test script, by typing in additional TSL functions and programming elements or by using XRunner's visual programming tool, the Function Generator.

Two modes are available for recording tests:

- *Context Sensitive* records the operations you perform on your application by uniquely identifying Graphical User Interface (GUI) objects.
- *Analog* records keyboard input, mouse clicks, and the precise coordinates traveled by the mouse pointer across the screen.

You can use both modes alternately during the same recording session.

You can further increase the power of your test scripts by adding GUI, bitmap and text checkpoints, as well as synchronization points. Checkpoints allow you to check your application by comparing its current behavior to its behavior in a previous version. Synchronization points solve timing and window location problems that may occur during a test run.

To create a test script, you perform the following main steps:

1. Decide on the functionality you want to test. Determine the checkpoints and synchronization points you need in the test script.
2. Document general information about the test in the Test Header form.
3. Select a Record mode (*Context Sensitive* or *Analog*) and start recording the test on your application.
4. Assign a test name and save the test in the file system.

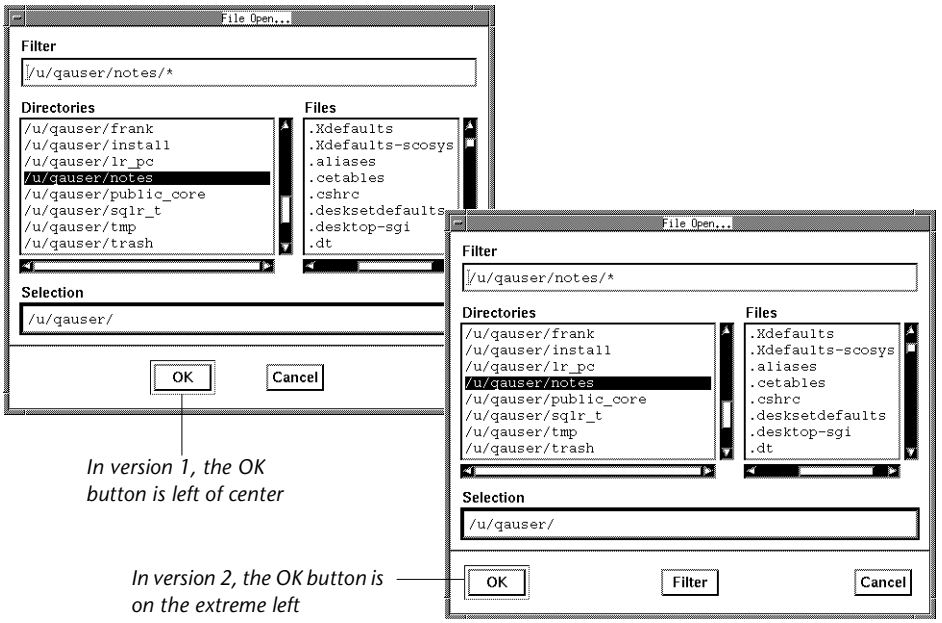
Context Sensitive Recording

Context Sensitive mode records the actual operations you perform on your application in terms of its GUI objects. As you record, XRunner identifies each GUI object you click on (such as a window, button, or list), and the type of operation performed (such as move, press, or select).

For example, if you click the OK button in the Open form, XRunner records the following:

```
button_press ("OK");
```

When it runs the test, XRunner looks for the Open form and the OK button represented in the test script. If, in subsequent runs of the test, the button is in a different location in the Open form, XRunner is still able to find it.



Use Context Sensitive mode to test your application by operating on its user interface. For example, XRunner can perform GUI operations (button clicks, menu or list selections, etc.), and then check the outcome by observing the state of different GUI objects (the state of a checkbox, the contents of a text field, the selected item in a list, etc.).

Remember that Context Sensitive tests work with GUI map files. It is strongly recommended that you read the “Understanding the GUI Map” section of this guide before you start recording.

Analog Recording

Analog mode records keyboard input, mouse clicks, and the exact path traveled by your mouse. For example, if you choose the Open command from a File menu in your application, XRunner records the movements of

the mouse pointer on the screen. When XRunner executes the test, the mouse pointer retraces the coordinates.

In your test script, the menu selection described above might be represented like this:

```
move_locator_track (1);          mouse track
mtype ("<T110><kLeft>-");        left mouse button press
move_locator_track (2);          mouse track
mtype ("<kLeft>+");              left mouse button release
```

Use Analog mode when exact mouse movements are an integral part of the test, such as in a drawing application. Note that you can switch to Analog mode during a Context Sensitive recording session.

Checkpoints

Checkpoints allow you to compare the current behavior of the application under test to its behavior in an earlier version.

You can add three types of checkpoints to your test scripts:

- GUI checkpoints verify the attributes of GUI objects. For instance, you can check that a button is enabled or see which item is selected in a list. See Chapter 8, "Checking GUI Objects" for more information.
- Bitmap checkpoints take a "snapshot" of a window or area of your application and compare this to an image captured in an earlier version.You can add bitmap checkpoints for Context Sensitive or Analog testing. See Chapter 9, "Checking Bitmaps: Context Sensitive Testing" and Chapter 10, "Checking Bitmaps: Analog Testing" for more information.
- Text checkpoints read text from the screen and let you verify its contents. See Chapter 12, "Checking Text" for more information.

Synchronization Points

Synchronization points allow you to solve anticipated timing problems between the test and your application. For example, if you create a test that

opens a database application, you can add a synchronization point that causes the test to wait until the database records are loaded on the screen.

For Analog testing, you can also use a synchronization point to ensure that XRunner repositions a window at a specific location. When you run a test, the mouse cursor travels along exact coordinates. Repositioning the window enables the mouse to reach the correct elements in the window. See Chapter 14, "Synchronizing Test Execution: Analog Testing" for more information.

Planning a Test

Plan a test carefully before you begin recording or programming. The following are some points to consider:

- Determine the functionality you are about to test. It is best to design short, specialized tests that check specific functions of the application, rather than long tests that perform multiple tasks.
- Decide on the types of checkpoints and synchronization points you want to use in the test.
- If you plan to use recording, decide for which parts of your test it would be more appropriate to use the Analog recording mode and for which parts to use the Context Sensitive mode.
- Determine the types of programming elements (such as loops, arrays, and user-defined functions) that you want to add to the recorded test script.

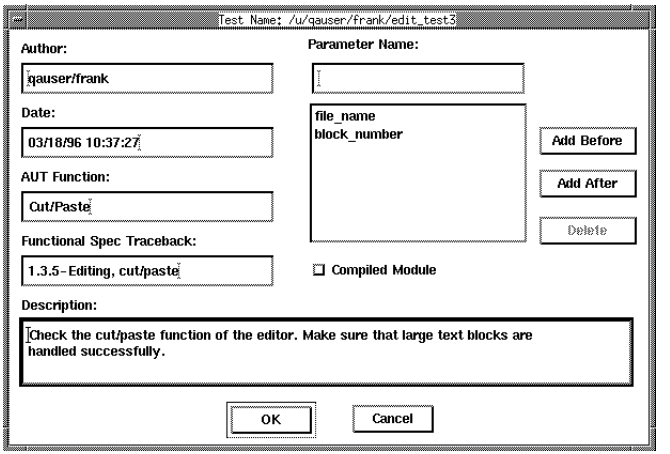
Documenting Test Information

Prior to creating a test, you should document information about the test in the Test Header form. You can enter the name of the test author, the type of functionality tested, a detailed description of the test, and a reference to the relevant functional specifications document.

You can also use the Test Header form to designate a test as a compiled module and to define parameters for a test. For more information, refer to Chapter 21, "Creating Compiled Modules" and Chapter 19, "Calling Tests."

To document test information:

1 Choose the Header command from the File menu. The Test Header form opens.



2 Add information about the test. Note that the Test Header form automatically displays the current date and time.

3 Click OK to save the test information and close the form.

Recording a Test

Consider the following guidelines when recording a test:

- Before you start to record, close all applications not required for the test.
- Create your test so that it "cleans up" after itself. When the test is completed, the environment should be as it was at the beginning of the test. (For example, if the test started with the application window closed, then the test should also close the window and not minimize it to an icon.)
- When recording, use mouse clicks rather than the Tab key to move around within a window in the application under test.
- If you are recording in Analog mode, insert checkpoints using the softkeys rather than the menus or buttons.

➤ If you are recording in Analog mode, avoid typing ahead. For example, when you want to open a window, wait until it is completely redrawn before continuing to work. In addition, avoid holding down a mouse button when this results in a repeated action (for example, using the scroll bar to move the screen display). Doing so can initiate a time-sensitive operation that cannot be precisely recreated. Instead, use discrete, multiple clicks to achieve the same results.

To record a test:

1. From the Create menu, select Record–Context Sensitive or Record–Analog.
2. Perform the test as planned using the keyboard and mouse.

   Insert checkpoints and synchronization points as needed by selecting the appropriate commands from the Create menu: Check GUI, Check Bitmap, Wait Bitmap, or Get Text. (Note that for text verification, XRunner must first learn the font of the AUT).
3. To stop recording, select Stop Recording from the Create menu.

Programming a Test

You can use programming to create an entire test script or to enhance your recorded tests. XRunner contains a visual programming tool, the Function Generator, which provides a quick and error-free way to add TSL functions to your test scripts. To generate a function call, you simply point to an object in your application or select a function from a list. For more information, see Chapter 18, “Using Visual Programming.”

You can also add general purpose programming features to your test scripts such as variables, control-flow statements, arrays, and user-defined functions. You type these elements directly into your test scripts. For more information on creating test scripts with programming, see the “Programming with TSL” section of this guide.

Editing a Test

Use the commands in the Edit menu, or the corresponding buttons in the toolbar to help you make changes to a test script. The following commands are available:

Edit Command	Description
Cut	Deletes the selected text in the test script and places it onto the Clipboard.
Copy	Makes a copy of the selected text and places it onto the Clipboard.
Paste	Pastes the text on the Clipboard at the insertion point.
Delete	Deletes the selected text.
Select All	Selects all the text in the active test window.
Find	Finds the specified character(s) in the active test window.
Go to Line	Moves the insertion point to the specified line in the test script.

Managing Test Files

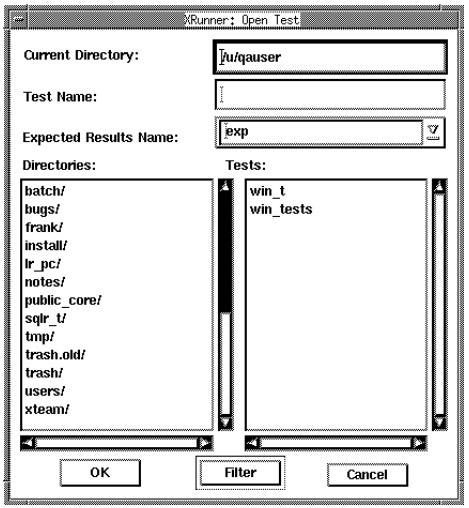
You use the commands in the File menu to create, open, save, and close test files.

Creating a New Test

Choose New from the File menu or click the New button. You are ready to start recording or programming a test script.

Opening an Existing Test

Choose the Open command from the File menu or click the Open button. The Open Test form is displayed.



You can use the Open Test form to load a test from the UNIX file system.

Note: The maximum number of tests you may have open simultaneously is one hundred.

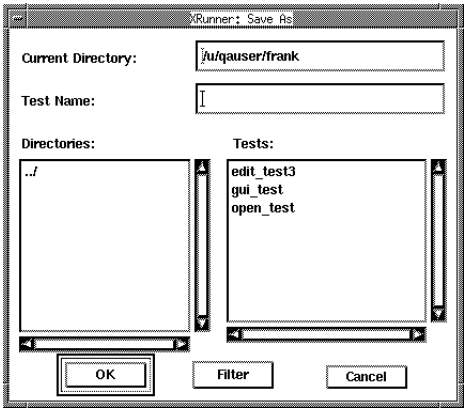
To open a test from the file system:

1. Select File > Open to open the Open Test form.
2. Use the Directories and Tests list boxes to locate your test.
3. In the Test Name field, select the name of the test to open.
4. If the test has more than one set of expected results, select the directory you want to use from the Expected list. The default directory is *exp*.
5. Click OK to open the test.

Saving a Test

The following options are available for saving tests:

- Save changes to a previously saved test by selecting the Save command from the File menu or clicking the Save button.
- Save two or more open tests simultaneously by selecting Save All from the File menu.
- Save a new test script by selecting the Save As command from the File menu or clicking the Save button. The Save As form is displayed.



You can use the Save As form to a save a test in the UNIX file system.

To save a new test in the file system:

1. Open the Save As form.
2. Use the Directories and Tests list boxes to select the location that you want to save the test.
3. In the Test Name field, type a name for the test. Use the standard UNIX naming conventions.
4. Click OK to save the test.

Closing a Test

To close the current test, choose the Close command from the File menu.

8

Checking GUI Objects

By adding GUI checkpoints to your test scripts, you can compare the behavior of GUI objects in two versions of your application.

This chapter describes:

- Checking a Single Object or Window
- Checking Two or More Objects in a Window
- Checking All Objects in a Window
- Modifying GUI Checklists
- Checking Attributes Using check_info Functions
- Default Checks and Custom Checks

About Checking GUI Objects

Use GUI checkpoints in your test scripts to help you examine GUI objects in your application and detect bugs. When you run a test, a GUI checkpoint compares the behavior of GUI objects in the current version of the application under test with the behavior in an earlier version.

For example, suppose you add several new customers to a client database, and you want to make sure that their names appear in a client list. You can create a GUI checkpoint that captures all the names in the list and saves this information as expected results. When you run the test, XRunner compares the current names in the list with the names captured earlier. If any differences are detected, XRunner sends this information to the test report.

To create a GUI checkpoint, you point to GUI objects and select the attributes that you want XRunner to check. You can check the default

attributes recommended by XRunner, or you can define a custom check by selecting different attributes. Information about the GUI objects and the selected attributes is saved in a checklist. XRunner then captures the current attribute values for the GUI objects and saves this information as *expected results*. A GUI checkpoint is automatically inserted into the test script. This checkpoint appears as an **obj_check_gui or a win_check_gui** statement.

When you run the test, XRunner compares the current state of the GUI objects in the application under test to the expected results. If the expected results and the current results do not match, the GUI checkpoint fails. The results of the checkpoint can be viewed in the XRunner Report form. For more information, see Chapter 25, "Analyzing Test Results."

If you are manually programming a test, you can also use the function **check_gui** to check GUI objects. For more information see the *TSL Reference Guide*.

Note that any GUI object you check that is not already in the GUI map is added automatically to the temporary GUI map file. See Chapter 3, "Introducing Context Sensitive Testing" for more information.

Checking a Single Object or Window

You can use a GUI checkpoint to check a single object or window in the application under test.

To create a GUI checkpoint for a single object or window:

1 For checking a window, select Check GUI > Window from the Create menu. For checking an object, select Check GUI > Object.

The mouse pointer turns into a pointing hand.

2 To check the object or window with its default checks, click on it once. If you clicked on a window, a help window asks if you want to check all the objects inside it. Select No to check only the window itself.

To define a custom check, double-click on the object to bring up the appropriate check form.

Select checks and click OK to close the form.

XRunner captures the GUI information and stores it in the test's expected results directory. A GUI checkpoint is inserted in your test script.

For an object, a GUI checkpoint has the following syntax:

obj_check_gui (*object, checklist, expected results, time*);

For a window, the syntax is:

win_check_gui (*object, checklist, expected results, time*);

For example, if you are performing an object check on the Departure Time button in the Flight Reservation window, the resulting statement might be:

```
obj_check_gui ("Departure Time:", list1.ckl, gui1, 1);
```

However, if you are performing a window check on the same window, the resulting statement might be:

```
win_check_gui ("Flight Reservation", list1.ckl, gui1, 1);
```

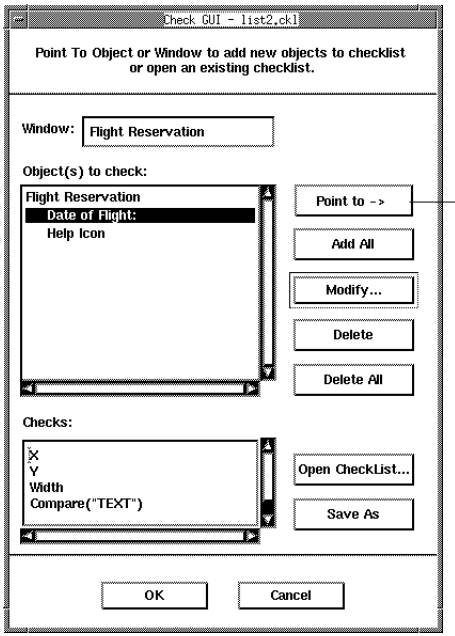
Note that XRunner names the first checklist in the test *list_1.ckl* and the first expected results file *gui_1*. For more information on the **obj_check_gui** and **win_check_gui** functions see the *TSL Reference Guide*.

Checking Two or More Objects in a Window

You can use a GUI checkpoint to check two or more objects in a window. For example, you can create a single checkpoint which checks the state of all the pushbuttons in a certain window of your application.

To create a GUI checkpoint for two or more objects in a window:

1 Select Check GUI > Checklist from the Create menu. The Check GUI form opens.

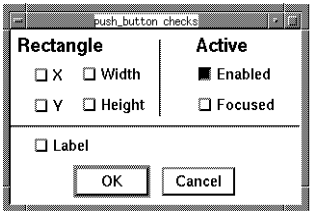


Lets you add a window or object to the checklist

2 Select Window or Object from the Point to options. Click Point To. Select Object or Window from the options list. The mouse pointer turns into a pointing hand.

3 To check an object or window according to its default checks, click on it once. If you are performing a check on a window, a help window asks if you want to check all the objects inside the window. Press No to check only the window itself.

To define a custom check, double-click on the object or window to bring up the appropriate check form.



For example, when you double-click on a pushbutton, this check form is displayed.

Mark the checks you want to perform and press the OK button.

4. The pointing hand remains active. You can continue to select objects by repeating step 3 above for each object you want to check.
5. Press the right mouse button to stop the selection process and to restore the mouse pointer to its original shape.

   The "Objects to check" field contains the names of the window and objects included in the GUI checkpoint.
6. To save the checklist and close the form, click the OK button.

   XRunner captures the GUI information and stores it in the expected results directory of the test. A **win_check_gui** statement is inserted in the test script.

Checking All Objects in a Window

You can create a GUI checkpoint containing all the GUI objects in a window.

To create a GUI checkpoint that checks all GUI objects in a window:

1. Select Check GUI > Checklist from the Create menu. The GUI Checklist form opens.
2. Select Window from the Point to options. The mouse pointer turns into a pointing hand.
3. Click once on the window you want to check. A help window asks whether you want to check all the objects in the window. Select Yes.

4 XRunner generates a new checklist containing all the objects in the window. This may take a few seconds.

5 Note that the Object(s) to check field of the GUI Checklist form now contains all the objects in the window. Select the name of an object. The corresponding object flashes in the application window and the check(s) for the object are displayed in the Checks field.

6 Click OK.

XRunner captures the GUI data, stores it in the test's expected results directory, and enters a **win_check_gui** statement in your test script.

Note: You can also add all objects in a window to the current checklist by clicking the Add All button before you point to the window you wish to capture. XRunner automatically inserts all GUI objects into the checklist according to the default checks for each object class.

Modifying GUI Checklists

In addition to creating and using a GUI checklist while building your test, you can use or make changes to a checklist you created earlier. You can:

- edit a checklist
- use a previously created GUI checklist in a new checkpoint
- make a checklist available to other users

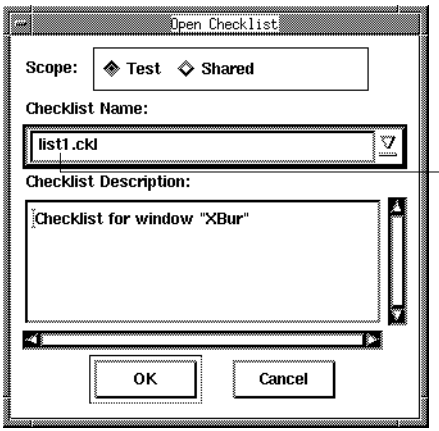
Editing Checklists

You can edit a GUI checklist that you created previously by adding and deleting objects and checks. Note that before you start working, the objects in the checklist must be loaded into the GUI map.

To edit an existing GUI checklist:

1 Select Check GUI > Checklist from the Create menu. The Check GUI form opens.

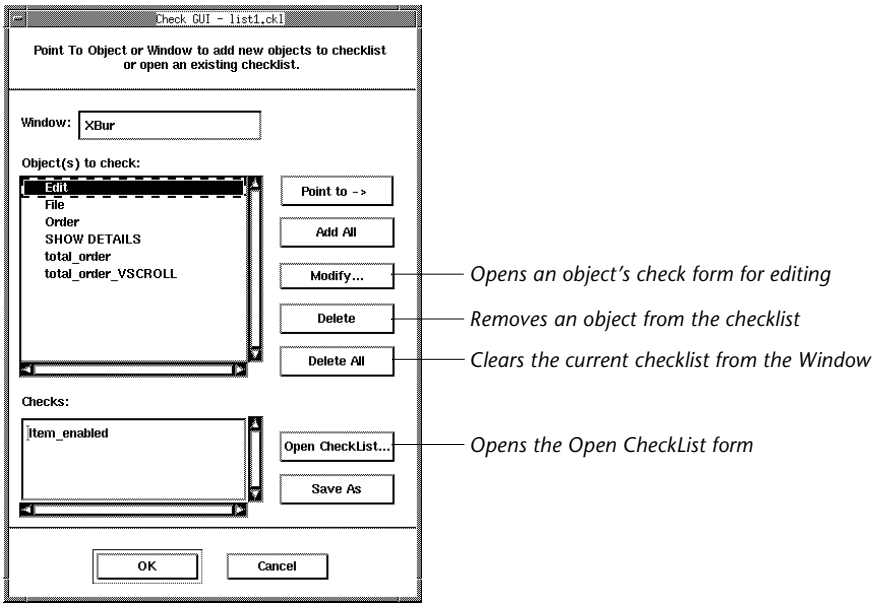
2 Select Open CheckList to display the Open CheckList form. All checklists created for the current test appear in the list box. To see checklists in the Shared directory, select Shared from the Scope options.



3 Select a checklist from the list.

4 Click OK.

The Open CheckList form closes and the selected list appears in the GUI CheckList form.



➤ To make changes to the checks selected for a specific object, select the object in the list box and press Modify.

The appropriate Check form opens. Make the desired changes and click OK to close the form. The changes you made appear in the Checks field.

➤ To remove an object from the checklist, select the object in the list box and press Delete.

➤ To add an object to the checklist, the window that it checks must be open. Click on the pointing hand and then on the object you want to add. When the Check form for the object opens, select the checks you want and click OK to close the form.

5 To save your changes, Click OK in the GUI CheckList form. XRunner overwrites the previous checklist and inserts a checkpoint into the test script.

6 To save the checklist under a new name, click Save As. The Save Checklist form appears.

Recall that if you overwrite an existing checklist, any expected results captured earlier remain unchanged until the test is run. To update the expected results, run the test.

7 Type a name into Checklist Name field. Select the appropriate scope option. Press OK.

Using an Existing GUI Checklist in a Test

The following steps describe how to use an existing GUI checklist in a test. Note that the appropriate GUI map file must be loaded before you run the test. This ensures that XRunner can locate the objects to check in your application.

To use an existing checklist in a test:

1 Select Check GUI > Checklist from the Create menu. The GUI CheckList form opens.

2 Select Open CheckList. The Open Checklist form opens.

3 To see the checklists for the current test, select Test from the Scope options.

4 Display the list of checklists from the Checklist Name field.

5 To see checklists in the Shared directory, select Shared. Select the checklist you want and click OK. The Open Checklist form closes and the selected list appears in the Check GUI form.

6 Open the window in the application under test which contains the objects shown in the checklist (if it is not already open).

7 Click OK. XRunner captures the GUI information and a **GUI checkpoint** is inserted into your test script.

Saving a GUI Checklist in a Shared Directory

By default, checklists are stored in the directory of the current test. You can specify that a checklist be placed in a shared directory to afford wider access. This can be done while creating a GUI checklist or afterwards.

The configuration parameter XR_SHARED_CHECKLIST_DIR in the xrunner.cfg file determines where the shared checklists are stored. For more information, see Chapter 33, “Changing System Defaults.”

To save a GUI checklist in a shared directory:

1. Select Check GUI > Checklist from the Create menu. The Check GUI form opens.
2. In the Check GUI form, create a new GUI checklist. Alternatively, select a checklist from the Open Checklist form and press the OK button.
3. Save the checklist by pressing the Save As button. The Save Checklist form is displayed.

   Click the Shared button in the Scope area. Enter a name for the shared checklist. Click OK to save the checklist and to close the form.

Checking Attributes Using check_info Functions

You can check the attribute values of GUI objects using **check_info** functions. A check function is available for each standard object class (for example, **button_check_info**, **edit_check_info**, **menu_check_info**). You must manually program these functions or generate them using the Function Generator.

For example, **button_check_info** has the following syntax:

button_check_info (*button*, *attribute*, *attribute_value*);

The button parameter is the name of the button. The *attribute* parameter is the attribute you want to check. The *attribute_value* parameter is a variable which stores the current value of the attribute.

In most cases you use a **check_info** function in conjunction with a **get_info** function. The **get_info** function gets the current value of an attribute and the **check_info** function checks that this attribute value is correct.

For example, suppose that in the Flight Reservation application you want to check the value that appears in the "Total" field. This value should be equal to the number in the "Tickets" field multiplied by the price per ticket in the "Price" field. In the test segment shown below, the **edit_get_info** function gets the value in the "Tickets" field and assigns it to the variable "T". This function also gets the value in the "Price" field and assigns it to the value "P". The **edit_check_info** function then checks that the value in the "Total"

field is equal to P multiplied by T. If this value is incorrect, a message is sent to a report.

```
edit_get_info("Tickets","value",T);
edit_get_info("Price","value",P);
if (edit_check_info ("Total", "value", P*T)!=E_OK)
    report_msg("The total is incorrect);
```

For more information, refer to the *TSL Reference Guide*.

Default Checks and Custom Checks

When creating a GUI checkpoint, you determine the types of checks to perform on GUI objects in your application. For each object class, XRunner recommends a default check. For example, if you select a pushbutton, the default check determines whether the pushbutton is enabled. Alternatively, you can choose to define a custom check by selecting attributes from a check form. For example, you can choose to check a pushbutton's width, height, label, and position in a window (x,y coordinates).

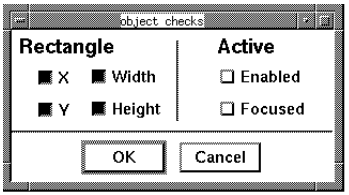
To use the *default check*, select a Check GUI command from the Create menu. Point to a window or object in your application and click on it. XRunner automatically captures default information about the window or object and inserts a GUI checkpoint into the test script.

To create a *custom check* for an object, select a Check GUI command from the Create menu. Point to a window or object and double-click on it. A check form opens in which you select the attributes you want XRunner to check. This form is different for each object class. Select checks and close the form. XRunner captures information about the GUI object and inserts a GUI checkpoint into the test script.

The following sections present the check forms for the different object classes and the types of checks available.

Object Check Form

This form opens when you select a Check GUI command and double-click on a GUI object that belongs to the Mercury object class.



X and Y: Check the x and y coordinates of the top left corner of the GUI object, relative to the window origin (default checks).

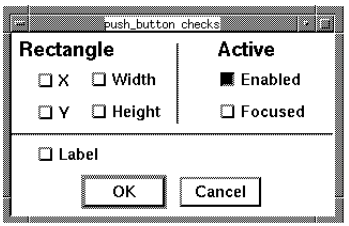
Width and Height: Check the object's width and height, in pixels (default checks).

Enabled: Checks whether the object can be selected.

Focused: Checks whether keyboard input will be directed to this object.

Pushbutton Check Form

This form opens when you select a Check GUI command and double-click on a pushbutton.



X and Y: Check the x and y coordinates of the top left corner of the button, relative to the window origin.

Width and Height: Check the button's width and height, in pixels.

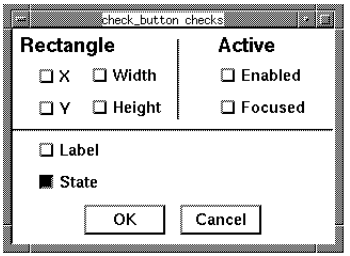
Enabled: Checks whether the button can be selected (default check).

Focused: Checks whether keyboard input will be directed to this button.

Label: Checks the button's label.

Checkbutton Check Form

This form opens when you select a Check GUI command and double-click on a check button.



X and Y: Check the x and y coordinates of the top left corner of the button, relative to the window origin.

Width and Height: Check the button's width and height, in pixels.

Enabled: Checks whether the button can be selected.

Focused: Checks whether keyboard input will be directed to this button.

Label: Checks the button's label.

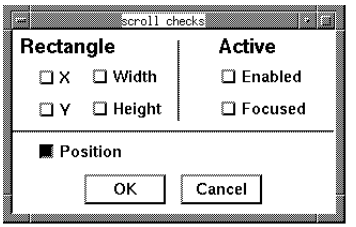
State: Checks the button's state (on or off) (default check).

Radio Button Check Form

This form opens when you select a Check GUI command and double-click on a radio button. The radio button check form identical to the Check Button Check form. See "Checkbutton Check Form."

Scroll Check Form

This form opens when you select a Check GUI command and double-click on a scrollbar object.



X and Y: Check the x and y coordinates of the top left corner of the scrollbar, relative to the window origin.

Width and Height: Check the scrollbar's width and height, in pixels.

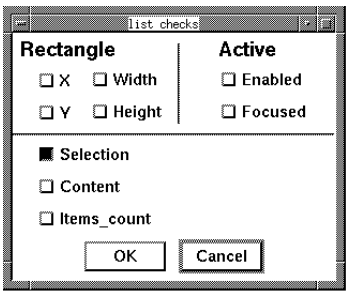
Enabled: Checks whether the scrollbar can be selected.

Focused: Checks whether keyboard input will be directed to this scrollbar.

Position: Checks the current position of the scroll thumb within the scrollbar. (default check).

List Check Form

This form opens when you select a Check GUI command and double-click on a list object.



X and Y: Check the x and y coordinates of the top left corner of the list box, relative to the window origin.

Width and Height: Check the list box's width and height, in pixels.

Enabled: Checks whether an entry in the list can be selected.

Focused: Checks whether keyboard input will be directed to this list box.

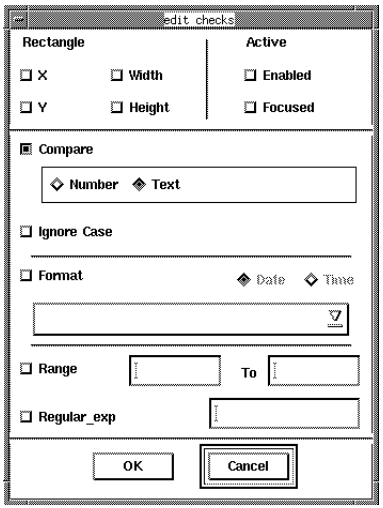
Selection: Checks the current list selection (default check).

Content: Checks the contents of the entire list.

Items count: Checks the number of items in the list.

Edit Check Form

This form opens when you select a Check GUI command and double-click on an edit object.



X and Y: Check the x and y coordinates of the top left corner of the edit object, relative to the window origin.

Width and Height: Check the edit object's width and height, in pixels.

Enabled: Checks whether the edit object can be selected.

Focused: Checks whether keyboard input will be directed to this edit object.

Compare: Checks the contents of the edit field as text (the default) or as numbers.

Ignore case: Indicates that the check will not be case sensitive.

Note: The following checks (Format, Range, and Regular Expression) apply to actual results only. They do not compare actual and expected results.

Format: Checks that the contents of the edit field are in the specified format. To specify the format, check the format type (date or time) and select the required format from the dropdown list.

Date Formats

XRunner supports a wide range of date formats. These are given below, with an example for each.

dd/mm/yy24/03/99

mm/dd/yy03/24/99

dd/mm/yyyy03/24/1999

yy/dd/mm99/24/03

dd.mm.yy03.24.99

dd.mm.yyyy03.24.1999

dd-mm-yy03-24-99

yyyy-mm-dd1999-03-24

Day, Month dd, yyyyTuesday, March 24, 1999

Day dd Month yyyyTuesday 24 March 1999

dd Month yyyy24 March 1999

When the day or month begins with a zero (such as 03 for March), the 0 is not required for a successful format check.

Time Formats

The following time formats are supported by XRunner:

hh.mm.ss10.20.56

hh:mm:ss10:20:56

hh:mm:ss ZZ10:20:56 A.M.

hh:mm: 10:20

Range: Checks that the contents of the edit field are within the specified range. Select Range and specify the lower limit in the left edit field, and the upper limit in the right edit field.

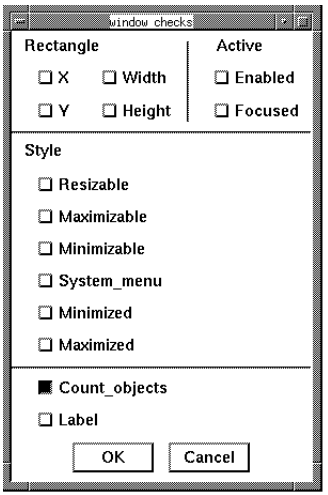
Regular Expression: Checks that the string in the edit field meets the requirements of the regular expression. To define a regular expression, select the Regular_Exp checkbox and type a string in the adjacent field. Note that you do not need to precede the regular expression with an exclamation point. For more information, see Chapter 23, "Using Regular Expressions."

Static Text Check Form

The static text check form is identical to the edit check form. See "Edit Check Form."

Window Check Form

This form opens when you select a Check GUI command and double-click on a window.



X and Y: Check the x,y coordinates of the top left corner of the window.

Width and Height: Check the window's width and height, in pixels.

Enabled: Checks whether the window can be selected.

Focused: Checks whether keyboard input will be directed to this window.

Resizable: Checks whether the window can be resized.

Minimizable or Maximizable: Check whether the window can be minimized or maximized.

System_menu: Checks whether the window has a system menu.

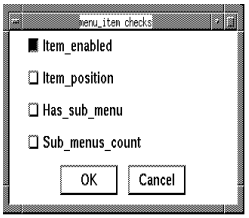
Minimized or Maximized: Check whether the window is minimized or maximized.

Count objects: Counts the number of GUI objects in the window (default check).

Label: Checks the window's label.

Menu Item Check Form

This form opens when you select a Check GUI command and double-click on a menu.



Item enabled: Checks whether the menu is enabled (default check).

Item position: Checks the position of each item in the menu.

Has_sub_menu: Checks whether menu item has a submenu.

Sub_menus_count: Counts the number of items in the submenu.

9

Checking Bitmaps: Context Sensitive Testing

XRunner allows you to compare two versions of an application under test by matching captured bitmaps. This is particularly useful for checking non-GUI areas of your application, such as drawings or graphs.

This chapter explains how to check bitmaps during Context Sensitive testing. You can also check bitmaps when working in Analog mode. For more information, see Chapter 10, "Checking Bitmaps: Analog Testing."

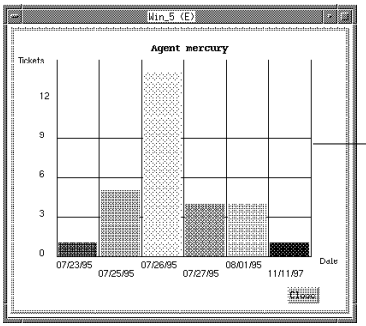
This chapter describes:

- Checking Window and Object Bitmaps
- Checking Area Bitmaps
- Using Data Compression

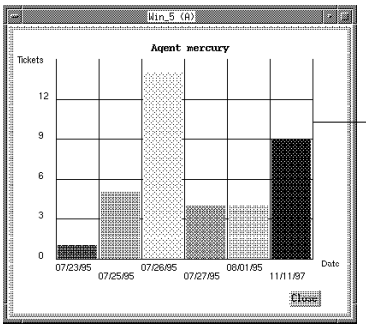
About Checking Bitmaps in Context Sensitive Testing

You can check an object, a window, or an area of a screen in your application as a bitmap. While creating a test, you indicate what you want to check. XRunner captures the specified bitmap, stores it in the expected results directory (*exp*) of the test, and inserts a checkpoint in the test script. When you run the test, XRunner compares the bitmap currently displayed in the application under test with the *expected* bitmap stored earlier. In the event of a mismatch, XRunner captures the current *actual* bitmap and generates a *difference* bitmap. By comparing the three bitmaps (expected, actual, and difference), you can identify the nature of the discrepancy.

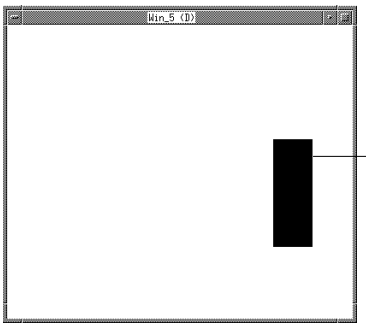
Suppose, for example, your application includes a graph that displays database statistics. You can capture a bitmap of the graph for comparison. If there is a difference between the graph captured for expected results and the one captured during the test run, XRunner generates a bitmap that shows the difference, pixel by pixel.



In the expected graph, captured when the test was created, 1 ticket was sold



In the actual graph, captured during the test run, 9 tickets were sold. This larger quantity is reflected in the column on the extreme right



The difference bitmap shows the discrepancy between the two graphs: in the difference in height of the column on the extreme right

When working in Context Sensitive mode, you can capture a bitmap of a window, object, or of a specified area of a screen. XRunner inserts a

checkpoint in the test script in the form of either a **win_check_bitmap** or an **obj_check_bitmap statement**.

To check a bitmap, you start by selecting Check Bitmap from the Create menu. To capture a window or other GUI object, you click on it with the mouse. For an area bitmap, you mark the area to be checked using a crosshairs pointer.

If the name of a window or object varies each time you run a test, you can define a regular expression in the GUI Map Editor. This instructs XRunner to ignore all or part of the name. For more information on using regular expressions in the GUI Map Editor, see Chapter 5, "Editing the GUI Map."

Note that some platforms and applications work with bitmaps that are not drawn with Xlib calls. For these cases, you can use your own utilities for capturing, verifying, and displaying bitmaps. For information, see Appendix C, "External Utilities for Bitmap Capture/Check/Display."

Checking Window and Object Bitmaps

You can capture a bitmap of any window or object in your application by pointing to it with the mouse pointer. Note that during recording, when you capture an object in a window that is not the active window, XRunner automatically generates a **set_window** statement.

Checking Window Bitmaps

To insert a checkpoint for a window bitmap:

1. Select Check Bitmap/Window from the Create menu. The mouse pointer changes into a pointing hand.
2. Point to the window you want to capture and click the left mouse button. XRunner captures the bitmap and generates a **check_bitmap** statement in the test script.

When you insert a checkpoint for a window bitmap, XRunner generates a **win_check_bitmap** statement with the following syntax:

win_check_bitmap (*object*, *bitmap*, *time***);**

For example, when you click on the Graph window of the Flight Reservation application, the resulting statement might be:

```
win_check_bitmap ( "Graph", "Img_2", 1);
```

For more information on the **win_check_bitmap** function, see the *TSL Reference Guide*.

Checking Object Bitmaps

To insert a checkpoint for an object bitmap:

1. Select Check Bitmap/Object from the Create menu. The mouse pointer changes into a pointing hand.
2. Point to the object you want to capture and click the left mouse button. XRunner captures the bitmap and generates a **check_bitmap** statement in the test script.

   When you insert a checkpoint for an object bitmap, XRunner generates an **obj_check_bitmap** statement with the following syntax:

   obj_check_bitmap (*object, bitmap, time***)**;

   If you click on the Flights button in the Flight Reservation application, the resulting statement might be:

   obj_check_bitmap ("oi FlightListPB", "Img1", 1);

   For more information on the **obj_check_bitmap** function, see the *TSL Reference Guide*.

Note: The execution of the **win_check_bitmap** and **obj_check_bitmap** functions is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Checking Area Bitmaps

You can define any rectangular area of the screen and capture it as a bitmap for comparison. The area can be any size; it can be part of a single window, or it can intersect several windows. The rectangle is identified by the coordinates of its upper left and lower right corners, relative to the upper left corner of the window in which the area is located. If the area intersects several windows or is part of a window with no title (for example, a popup window), its coordinates are relative to the entire screen (the root window).

To capture an area of the screen as a bitmap:

1. Select Check Bitmap > Area from the Create menu. The mouse pointer changes to a crosshairs pointer.
2. Mark the area to be captured: press the left mouse button and drag the mouse until a rectangle encloses the area, then release the mouse button.
3. To preview the area that will be captured, press and hold down the middle mouse button. The coordinates and dimensions of the area are displayed, as is the name of the window from which it is taken.
4. Press the right mouse button to complete the operation. XRunner captures the area and generates a **win_check_bitmap** statement in your script.

Note: The execution of the **win_check_bitmap** function is affected by the current specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_WINDOWS_FRAMES, XR_WM_BORDER and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

The **win_check_bitmap** statement for an area of the screen has the following syntax:

win_check_bitmap (*window*, *bitmap*, *time*, *x*, *y*, *width*, *height* **);**

For example, when you define an area in the Flight Reservation window, the resulting statement might be:

```
win_check_bitmap ("Flight Reservation", "Img2", 1, 25, 25, 75, 60);
```

For more information on the **win_check_bitmap** function, see the *TSL Reference Guide*.

Using Data Compression

XRunner's Data Compression feature allows you to reduce the size of files generated during bitmap capture. To activate Data Compression, open the Controls form and make sure that the Compress checkbox is selected (the default).

As long as compression is activated, all captured bitmaps are compressed. When you display a captured bitmap which is compressed, XRunner automatically restores the uncompressed bitmap. No other operation is required.

While XRunner takes slightly longer to capture and display compressed bitmaps, the file size is significantly reduced. For example, a full-screen capture that ordinarily requires 1 Mb of disk space takes only 30 Kb of disk space in the compressed format.

XRunner uses the standard UNIX compression program; compressed files are designated by the extension .Z in the file name. Such files can be accessed and manipulated from the operating system shell. For example, the UNIX command: `zcat IMG.Z | xwud` pipes an uncompressed version of the bitmap to xwud.

You can also use the operating system to compress bitmaps that were not compressed from within XRunner.

10

Checking Bitmaps: Analog Testing

You can capture all or part of your screen as a bitmap in order to compare different versions of your application.

This chapter explains how to check bitmaps during Analog testing. You can also compare bitmaps when working in Context Sensitive mode. For more information, see Chapter 9, "Checking Bitmaps: Context Sensitive Testing."

This chapter describes:

- Checking Window Bitmaps
- Checking Area Bitmaps
- Checking Windows with Varying Names
- Checking Unnamed Windows

About Checking Bitmaps in Analog Testing

You can check a window or an area of your screen as a bitmap during Analog testing. While creating a test, you indicate the part of the screen that you want to check. XRunner inserts a checkpoint in the script, captures the specified bitmap, and stores it in the expected results directory (*exp*) of the test. When you run a test, XRunner compares the bitmap currently displayed with the *expected* bitmap stored earlier. In the event of a mismatch, XRunner captures the *current* bitmap and generates a *difference* bitmap. By comparing the three bitmaps (expected, current, and difference), you can identify the nature of the discrepancy. This is particularly useful for Analog testing, where the precise movements of the mouse within your application are critical.

XRunner also repositions the current window during a test run if it appears in a different location. Suppose your test involves opening a bitmap in a drawing application and positioning it next to your application. By inserting a bitmap checkpoint, you can ensure that during a test run XRunner checks the window and positions it in the correct location.

When working in Analog mode, you can check either an entire window or an area of the screen. You use a softkey to indicate the window or region to be captured.

When you capture a bitmap for comparison, XRunner generates a **check_window** statement in the test script.

XRunner's Data Compression feature allow you to reduce the size of files generated during bitmap capture. For more information, see Chapter 9, "Checking Bitmaps: Context Sensitive Testing."

Note that some platforms and applications work with images in non-standard formats. For these cases, XRunner allows you to use your own utilities for capturing, verifying, and displaying images. For more information, see Appendix C, "External Utilities for Bitmap Capture/Check/Display."

Checking Window Bitmaps

You can capture a bitmap of any window in your application by selecting it with the mouse pointer.

To capture a window bitmap:

1. Select Record–Analog from the Create menu to start recording.
2. Place the mouse pointer anywhere within the desired window.
3. Press the CHECK WINDOW softkey. The window is captured and a **check_window** statement is generated in your test script. After you hear the second beep, you can continue recording.

   The **check_window** statement for a window bitmap has the following syntax:

   check_window (*time*, *image*, *window*, *width*, *height*, *x*, *y***);**

For example, when you capture the Flight Reservation window, the resulting statement might be:

```
check_window (10, "Win1", "Flight Reservation", 30, 50, 10, 120);
```

For more information on the **check_window** function, see the *TSL Reference Guide*.

Note: The execution of the **check_window** function is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WINDOWS, XR_RAISE_WINDOWS, XR_WM_BORDER and XR_WINDOW_FRAMES configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Checking Area Bitmaps

You can define any rectangular area of the screen and capture it as bitmap for comparison.The rectangular area can be any size; it can be part of a single window or it can intersect several windows. The rectangle is identified by the coordinates of its upper left and lower right corners, relative to the upper left corner of the window in which the area is located. If the area intersects several windows, or is part of a window with no title (a popup menu, for example), its coordinates are relative to the entire screen (the root window).

Note that you cannot capture a selected area when another mouse grab is in effect (for example, when a menu is displayed with a mouse button pressed). In such a case, program the command into the test script.

To capture an area bitmap:

1. Select Record–Analog from the Create menu to start recording.
2. Press the CHECK WINDOW AREA softkey. Recording is temporarily halted and the pointer turns into a crosshairs pointer.

3 Mark the area to be captured: press the left mouse button and drag the mouse until a rectangle encloses the area. Release the mouse button.

4 To preview the area that will be captured, press and hold down the middle mouse button. The coordinates and dimensions of the area are displayed, as is the name of the window from which it is taken.

5 Press the right mouse button to complete the operation. XRunner captures the area and generates a **check_window** statement in your script.

The **check_window** statement for an area bitmap has the following syntax:

check _window (*time*, *image*, *window*, *width*, *height*, *x*, *y*, *relx1*, *rely1*, *relx2*, *rely2***);**

For instance, when you press the CHECK WINDOW AREA softkey and define a region within the Flight Reservation window, the resulting statement might be:

```
check_window  (10, "Img2", "Flight Reservation", 30, 50, 10, 120, 20, 20, 50, 70);
```

For more information on the **check_window** function, see the *TSL Reference Guide*.

Note: The execution of the **check_window** function is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WINDOWS, XR_RAISE_WINDOWS, XR_WM_BORDER and XR_WINDOW_FRAMES configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Checking Windows with Varying Names

If the window you capture has a name that varies from run to run, you can use a regular expression in the **check_window** statement to instruct XRunner to ignore all or part of the window name.

To use a regular expression instead of a window name, the "!" character must precede the expression. For example, the following statement tells XRunner to compare a window whose title begins with the string /tmp/file and is followed by any single uppercase letter:

```
check_window (10, "Win2", "!/tmp/file[A-Z]", ...)
```

When XRunner searches for the window, it returns the first window of the designated size whose title matches the specified regular expression.

If there are several windows of the specified size whose name matches the specified regular expression, XRunner chooses the first window found in the X server hierarchy. This means that you cannot predict which window will be retrieved. When using regular expressions in your test script to describe window names, avoid ambiguity that can lead to incorrect results.

For more information, see Chapter 23, "Using Regular Expressions."

Checking Unnamed Windows

If the window you capture has no title (a popup menu, for example), the window parameter is an empty string, as follows:

check_window (*time*, *image*, "", *width*, *height*, *x*, *y***);**

During test execution, XRunner waits for an unnamed window image with the specified width and height to appear at the x, y coordinates.

If the window captured is not recognized by the server or if an icon is captured, the syntax of the **check_window** statement is:

check_window (*time*, *image***);**

11

Filtering Bitmaps

XRunner lets you use filters to include or exclude regions of a window while you perform bitmap checks.

This chapter describes:

- Creating Filters
- Displaying Filters
- Altering Filter Attributes
- Activating and Deactivating Filters
- Defining Filters with Regular Expressions
- Deleting Filters from the Database

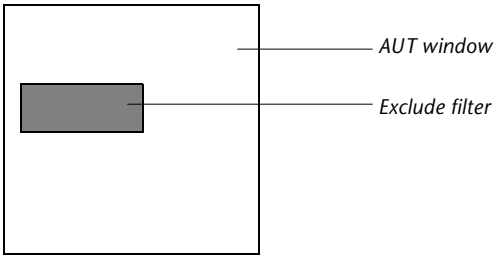
About Filters

XRunner allows you to check bitmaps in you application by comparing the current bitmap with a bitmap captured as the expected results. XRunner allows you check a specific area of the bitmap by setting a filter.

The Area of Interest

Filters allow you to define an area of interest, or the region of the bitmap which will be compared with the expected results during verification. To define the area of interest you can use two types of filters: exclude and include. For example, you can create an exclude filter on a region of a

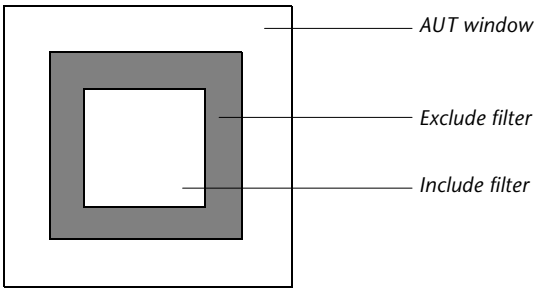
window in which today's date and time are displayed. During the check, this area will be excluded from the comparison.



AUT window and exclude filter

Include and Exclude Filters

An include filter defines a region of a bitmap to be included in a comparison. It can only be used in combination with an exclude filter. (Defining an include filter by itself is meaningless, since by default the entire window is included in the comparison.) For example, in the illustration below the white area is included in comparison and the shaded area is excluded. This is achieved by first defining an exclude filter and then defining a smaller include filter on top of it. The result is a "ring" which is excluded from comparison.



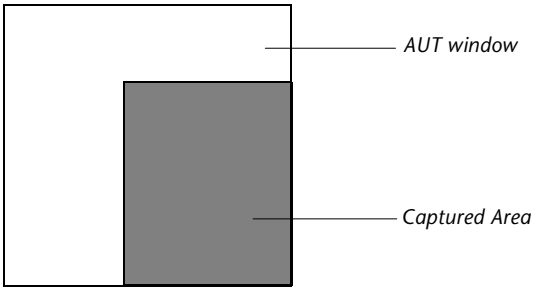
AUT window and exclude filter and include filter

To use a filter in your test, you first create the filter and define its properties. You activate the filter in your test using the **set_filter** function.

Filters versus Partial Bitmaps

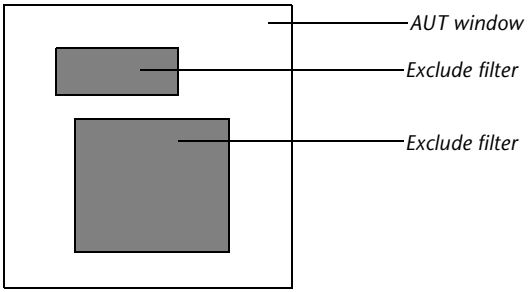
Filters and partial bitmaps share certain features. Both are used to capture and compare part of a bitmap. However, they have different attributes and uses.

A partial bitmap captures only part of a bitmap, thus using less disk space. Only the area defined as a partial bitmap is captured.



AUT window and partial bitmap

When filters are used, on the other hand, an entire bitmap is captured. For example, two exclude filters are defined on one bitmap. During verification, the entire bitmap is captured, but the filtered areas are not compared.



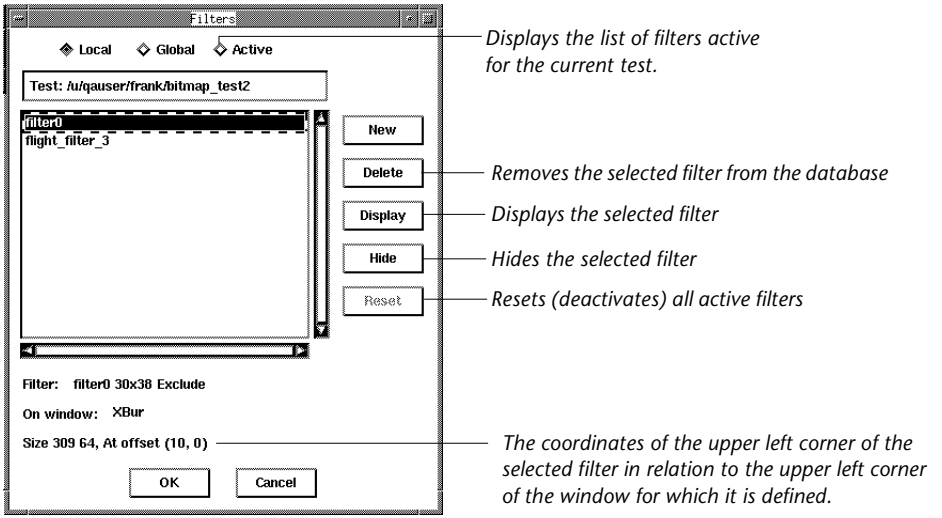
AUT window and two filters

Creating Filters

Before setting a new filter, you define its location by selecting the Local or Global checkbox. You may define the new filter locally (for the current test). In addition, all new filters are stored in the Global Filter Library.

To create a filter:

1 Choose the Filters command from the Tools menu. The Filters form opens.



2 Select the appropriate option button: Local or Global.

- ➤ Choose **Local** to display a list of filters defined for the current test and to add a new filter to the local list.
- ➤ Choose **Global** to display a list of filters stored in the Global Filter Library and to add a new filter to the Global Filter Library. (The Global Filter Library is defined by the XR_GLOB_FILTER_LIB configuration parameter.)

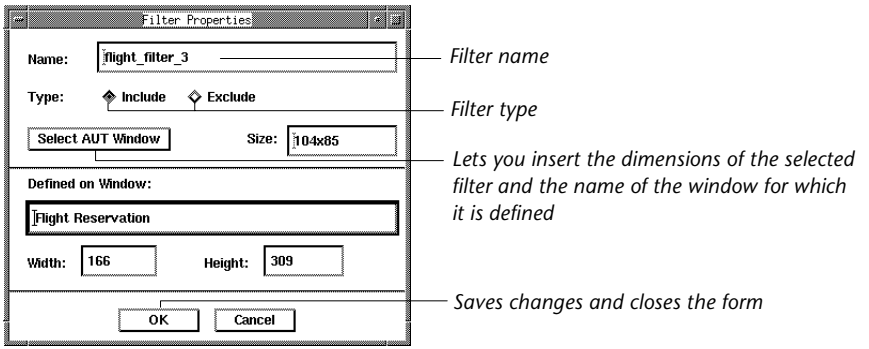
3 Click the New button. The mouse pointer changes to a crosshairs.

4 Place the pointer on the bitmap you want to filter. Press and hold down the left mouse button. Rubber-band the area you wish to filter. Release the mouse button. A filter window is displayed on the screen.

If needed, modify the size of the filter. (Treat the filter as you would any other bitmap.)

To check that the filter covers the desired area, place the pointer inside the filter window and press and hold down the left mouse button. When the button is depressed, the filter bitmap becomes transparent, displaying the area beneath it.

5 To save the new filter, click inside the filter window with the right mouse button. The Filter Properties form is displayed.



If desired, overwrite the filter name currently displayed in the Name field.

6 Select Include or Exclude to specify the filter type.

7 Click Select AUT window. The pointer turns into a pointing hand.

8 Place the pointer on the AUT window and click the left mouse button. The name of the AUT window appears in the Defined on Window field; its dimensions are displayed in the width and height fields.

If you defined a filter that extended beyond the boundaries of the selected AUT window, it is trimmed to the window boundaries.

If the AUT window has a banner that varies from run to run, you can edit the window name in the Defined on Window field to include a regular expression (starting with the "!" character). For more details, see "Defining Filters with Regular Expressions" on page 118.

Note that if you point on the screen (root window), the Defined on Window field remains the default *RootWindow*.

9 Click OK to save the filter properties you assigned and close the form.

Note: In each test you can define a maximum of ten local filters and ten global filters. During a test run, you can set filters on a maximum of fifty different windows. You can also set a maximum of fifty filters on a specific window.

Displaying Filters

You can display a filter you are currently setting or one that has been previously set.

To display a filter:

1. Select either the Local or Global filter list by clicking the appropriate button.
2. Select a filter from the list by clicking its name with the left mouse button.
3. Press Display. You can hide the filter by pressing the Hide button.

Altering Filter Attributes

You can change the size and position of a filter as you would a regular window. You save any modifications in the Filter Properties form.

1. Select a filter from the filters list.
2. Press Display to display the filter.
3. Display the Filters Properties form by clicking with the right mouse pointer inside filter window.
4. You can manipulate the filter window by positioning the mouse pointer on of its borders or corners. Press and hold down the left mouse button. Resize the filter window.
5. Press OK in the Filter Properties form.

Activating and Deactivating Filters

To activate a filter in a test you must enter a **set_filter** statement preceding the **check_window** statement that captures the window with the filter. During test execution, XRunner searches for the specified filter in the Local and Global Filter Library, searches for the specified AUT window and applies the filter to the window.

The **set_filter** statement has the following syntax:

set_filter ("*filter_name*"**,**"*window_name*"**,***width*,*height***);**

The last three parameters are optional. They are used in cases where the name in the window banner changes during a test. Note that in order for XRunner to identify the specified window, all three window parameters must be included.

Let's suppose, for instance, that you create an exclude filter on the Graph window of the Flights Reservation application. The filter is of the Exclude type. It has the dimensions 138 x 117 pixels and is saved locally with the name top_left_filter. To activate the filter in a test, you program the following statement:

```
set_filter ("top_left_filter", "Graph", 138, 117);
```

For further information about the **set_filter** function, refer to the *TSL Reference Guide*.

The order in which filters are activated in the test script determines the actual area of interest. For example, if an exclude filter that fully or partially overlaps an include filter is activated after the include filter, the overlapped region is excluded from the area of interest.

The **reset_filter** statement deactivates a filter that was previously activated. This statement has the syntax:

reset_filter ("*filter_name*" **[,**"*window_name*"**,** *width*, *height***]);**

If the optional window parameters were specified in the **set_filter** statement, then they must also be specified in the **reset_filter** statement.

For further information about the **reset_filter** function, please refer to the *TSL Reference Guide*.

Defining Filters with Regular Expressions

When you enter a regular expression (starting with the exclamation point (!) character) in the Defined on Window field of the Filter Properties form, the filter is used on every window having a banner that matches this regular expression. This feature is useful for assigning filters to windows with names that vary from run to run.

For example, if you enter the expression !/tmp/file[A-Z] in the Defined on Window field, the filter is used on any window which has a banner that starts with the string /tmp/file, and which is followed by any single uppercase letter.

To define a filter with a regular expression, follow steps 1 through 8 described in "Creating Filters" in this chapter. After you click the pointing hand in the AUT window and the window name is entered in the Defined on Window field, edit the window name and enter the desired expression. The expression must start with an exclamation point (!). For more information, see Chapter 23, "Using Regular Expressions."

Note the following points:

- Using several overlapping filters that each use a regular expression, may produce unpredictable results.
- To avoid ambiguity, it is recommended that you use exactly the same regular expression to define filters and the windows for which they are applied.
- When a **check_window** or **wait_window** function is called, XRunner implements filters in the following sequence: First, all filters defined specifically for the designated window (in the order in which they were set); next, all absolute filters (defined on the root window); finally, all filters having regular expression banners that match the banner of the window.

Deleting Filters from the Database

You can delete defined filters from the XRunner database.

To delete a filter:

1. In the Filters form list box, select the name of the filter to be deleted.
2. Click Delete.

12

Checking Text

XRunner allows you to read and check text in any area of your application.

This chapter describes:

- Identifying Application Fonts
- Identifying Fonts Supported by XRunner
- Teaching Fonts to XRunner
- Reading Text
- Searching for Text
- Comparing Text

About Checking Text

You can use text checkpoints in your test scripts to read and check text in GUI objects and in areas of the screen. While creating a test, you mark an area containing text. XRunner reads the text and writes a TSL statement to the test script. You can then add simple programming elements to your test scripts to verify the contents of the text.

You can use a text checkpoint to:

- read text from the screen, using the Get Text menu command and the **get_text** function
- search for text on the screen, using the **find_text** function
- move the mouse pointer to a specified string on screen, using the **move_locator_text** function

- ➤ click on a specified string on screen, using the **click_on_text** function
- ➤ compare two strings, using the **compare_text** function

For example, the following script segment depicts a test in which the AUT searches for the input name (read from an array). When the desired name is found, XRunner reads the contents of the field containing the associated address. Both the input name and address are then printed in an external report file. This search and print operation is repeated until the AUT fails to find an address for the last input name:

```
for i in names{

    # mouse is brought to Search Address command
    move_locator_track(5);
    click ("Left");

    # name is input from the name array
    type (name[i]);

    # input name is searched for
    type ("<kReturn>");

    # acquire contents of address field
    AddressForName = get_text (100, 34, 150, 50);
    printf ("Name : %s, Address : %s \n",
            name[i],
            AddressForName) >> "/u/bart/srch_res.rep";
}

close ("/u/bart/srch_res.rep");
```

In the above example, the address is read from the application window using the **get_text** function. Two additional functions are available: **find_text** performs the reverse operation—searching for a specified string and returning its location on the AUT screen; **compare_text** compares two strings, ignoring any differences specified.

Before you can use XRunner to check text, you must first teach XRunner the fonts used by your application.

Identifying Application Fonts

To identify the names of the fonts used by an application, you can start by checking the application documentation.

If this information is not supplied, you must: (1) identify the font requested by your application using the appropriate protocol analyzer, and (2) identify the font provided by the server using the xfd utility. (This two-step process is required since your application uses the font provided by the server. This is not necessarily the same font initially requested by the application.)

Depending on the system you are using, begin by identifying the font requested by your application.

- The xmon utility is provided with XRunner for Sun-Solaris and HP machines.
- For IBM machines, the xscope utility is provided.

Identifying the Requested Font—Sun/Solaris and HP Machines

If you are using a Sun/Solaris or HP machine, start by using the xmon protocol analyzer, as follows:

1. At the system prompt, enter the command:

   xmonui | xmond &

   The main window of xmonui is displayed.
2. In the Selected Requests region of this window, select "Main" from the Detail options.
3. In the Selected Requests list box, select request 45 OpenFont.
4. Start your application from the command line with the -display option as follows:

   <application_name> -display <machine_name>:1

   For example, if you are testing the textedit application on a machine called venus, type:

   textedit -display *venus:1*

The names of the fonts requested by your application are now displayed in the shell window in which you entered the command from step 1. For example:

```
REQUEST: OpenFont
name: "screen-bold"
```

Identifying the Requested Font—IBM Machines

For IBM applications, you can use xscope or a similar protocol analyzer to determine the fonts used:

1 At the system prompt, enter the command

```
xscope -v1 -i1 > scope_file
```

This connects xscope to the current X server and sets up a simulated X server on display 1. Be sure to set the verbosity level to 1, using the parameter value -v1. The output is redirected to the file scope_file.

2 Connect your application to the display monitored by xscope:

<*application_name*> -display <*machine_name*>:1

For example, if you are testing the textedit application on a machine called venus, type:

```
textedit -display venus:1
```

All the requests and events passed between your application and the X server are written to the file scope_file.

3 Open the file scope_file using a text editor.

4 Search the scope_file file for the string

```
"REQUEST: OpenFont."
```

A few lines below this entry you will find a line with the format:

```
name: "screen-bold"
```

The string is the name of the font requested by your application.

Identifying the Font Provided by the Server

Regardless of the platform you are working on, you must identify the font provided by the server. To do so, use the X Windows xfd utility as follows:

1 At the system prompt, enter the command

xfd –fn <*font_requested_by_application*>

It is recommended that you enclose the name of the font in double quotation marks ("LucidaSans....."), since font names often contain spaces and special characters such as * and & which are otherwise expanded by the shell.

2 The font provided by the server (and which is used by your application) is displayed.

Once you have identified the font used by your application, you may want to verify that XRunner is able to recognize this font. To do so, use the method described below.

Identifying Fonts Supported by XRunner

To check whether the font used by your application is supported by XRunner, you can use the following method:

XRunner provides a test which automatically checks whether a font is supported. This test is located under the pathname $(M_ROOT)/lib/font_test. To invoke this test, call it from an XRunner test that includes the line:

t = call font_test (*font name* **);**

This test creates 300 random two-word pseudo-sentences and then types each sentence in the vi text editor. The test then uses the **get_text** function to access each of the typed sentences and compares each one with the original generated sentence. The test assigns a grade between 0 and 100 to the variable *t*. This grade indicates the percentage of matches during comparison. It is recommended that when testing an application, you use only fonts that have achieved a grade of 99.0 or higher.

This test assumes that you have the file /usr/dict/words on your machine. If you do not have this file, you must generate the random sentences yourself, as explained in the test script.

Note: XRunner does not support recognition of all available X fonts. In particular, XRunner may have difficulty recognizing very small, italic, or oblique fonts. In general, when you display the fonts used by your application using xlsfonts, the fourth field in the displayed standard name indicates an *i* for italicized characters and *o* for oblique characters.

Teaching Fonts to XRunner

A *font group* is a collection of fonts that have been bound together for specific testing purposes. Note that at any time, only one font group may be active in XRunner. In order for any learned font to be recognizable, it must belong to the active font group. However, a learned font can be assigned to multiple font groups.

A *font* is a set of characters used by the application under test. For a given font, each ASCII code corresponds to one character. Note that the size of a font sets it apart from the same character set of a different size.

You teach XRunner fonts in three main steps:

1. Teach XRunner the set of characters (the font) used by the application, using the xrmkfont utility.
2. Create a font group using the xrfontgrp utility, and add the learned font to the group.
3. Use the **setvar** function from within a test script to activate the appropriate font group before using any of the functions for checking text.

   All learned fonts and defined font groups are stored in a *font library*. This library is designated by the XR_GLOB_FONT_LIB parameter in the system configuration file; by default, it is located in the $M_ROOT/fonts subdirectory.

Teaching XRunner Fonts (xrmkfont)

You run the xrmkfont utility for each font XRunner learns. To run this program, enter the following command line at your system level prompt:

```
xrmkfont -add | -del | -list | -string font | -blank [ ["] characters ["] ]
```

You may specify one of four options when running xrmkfont. When none of the options is invoked, the program learns all the uppercase and lowercase letters and digits for the designated font.

- -add appends any *characters* enclosed between optional quotation marks to the file that stores the characters learned for the designated *font*. This can be any font (or its alias) available on your server.

  To view the names of all fonts recognized by your server, enter:

  xlsfonts

  at your system prompt.

  Note that for XRunner to learn special characters, you may need to enter an escape sequence. For example, to learn the exclamation mark character (!), you must precede it with a backslash: \!

  To boost XRunner's ability to read text, keep the symbols you define for a particular font to a minimum.

- -blank defines the size of the specified blank *character* for the designated *font*.
- -del deletes the specified *characters* from the designated font.
- -list displays all characters currently stored for the designated font.
- -string defines the specified *characters* as the default set of characters that are learned the first time that XRunner is taught the font.

  This option can be used, for example, when learning a font with missing characters, such as a font with the following character set: abcdfghklm.

For example, to add the number (#) and dollar sign ($) characters to the previously learned font 9x15, enter the following at the command line:

```
xrmkfont -add 9x15 '#$'
```

To change the size of the blank character used for the previously learned font 9x15 to 12, enter the following command line:

```
xrmkfont -blank 9x5 12
```

The font set data is stored in a file called font_name.mfn. By default, this file is stored in the $(M_ROOT)/fonts subdirectory. To store the .mfn files in a different directory, use the XR_GLOB_FONT_LIB configuration parameter.

Creating a Font Group (xrfontgrp)

After the application font has been learned, you must assign it to a font group, using the xrfontgrp utility. This utility creates a font group that includes all specified fonts. The command syntax for this program is:

xrfontgrp -add | -del | -list *group_name font*$_1$ **[***font*$_2$ *... font*$_n$**]**

When running xrfontgrp, you must specify one of three options:

- -add creates a new font group, or adds new fonts to an already existing group. Note that you must also use this option if you make a change to an existing font already stored in a font group.
- -del deletes fonts from the specified font group.
- -list displays the fonts stored in the specified font group.

A font can be added to a font group only after it has been learned using the xrmkfont utility.

For example, to add a new font 7x13 to the font 9x15 in the font group editor, enter the following command line:

```
xrfontgrp -add editor 9x15 7x13
```

Note: the same font can be assigned to more than one font group. It is necessary to define multiple font groups as xrfontgrp's ability to recognize text degrades as the number of fonts assigned to a group increases. In general, limit the number of fonts contained in a font group to a maximum of five.

The font group data is stored in a file called group_name.grp. By default, this file is stored in the $(M_ROOT)/fonts subdirectory. To store the .grp files in another directory, use the XR_GLOB_FONT_LIB configuration parameter. (Note, however, that .grp files and the .mfn files must be stored in the same directory.)

Note: The file format of fonts is platform-specific. If you are working on several machines operating on different platforms, you must create a separate fonts directory for each machine. For instance, (1) /u/bart/ibm/fonts, (2)/u/bart/sun/fonts, (3) /u/bart/ncr/fonts.

Designating the Active Font Group

The final step before you can perform any of the functions for checking text is to activate the font group that includes the font(s) used by your application.

Only one group can be defined as active at any time. By default, this is the group designated by the XR_FONT_GROUP configuration parameter. However, within a test script you can activate a different font group using the **setvar** function together with the XR_FONT_GROUP configuration parameter.

For example, to activate the font group named editor, add the following statement to your script:

```
setvar ("XR_FONT_GROUP", "editor");
```

You may also specify the active font group

- through the command line interface, using the -fontgrp *group_name* option. For more information, see Chapter 27, “Running Tests from the Command Line.”
- in the Configuration form, using the XR_FONT_GROUP configuration parameter. For more information, see Chapter 33, “Changing System Defaults.”

Reading Text

Once you teach XRunner the necessary fonts, place them in a font group and activate that group, you can read text using the **get_text** function. This function reads a single line of text from a specified area of the screen and assigns it to a variable. The **get_text** function can be generated automatically, using the Get Text menu command, or manually, by programming.

Recording get_text

You can record a **get_text** statement in your test script using the Get Text menu command or the GET TEXT softkey.

To record a get_text statement:

1. Select Get Text from the Create menu or press the GET TEXT softkey. The recording of mouse and keyboard input stops and the mouse pointer becomes a cross hairs.
2. Use the cross hairs to enclose the text to be read within a rectangle. Move the mouse pointer to one corner of the text string. Press and hold down the left mouse button. Then drag the mouse until the rectangle encompasses the entire string and release the mouse button.
3. You can preview the string that is captured by pressing the middle mouse button. The string is displayed directly beneath the text. If there is not enough space there, the string is displayed in the upper left corner of the screen. This step is optional.
4. To capture the string, click the right mouse button.

   XRunner generates a **get_text** statement in the test script and assigns the string located within the defined rectangle to the variable *t*. XRunner also generates a comment (indicated by the # sign) in the test script, detailing the actual text captured.

   The syntax of the get_text statement is:

   t = **get_text** (*x1*, *y1*, *x2*, *y2*);

   For instance, when capturing the Date of Flight: label in the Flights application, the generated statement might read:

t=get_text(74, 165, 169, 182) *#Date of Flight*:

For more information on the **get_text** function, refer to the *TSL Reference Guide*.

Note: If the text in the rectangle is more than one line long, only a single line is read—the line that is the farthest to the left. If two or more lines have the same left margin, the bottom line is read.

Programming get_text

You can also program the **get_text** statement. You can specify the location of the string to be read in either of the following ways:

- variable = get_text(x,y);

  The x and y coordinates define a single pixel on the AUT window. The variable is assigned the value of the string closest to this pixel. (The search radius around the specified point is defined by the XR_TEXT_SEARCH_RADIUS configuration parameter.)

- variable = get_text ();

  When you do not specify a parameter (you leave the parentheses empty), then the string closest to the location of the mouse pointer is read. (The search radius is defined by the XR_TEXT_SEARCH_RADIUS configuration parameter.)

For more information on the **get_text** function, refer to the *TSL Reference Guide*.

Searching for Text

You can use the **find_text** function to search for text on the screen. This function performs the reverse of **get_text**. Whereas **get_text** accesses any text found in the designated area, **find_text** looks for a specified string and returns its location on the screen. This location is expressed as a pair of x,y coordinates that delineate a rectangle.

The **find_text** function must be programmed in the test script, using the following syntax:

find_text (*string***,** *result_array***,** x_1, y_1, x_2, y_2**);**

In the following example, the **find_text** function locates the calculator application. If the string"calculator" is found (the return value of the function is 0), then the application is opened by clicking its icon and a calculation is performed. If the calculator is not found, a message is sent to the test execution report.

```
call init_test();
static coord_array[];
if (find_text("calculator",coord_array,0,0,100,1000))
        report_msg ("Calculator not found");
else   {
    # Calculate x coordinate of the icon.
    x_coord = (coord_array [1] + coord_array [3])/2;
    # Calculate y coordinate of the icon.
    y_coord = (coord_array [2] + coord_array [4])/2;
    # Move locator to calculator icon.
    move_locator_abs (x_coord, y_coord);
    # Open calculator.
    mtype ("<kLeft>-<kLeft>+<kLeft>-<kLeft>+");
    call calculator_test ();
}
```

Note: to boost XRunner's ability to read text, keep the search area defined for **find_text** to a minimum.

For more information on the **find_text** function, refer to the *TSL Reference Guide*.

Moving the Pointer to a String

The **move_locator_text** function searches for the specified string in the indicated area of the screen. The position of the string is specified in terms of the rectangle that encloses it. Once the text is located, the screen pointer is moved to the center of this rectangle.

The **move_locator_text** function has the following syntax:

move_locator_text (string, search_area [, x_shift, y_shift]);

In the following example, **move_locator_text** is used to move the mouse pointer to the appropriate edit fields in a form used to copy files.

```
# open "Copy" form and move pointer 30 pixels left of the center of the rectangle and click
call open_copy_form ();
move_locator_text ("From", 0, 0, 100, 100, 30, 0);
click ("Left");

# type source file name and move pointer 40 pixels left of the center of the rectangle and click
type (source_filename);
move_locator_text ("To", 0, 0, 100, 100, 40, 0);
click ("Left");

# type destination file name
type (dest_filename);

# confirm copying and close form
type ("<kReturn>");
```

For more information on the **move_locator_text** function, see the *TSL Reference Guide*.

Clicking on a Specified Text String

The **click_on_text** function searches for a specified string in the indicated area of the screen, moves the screen pointer to the center of the string, and enters a sequence of mouse button clicks.

The **click_on_text** function has the following syntax:

click_on_text (*string*, *search_area* [,"*click_sequence* "]**)**;

In the following example, **click_on_text** is used to delete a group of files in a file manager application.

```
call open_file_manager();

# select first file to be deleted
click_on_text ("cost_001_dat", 20, 30, 220, 130, "Left");

# select shift and last file to be deleted, to select a group
click_on_text ("cost_007_dat", 20, 30, 220, 130, "Left");

# open the "File" menu
click_on_text ("File", 20, 30, 220, 130, "Left");

# select "Delete" from the menu
click_on_text ("Delete", 20, 30, 220, 130, "Left");

# confirm deletion (click "OK" in the dialog box)
click_on_text ("OK", "Full_screen", "Left");
```

For more information on the **click_on_text** function, see the *TSL Reference Guide*.

Comparing Text

The **compare_text** function compares two strings, ignoring any specified differences. It may be used in conjunction with the **get_text** function, or separately.

The compare_text function has the following syntax:

variable = **compare_text (***str1, str2 [,chars1, chars2]***);**

The **compare_text** function returns 1 when the compared strings are considered the same, and 0 when the strings are considered different. For example, a portion of your test script compares the text string "File" returned by the **get_text** function. Because the lowercase "l" character has the same shape as the uppercase "I", you specify that these two characters be ignored.

```
t = get_text (10, 10, 90, 30);
if (compare_text (t, "File", "l", "l"))
    move_locator_abs (10, 10);
```

13

Synchronizing Test Execution: Context Sensitive Testing

Synchronization compensates for inconsistencies in the performance of your application during a test run. By inserting a synchronization point in your test script, you can instruct XRunner to suspend the test run and wait for a visual cue to be displayed before resuming execution.

This chapter describes:

- Waiting for Window and Object Bitmaps
- Waiting for Area Bitmaps
- Waiting for Attribute Values

About Synchronizing Test Execution

Applications do not always respond to user input at the same speed from one test run to another. A synchronization point in your test script tells XRunner to suspend test execution until the application under test is ready and then to continue the test.

For example, suppose that when testing a drawing application you want to import a bitmap from a second application and then rotate it. A human user knows to wait for the bitmap to be fully redrawn before trying to rotate it. XRunner, however, requires a synchronization point in the test script after the import command and before the rotate command. Each time the test is run, the synchronization point tells it to wait for the import command to be completed before rotating the bitmap.

You can synchronize your test on a window or GUI object in your application, or on any rectangular area of the screen that you select. To create a synchronization point, you select the Wait Bitmap command from the Create menu and indicate a bitmap area or object in the application under test. XRunner inserts a synchronization point in the script, captures an image of the specified bitmap or object, and stores it in the expected results directory (*exp*). The synchronization point appears as a **win_wait_bitmap** or **obj_wait_bitmap** statement in the test script. When you run the test, XRunner suspends test execution and waits for the expected bitmap to appear. It then compares the current *actual* bitmap with the *expected* bitmap captured earlier. When the bitmap appears, test execution resumes.

If you are manually programming a test, you can also use the Analog function **wait_window** to wait for a bitmap. For more information see the *TSL Reference Guide*. Note that some platforms and applications work with bitmaps that are not drawn with Xlib calls. For these cases, XRunner allows you to use your own utilities for capturing, verifying, and displaying bitmaps. For more information, see Appendix C, "External Utilities for Bitmap Capture/Check/Display."

XRunner also synchronizes your test when working in Analog mode. For more information, see Chapter 14, "Synchronizing Test Execution: Analog Testing."

Waiting for Window and Object Bitmaps

You can create synchronization points that wait for window or object bitmaps to appear in the application under test. During a test run, XRunner suspends test execution until the specified bitmap is redrawn, and then compares the current bitmap with the expected one captured earlier. If the bitmaps match, test execution proceeds. In the event of a mismatch, XRunner displays an error message (when the XR_MISMATCH_BREAK configuration parameter is set to on. For more information, see Chapter 33, "Changing System Defaults.")

If the window or object you capture has a name that varies from run to run, you can define a regular expression in the physical description of a window or object in the GUI map. This instructs XRunner to ignore all or part of the

name. For more information, refer to Chapter 5, "Editing the GUI Map" and Chapter 23, "Using Regular Expressions."

During recording, when you capture an object in a window other than the active window, XRunner automatically generates a **set_window** statement.

Waiting for Window Bitmaps

To insert a synchronization point for a window bitmap:

1. Select Wait Bitmap/Window from the Create menu. If you are recording in Analog mode, press the WAIT BITMAP softkey. The mouse pointer changes into a pointing hand.
2. Point to the window you want to capture and click the left mouse button. XRunner captures the bitmap and generates a **wait_bitmap** statement in the test script.

   The **win_wait_bitmap** statement has the following syntax:

   win_wait_bitmap (*object*, *image*, *time*);

   For example, when you click on the Flight Reservation window, the resulting statement might be:

   ```
   win_wait_bitmap ("Flight Reservation", "Img2", 1);
   ```

   For more information on the **win_wait_bitmap** function, see the *TSL Reference Guide*.

Waiting for Object Bitmaps

To insert a synchronization point for an object bitmap:

1. Select Wait Bitmap/Object from the Create menu. The mouse pointer changes into a pointing hand.
2. Point to the object you want to capture and click the left mouse button. XRunner captures the bitmap and generates a **wait_bitmap** statement in the test script.

   The **obj_wait_bitmap** statement has the following syntax:

   obj_wait_bitmap (*object*, *image*, *time*);

If you click on the Flights button in the Flight Reservation window, the resulting statement might be:

obj_wait_bitmap (*"oiFlightsPB", "Img1", 1***);**

For more information on the **win_wait_bitmap** function and **obj_wait_bitmap** function, see the *TSL Reference Guide*.

Note: The execution of the **win_wait_bitmap** and the **obj_wait_bitmap** functions is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WIN and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Waiting for Area Bitmaps

You can use a synchronization point to have XRunner wait for a selected area bitmap in your application. The selected area may be any size; it may be part of a single object or window, or it may intersect several objects or windows.

To select an area bitmap, you use a crosshairs pointer to define a rectangle around the area. XRunner defines the rectangle using the coordinates of its upper left and lower right corners. These coordinates are relative to the upper left corner of the object or window within which the selected area is located. If the selected area intersects several objects within a window, its coordinates are relative to the window. If the selected area intersects several windows, or is part of a window with no title (a popup menu, for example), its coordinates are relative to the entire screen, or the root window.

During a test run, XRunner suspends test execution until the specified bitmap is displayed. It then compares the current bitmap with the expected bitmap. If the bitmaps match, the operation was successful and test execution proceeds. In the event of a mismatch, XRunner displays an error

message (when the XR_MISMATCH_BREAK configuration parameter is set to on). For more information, see Chapter 33, "Changing System Defaults.")

To define a synchronization point for an area bitmap:

1. Select the Wait Bitmap > Area command from the Create menu. If you are recording in Analog mode, press the WAIT WINDOW (AREA) softkey. The mouse pointer changes into a crosshairs pointer.
2. Use the crosshairs pointer to mark the area to be captured. Press the left mouse button and drag the mouse until a rectangle encloses the area. Release the mouse button.
3. You can preview the area that will be captured by pressing the middle mouse button.

   The selected region bitmap coordinates and dimensions are displayed, as is the window name.
4. Press the right mouse button to complete the operation. XRunner captures the bitmap and generates a **win_wait_bitmap or obj_wait_bitmap** statement in your test script.

   The **obj_wait_bitmap** statement for a selected area bitmap has the following syntax:

   obj_wait_bitmap (*object, image, time, x, y, width, height* **);**

   The **win_wait_bitmap** statement for a selected area bitmap has the following syntax:

   win_wait_bitmap (*window, image, time, x, y, width, height* **);**

   For example, suppose you are working with an application that downloads complex bitmaps from your hard disk. You insert a synchronization point in order to wait for a selected area bitmap to appear.

   XRunner generates the following statement:

   ```
   win_wait_bitmap ("Graph", "Img6", 30, 25, 25, 75, 60);
   ```

   For more information on the **win_wait_bitmap** function and **obj_wait_bitmap** function, see the *TSL Reference Guide*.

Note: The execution of the **win_wait_bitmap** and the **obj_wait_bitmap** functions is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WIN and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Waiting for Attribute Values

You can create a synchronization point that instructs XRunner to wait for a specified attribute value to appear in a GUI object. During a test run, XRunner suspends test execution until the attribute value is detected, and then continues the test. For example, you can have XRunner wait for a button to become enabled or for an item to be selected in a list.

To create a synchronization point that waits for an attribute value, add one of the following functions to the test script: **edit_wait_info**, **menu_wait_info**, **win_wait_info**, **list_wait_info**, **static_wait_info**, **obj_wait_info**, **button_wait_info**, **scroll_wait_info**.

For example, **button_wait_info** has the following syntax:

button_wait_info (*button*, *attribute*, *value*, *time* **);**

The *button* parameter is the name of the button. The *attribute* parameter is any attribute that is used by the button object class. The *value* parameter is the attribute value that must appear before the test run can continue. The *time* parameter is the maximum number of seconds XRunner should wait, added to the timeout value defined for the XR_TIMEOUT configuration parameter.

Suppose while testing the Flight Reservation application you order a plane ticket by filling in passenger and flight information and pressing an Insert button. The application takes a few seconds to process the order. Once the operation is completed, you press the Delete button to delete the order.

In order for the test to run smoothly, a **button_wait_info** statement is needed in the test script. This function tells XRunner to wait up to 10 seconds (plus the timeout interval) for the Delete button to become enabled. This ensures that the test does not attempt to delete the test while the application is still processing the order. The following is a segment of the test script:

```
button_press ("Insert");
button_wait_info ("Delete","enabled",0,"10");
button_press ("Delete");
```

14

Synchronizing Test Execution: Analog Testing

Synchronizing your test ensures that during a test run, XRunner checks the position of the window and relocates it if necessary and delays test execution until the window is redrawn.

This chapter explains how to synchronize test execution during Analog testing. You can also synchronize test execution when working in Context Sensitive mode. For more information, see Chapter 13, "Synchronizing Test Execution: Context Sensitive Testing."

Note that some platforms and applications work with bitmaps in non-standard formats. For these cases, XRunner allows you to use your own utilities for capturing, verifying, and displaying bitmaps. For more information, see Appendix C, "External Utilities for Bitmap Capture/Check/Display."

This chapter describes:

- Waiting for Window Bitmaps
- Waiting for Area Bitmaps
- Windows with Varying Names
- Waiting for Windows or Selected Regions to be Redrawn

About Synchronizing Tests in Analog Testing

In the X Windows environment, applications do not always respond at the same speed: the time that elapses until a window is displayed may vary from one execution to another. In addition, windows are likely to open at varying screen locations during each run.

By defining synchronization points in your test script, you instruct XRunner to pause test execution until a given window is completely displayed and repositioned in a specific location. If the window appears within the specified timeout, test execution proceeds.

Suppose while testing a drawing application, you wish to import a bitmap from a second application and, once it is fully reproduced, rotate it. A synchronization point inserted after the import command instructs XRunner to capture the given bitmap and store it in the expected results directory (*exp*) of your test. Then, during test a test run, XRunner locates the window in the same position on the screen and waits for the bitmap to be redrawn completely. XRunner compares the *current* bitmap displayed with the *expected* bitmap stored earlier. If the bitmap is displayed completely within the specified timeout, XRunner sends the rotation command to the application and test execution proceeds.

While working in Analog mode, you can synchronize tests by using the following softkeys:

- The WAIT WINDOW softkey instructs XRunner to wait for a window bitmap to be redrawn.
- The WAIT PARTIAL WINDOW softkey instructs XRunner to wait for a selected area bitmap to be redrawn.

You may want to suspend test execution only long enough for a window bitmap or selected region bitmap to be redrawn—without evaluating its contents. In such instances, you can define synchronization points by using the following softkeys:

- The WAIT REDRAW softkey instructs XRunner to wait for a window to be redrawn; the contents of the window are disregarded.
- The WAIT REDRAW PARTIAL WINDOW softkey instructs XRunner to wait for a partial window to be redrawn; the contents of the window are disregarded.

All four softkeys generate a **wait_window** statement in the test script. The parameters included in the statement vary, according to the softkey used and the bitmap captured.

While working in Context Sensitive mode, you may also synchronize test execution. For more information see "Chapter 13, "Synchronizing Test Execution: Context Sensitive Testing."

Waiting for Window Bitmaps

You can instruct XRunner to suspend test execution in order to wait for a window bitmap to be displayed. You do this by using the WAIT WINDOW softkey to insert a synchronization point in your test script.

To define a synchronization point for a window bitmap:

1. Select Record–Analog from the Create menu to start recording.
2. Select a window in the application under test.
3. Press the WAIT WINDOW softkey. A **wait_window** statement is generated in your test script. After you hear a second beep, you can continue recording.

The **wait_window** statement for a window bitmap has the following syntax:

wait_window (*time*, *image*, *window* [, *width*, *height* , *x*, *y*] **);**

For example, if you place the mouse pointer in the Flight Reservation window of the and press the WAIT WINDOW softkey, the resulting **wait_window** statement might be:

```
wait_window (20, "Win2", "Flight Reservation", 800, 600, 100, 120);
```

For more information on the **wait_window** function, refer to the *TSL Reference Guide*.

Note: The execution of the **wait_window** function is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WINDOWS and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Waiting for Area Bitmaps

You can use a synchronization point to have XRunner wait for a selected area bitmap in your application. You do this by using the WAIT PARTIAL WINDOW softkey to insert a **wait_window** statement in your test script. This can be particularly useful when capturing information such as an icon's title or a menu extending beyond the window border.

The selected area bitmap you capture may be any size; it may be part of a window or intersect several windows. XRunner defines the area using the coordinates of its upper left (*relx1, rely1)* and lower right (*relx2, rely2*) corners. These coordinates are relative to the upper left corner of the window within which the selected area is located. However, if the selected area intersects several windows, or is part of a window with no title (a popup menu, for example), its coordinates are relative to the entire screen.

To define a synchronization point for an area bitmap:

1. Select Record–Analog from the Create menu to start recording.
2. Press the WAIT PARTIAL WINDOW softkey. Recording is temporarily halted and the pointer turns into a crosshairs pointer.
3. Use the crosshairs pointer to enclose the area to be captured: press and hold down the left mouse button and drag the mouse until a rectangle encloses the desired area. Release the mouse button.
4. You may preview the area that will be captured by pressing the middle mouse button. The area bitmap coordinates and dimensions are displayed, as well as the window name.

5 Press the right mouse button to complete the operation. Note that a **wait_window** statement is generated in your test script. The pointer is restored to its position prior to pressing the softkey and recording resumes.

The **wait_window** statement for a selected area bitmap has the following syntax:

wait_window (*time*, *image*, *window*, *width*, *height*, *x*, *y*, *relx1*, *rely1*, *relx2*, *rely2***);**

For instance, when you press the WAIT PARTIAL WINDOW softkey and define an area within the Flight Reservation window, the resulting **wait_window** statement might be:

```
wait_window (1, "Img1", "Flight Reservation", 800, 600, 100, 120, 10, 10, 50, 50);
```

For more information on the **wait_window** function, refer to the *TSL Reference Guide*.

Note: The execution of the **wait_window** function is affected by the current values specified for the XR_RETRY_DELAY, XR_TIMEOUT, XR_MIN_DIFF, XR_MOVE_WINDOWS and XR_RAISE_WINDOWS configuration parameters. For more information on configuration parameters and how to modify them, see Chapter 15, "Enhancing Window Comparison and Synchronization."

Windows with Varying Names

If the window you wait for has a name that varies from run to run, you can use a regular expression in the *window* parameter of the **wait_window** statement to instruct XRunner to ignore all or part of the name. For more information on using regular expressions, see Chapter 23, "Using Regular Expressions".

Waiting for Windows or Selected Regions to be Redrawn

You may want XRunner to suspend test execution only long enough for a window with a specified name and dimensions to be redrawn—without evaluating its contents. For synchronizing your test in this manner, you use the WAIT REDRAW softkey and the WAIT REDRAW PARTIAL WINDOW softkey.

Wait Redraw Softkey

Suppose that while testing your application, you have to display a bitmap which changes with each run. Pressing the WAIT REDRAW softkey instructs XRunner to wait for the window to be redrawn without evaluating the actual object. The **wait_window** statement generated has the following syntax:

```
wait_window (35, "", "Flight Reservation",800, 600, 100, 120);
```

Note that the **image** parameter is left empty—reflecting the fact that XRunner does not store a bitmap; only data related to the window is recorded in the test script. During test execution, XRunner uses this data to identify and position the window to be redrawn before continuing execution but the contents of the window are not evaluated.

Often, the window you want XRunner to wait for is a menu or some other unnamed window (having no banner). Pressing the WAIT REDRAW softkey generates a **wait_window** statement with the following syntax:

```
wait_window (35);
```

In this case only the *time* parameter is assigned a value. During test execution, XRunner waits for a window to be displayed beneath the mouse pointer and to be completely redrawn before continuing execution.

Wait Redraw Partial Window Softkey

The WAIT REDRAW PARTIAL WINDOW softkey instructs XRunner to wait for a area bitmap to be redrawn, without evaluating its contents.

By using this softkey, you temporarily suspend recording. The pointer changes to a crosshairs pointer with which you can define a rectangle

around the desired bitmap. The **wait_window** statement generated has the following syntax.

```
wait_window (35, "", "Flight Reservation", 800, 600, 100, 120);
```

XRunner stores the size and position of the rectangle you define, but does not capture the enclosed bitmap. During test execution, XRunner identifies and repositions the window containing the selected region bitmap to be redrawn, but the bitmap itself is not evaluated.

For details on how to define a area bitmap, refer to the section on the WAIT WINDOW AREA softkey earlier in this chapter.

15

Enhancing Window Comparison and Synchronization

Configuration parameters can be adjusted in order to fine-tune window comparison and synchronization.

This chapter describes:

- How Configuration Parameters Affect Window Functions
- Adjusting the XR_TIMEOUT Parameter
- Setting the Delay

About Adjusting Configuration Parameters

The execution of the **win_check_bitmap**, **obj_check_bitmap**, **check_window** and **wait_window** functions is affected by the values specified for six configuration parameters: XR_TIMEOUT, XR_SCR_REDRAW, XR_RETRY_DELAY, XR_MOVE_WINDOWS, XR_RAISE_WINDOWS, and XR_MIN_DIFF. Some "problematic" windows, such as large windows that are redrawn slowly, require the adjusting of one or more of these variables in order to ensure reliable test execution.

The value of configuration parameters can either be modified using the Configuration form or from within the test script by using a **setvar** statement. For a detailed description of each of these parameters and how they can be set, see Chapter 33, "Changing System Defaults."

The following examples demonstrate how configuration parameters affect window functions. For each example, assume that the test is being executed

in the default Verify mode and that the following values are assigned to XRunner configuration parameters:

Configuration Parameter	Value
XR_TIMEOUT	10 (*seconds*)
XR_RETRY_DELAY	2 (*seconds*)
XR_SCR_REDRAW	5 (*seconds*)
XR_MOVE_WINDOWS	"on"
XR_RAISE_WINDOWS	"on"
XR_MIN_DIFF	15 (*pixels*)

How Configuration Parameters Affect Window Functions

The following example illustrates how various configuration parameters affect the execution of a **check_window** function.

```
check_window (7, "Win_3", "calctool", 400, 300, 120, 180);
```

When this statement is executed, XRunner first sets a time limit of 17 seconds for the **check_window** operation. (This is the value of the *time* parameter (7) plus the value set for the XR_TIMEOUT configuration parameter (10).) It then waits for a window named calctool having a width of 400 and a height of 300 pixels.

Suppose that the specified window takes 3 seconds to come up on the screen. When it appears, XRunner repositions the window so that its upper left corner is located at screen coordinates 120, 180. XRunner then waits 5 seconds (the value set for the XR_SCR_REDRAW configuration parameter) to allow the entire window to be fully redrawn.

At this point, 14 seconds still remain before a window comparison is made. (Note that the 5 second redraw interval is not part of the time remaining to complete the search.)

XRunner now moves on to the evaluation stage of the algorithm: The window is sampled every 2 seconds (the value of the XR_RETRY_DELAY

configuration parameter) and with each sampling, the displayed window bitmap is compared to the bitmap Win_3, stored in the current expected results directory. Up to 15 pixels may differ between the two bitmaps (the value of the XR_MIN_DIFF configuration parameter is 15). If a matching bitmap does not appear within 14 seconds, XRunner captures the current window and stores it as the *actual* bitmap in the current results directory. In addition, the differing pixels are captured and are stored as the difference bitmap.

Adjusting the XR_TIMEOUT Parameter

Drawing applications such as Icon Editor allow you to enlarge bitmaps by issuing a "zoom" command. When you give this command, the program performs a complex and lengthy calculation, then gradually displays the enlarged bitmap. The content of the window changes continuously until the zoom operation is completed. In the test you are creating, you may want execution to wait until the final zoomed bitmap is displayed before entering any further input to the AUT.

In this situation, when recording the test, move the pointer into the window after the zoomed bitmap is displayed and press the WAIT WINDOW softkey. A statement like the following is generated in the TSL script:

```
wait_window (56, "Win_12", "Icon Editor", 800, 600, 100, 100);
```

When the test is played back, this statement will fail if the value recorded for the *time* parameter (56 seconds) is too small to allow the desired bitmap to be fully displayed.

To ensure that the screen has sufficient time to come up, increase the specified value of the XR_TIMEOUT parameter to 100 seconds.

Note that the setting for the XR_TIMEOUT parameters defines the *maximum* interval XRunner waits before it continues to the next statement in the test script; if the expected bitmap is found before the time has lapsed, test execution continues immediately.

Setting the Delay

Suppose that you want to wait only for a window in the Icon Editor application to be redrawn. Because the window is large, you do not want to capture its bitmap; in addition, the contents of the window may change with each run. In this situation, you can use the WAIT REDRAW softkey to produce the following type of statement:

```
wait_window (150, "", "Icon Editor", 900, 700, 125, 116);
```

During the test run, XRunner may sample the window twice and then move on to the next line before the window is redrawn. This occurs if the intervals between consecutive samplings are too short. For example, if the redraw starts only after 20 seconds and XRunner samples the window every 2 seconds, then after the first two samplings XRunner concludes that the window is stable and is completely redrawn.

XRunner allows you to adapt the script to the behavior of a specific window by controlling the XR_RETRY_DELAY configuration parameter. For a window which is redrawn slowly, you can use the **getvar** and **setvar** functions from within the test script to temporarily increase the delay. For example:

```
old_delay = getvar("XR_RETRY_DELAY");

# sample window every 30 seconds
setvar("XR_RETRY_DELAY", 30);
wait_window(150,"Icon Editor", 900, 700, 125, 116);

# revert to previous value
setvar("XR_RETRY_DELAY", old_delay);
```

In the above example, XRunner samples the window every 30 seconds. This is enough time for a change to appear in the window. XRunner therefore concludes that the window is redrawn after only two consecutive bitmaps, 30 seconds apart, are the same.

Note: If the XR_RETRY_DELAY configuration parameter is set to 0, bitmap checking is disabled.

16

Handling Unexpected Events and Errors

You can instruct XRunner to handle unexpected events and errors that occur in your testing environment during a test run.

This chapter describes:

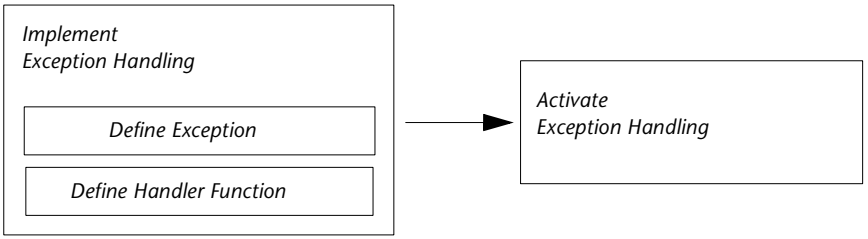
- Handling Popup Exceptions
- Handling TSL Exceptions
- Handling Object Exceptions

About Handling Unexpected Events and Errors

Unexpected events and errors during a test run can disrupt your test and distort test results. This is a problem, particularly when running batch tests unattended: the batch test is suspended until you perform the action needed to recover.

Using *exception handling,* you can instruct XRunner to detect an unexpected event when it occurs and act to recover the test run. For example, you can instruct XRunner to detect a "Printer out of paper" message and call a handler function. The handler function recovers the test run by clicking the OK button to close the message.

To use exception handling, you must implement and activate it.



Implement exception handling: Describe the event or error you want XRunner to detect, and define the actions it will perform in order to resume test execution. To do this, you define the exception and define a handler function.

Activate exception handling: Instruct XRunner to look for any occurrence of the exception you defined.

Normally, you implement and activate exception handling using the Exceptions form. Alternatively, you can program exception handling in your test script using TSL statements. Both methods are described in this chapter.

XRunner enables you to handle the following types of exceptions:

- **Popup exceptions:** Instruct XRunner to detect and to handle the appearance of a specific window.
- **TSL exceptions**: Instruct XRunner to detect and to handle TSL functions that return a specific error code.
- **Object exceptions:** Instruct XRunner to detect and to handle a changed attribute for a specific GUI object.

Handling Popup Exceptions

Test execution is often disrupted by a window that pops up unexpectedly during a test run, such as a message box indicating that the printer is out of paper. Test execution cannot continue until you close the window.

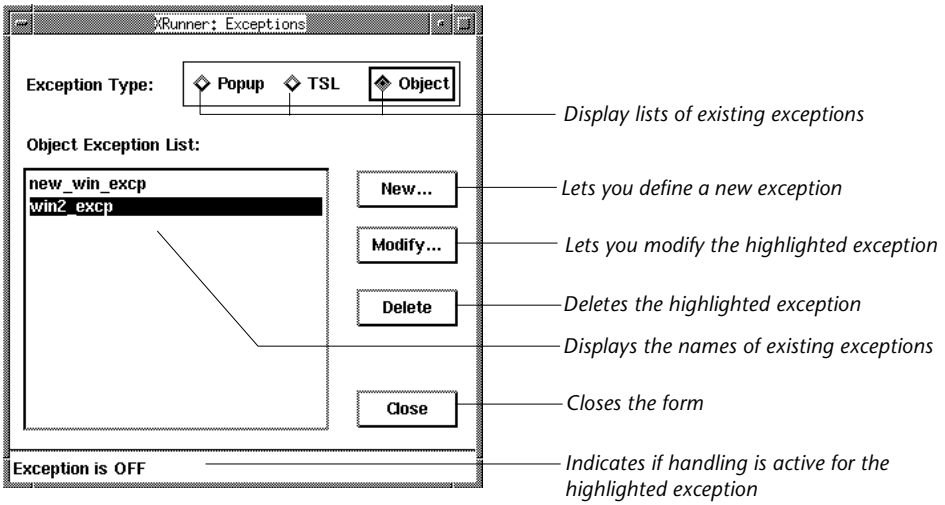
A popup exception instructs XRunner to detect a specific window that may pop up during a test run and to recover test execution, for example by clicking an OK button to close a window.

Defining Popup Exceptions

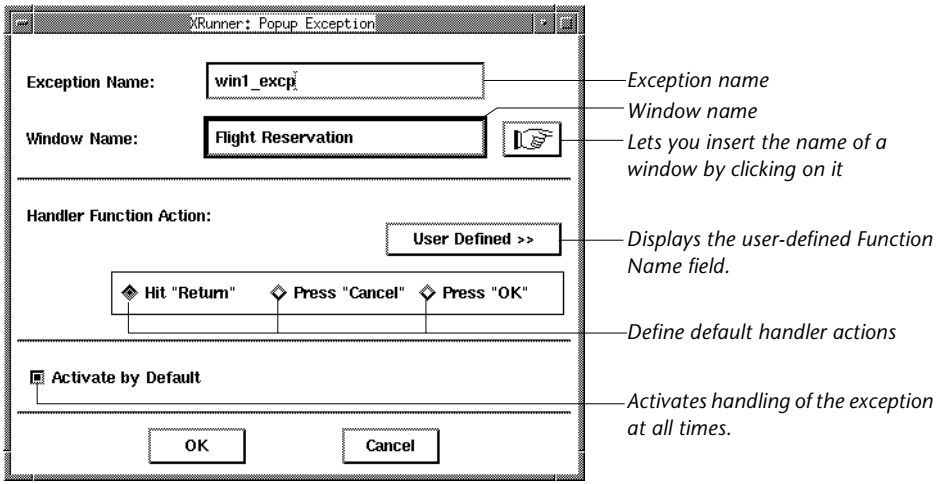
You use the Popup Exception form to define popup exceptions.

To define a popup exception:

1 Select Tools > Exception Handling to open the Exceptions form.



2 Select Popup from the Exception Type list. Click New to open the Popup Exception form.



3 In the Exception Name field, enter a new name.

4 In the Window Name field, enter the name of the popup window.

Type in the name of the window or click the pointing hand and click on the window to insert the name of the window.

You can enter the window's title or its logical name. If the window is not in the GUI map, XRunner assumes that the name you have specified is the window's title. You can also specify a regular expression.

5 Choose a handler function: select one of the defaults (click OK, click Cancel, or hit the Return key), or press the User Defined button to specify a user-defined handler. The form changes to display the user-defined Handler Function Name field.

If you specify a user-defined handler function that is undefined or in an unloaded compiled module, the Handler Function Definition form opens automatically, displaying a handler function template. For more information on defining handler functions, see "Defining Handler Functions for Popup Exceptions" on page 161.

6 To make exception handling active at all times, click the Activate by Default checkbox.

7 Click OK to complete the definition and close the form. The new exception appears in the Exceptions List in the Exceptions form. If the specified window is not in the GUI map, XRunner adds it the map.

XRunner activates handling and adds the new exception to the list of default popup exceptions in the Exceptions form. Default exceptions are defined by the XR_EXCP_POPUP configuration parameter in the .xrunner configuration file.

Instead of using the Popup Exception form, you can program the **define_popup_exception** TSL function in your test script. Note that exceptions you define using TSL are valid only for the current XRunner session. For more information on the **define_popup_exception** function, refer to the *TSL Reference Guide*.

Note: If you are running the X11 NeWS X server, XRunner may take several seconds to detect popup exceptions. To overcome this problem, define an *object exception* for the window, or use a different X server. (The MIXTrap and MIXsun X servers are supplied with XRunner.)

Defining Handler Functions for Popup Exceptions

The handler function is responsible for recovering test execution. When XRunner detects a specific window, it calls the handler function. You implement this function to respond to the unexpected event in a way that meets your specific testing needs.

When defining an exception from the Popup Exception form, you can specify one of two types of handler functions:

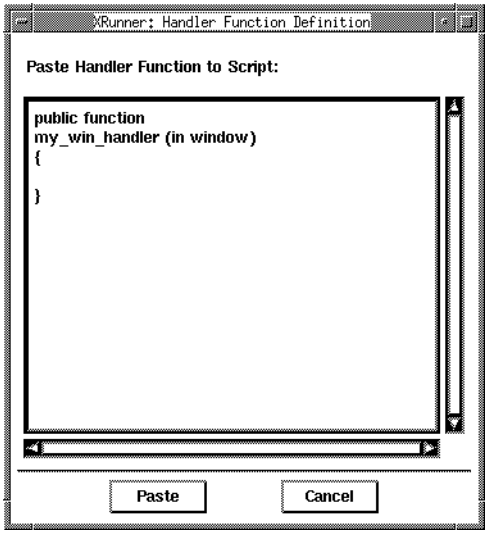
- **Default actions:** XRunner presses the OK or Cancel button in the popup window, or hits the Return key on the keyboard. To select a default handler, click the appropriate button in the form.
- **User-defined handler:** If you prefer, specify the name of your own handler. Click the User Defined button and enter a name in the Function Name field.

  If you specify a user-defined handler which is either undefined or in an unloaded compiled module, XRunner automatically displays a template in

the Handler Function Definition form. You can use the template to help you create a handler function. The handler function must receive the *window_name* as a parameter.

To define your own handler function using the Handler Function Definition form:

1. Define an exception using the Popup Exception form. Specify a new name for the handler function.
2. Press OK. The form closes and the Handler Function Definition form opens, displaying the handler function template.



3. Create a function that closes the window and recovers test execution.
4. Press Paste to paste the statements into the XRunner editor. The Handler Function Definition form closes.
5. You can further edit the test script if necessary. When you are done, save the script in a compiled module.

   User-defined handler functions must be stored in a compiled module. For XRunner to call the function, the compiled module must be loaded when the exception occurs. For more information, see Chapter 21, "Creating Compiled Modules."

In the following example, the handler function is edited to handle an error message. A Flights Reservation application sometimes displays a "FATAL DATABASE ERROR" message, often as the result of a faulty database entry. You can create a handler function that gets the faulty entry number and its value, and writes the information to the test execution report. Then, it dismisses the error message.

The script segment below shows how the handler function (my_win_handler) can be edited in the template:

```
public function my_win_handler(string win_name)
{
auto order_num;
set_window("Open Order",2);
edit_get_text("Order Value",order_num);
report_msg("Database error. Order number:" & order_num);
button_press ("OK);
}
```

Activating Popup Exception Handling

To instruct XRunner to look for any occurrence of the specific window, exception handling must be activated.

Handling of all exceptions defined by the XR_EXCP_POPUP configuration parameter is automatically activated when you start XRunner. To activate exception handling from a test script, use TSL commands, as follows:

➤ To instruct XRunner to begin to detect exceptions, insert the **exception_on** TSL function at the appropriate point in your test script.

➤ To instruct XRunner to stop detecting exceptions, use the **exception_off** function. Use **exception_off_all** to stop the detection of all active exceptions.

For more information on these functions, refer to the *TSL Reference Guide*.

Handling TSL Exceptions

A TSL exception enables you to detect and respond to a specific error code returned during test execution.

Suppose you are running a batch test on an unstable version of your application. If your application crashes, you want XRunner to recover test execution.

A TSL exception can instruct XRunner to detect a crash, which is signaled by an E_APPLICATION_DEAD return value for any function, and to recover test execution by exiting the current test, restarting the application, and continuing with the next test in the batch.

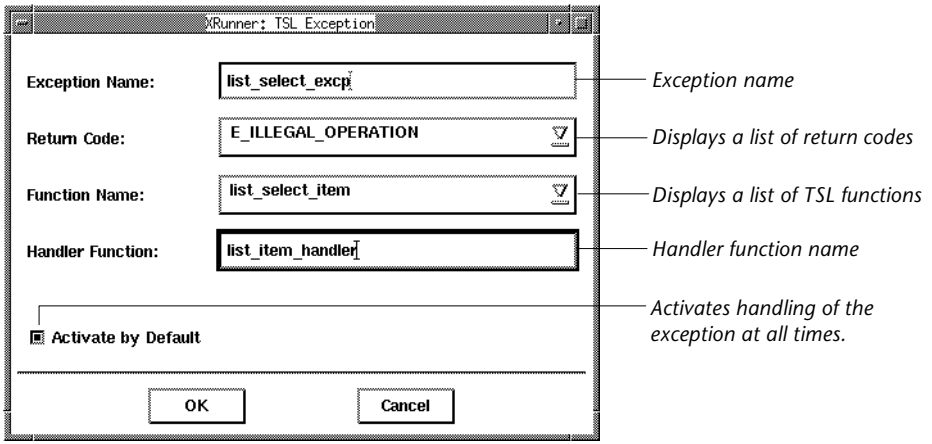
Defining TSL Exceptions

You use the TSL Exception Definition form to define, modify and delete TSL exceptions.

To define a TSL exception:

1 Choose Tools > Exception Handling to open the Exceptions form.

2 Choose TSL from the Exception Type options box and press New to open the TSL Exception form.



3 In the Exception Name field, enter a new name.

4 From the Return Code list, select a return code.

5 Select a TSL function from the Function Name list. If you do not specify a function, or select <<any function>>, XRunner defines the exception for any TSL function that returns the specified return code.

6 In the Handler Function field, enter the name of a handler function.

If you specify a handler function that is undefined or resides in an unloaded compiled module, the Handler Function form opens automatically, displaying a handler function template. For more information on defining handler functions, see "Defining Handler Functions for TSL Exceptions" on page 166.

7 To make exception handling active at all times, click the Activate by Default checkbox.

8 Click OK to complete the definition and close the form. The new exception appears in the Exceptions List in the Exceptions form.

Once you have defined the exception, XRunner activates handling and adds the exception to the list of default object exceptions in the Exceptions form. Default exceptions are defined by the XR_EXCP_TSL configuration parameter in the .xrunner configuration file.

As an alternative to using the TSL Exception form, you can create a TSL exception in a test script using the **define_TSL_exception** function. Note that exceptions you define using TSL are valid only for the current XRunner session. For more information on **define_TSL_exception**, refer to the *TSL Reference Guide*.

Note: For all exception types, XRunner does not handle TSL exceptions that occur within handler functions.

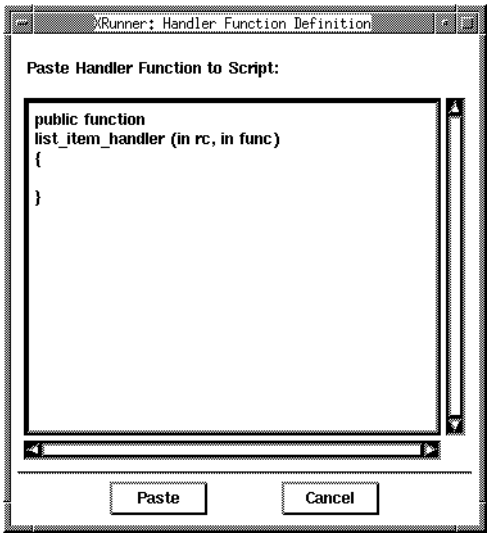
Defining Handler Functions for TSL Exceptions

The handler function is responsible for recovering test execution. When XRunner detects a specific error code, it calls the handler function. You implement this function to respond to the unexpected error in the way that meets your specific testing needs.

If you specify a handler which is either undefined or in an unloaded compiled module, XRunner automatically displays a template in the Handler Function Definition form. You can use the template to help you create a handler function. The handler function must receive the *return_code* and the *function_name* as parameters.

To define a handler function using the Handler Function Definition form:

1. Define an exception using the TSL Exception form. Specify a new name for the handler function.
2. Press OK. The form closes and the Handler Function Definition form opens, displaying the handler function template.



3. Create a function that recovers test execution.
4. Press Paste to paste the statements into the XRunner editor. The Handler Function Definition form closes.

5 You can further edit the test script if necessary. When you are done, save the script in a compiled module.

In order for the exception to call the handler function, the compiled module must be loaded when the exception occurs. For more information, see Chapter 21, "Creating Compiled Modules."

The following example uses the Flight Reservation application to demonstrate how you can instruct XRunner to record a specific event in the test report. In the application, it is illegal to select an item from the "Fly To:" list without first selecting an item from the "Fly From:" list.

Suppose that you program a stress test to create such a situation. The test selects the first item in the "Fly From:" list for every selection from the "Fly To:" list. If the "Fly From:" list is empty, the command:

```
list_select_item ("Fly From:","#0");
```

returns the error code E_ITEM_NOT_FOUND.

You could implement exception handling to identify each occurrence of the E_ITEM_NOT_FOUND return value for the **list_select_item** command. You do this by defining a handler function that reacts by recording the event in the test report.

Edit the handler function (list_item_handler) in the template as follows:

```
public function list_item_handler(rc,func_name)
{
report_msg("List Fly From: is empty");
}
```

Activating TSL Exception Handling

To instruct XRunner to look for any occurrence of the specific window, exception handling must be activated.

Handling of all exceptions defined by the XR_EXCP_TSL configuration parameter is automatically activated when you start XRunner.

In addition, you can activate exception handling from a test script using TSL commands:

➤ To instruct XRunner to begin exception detection, insert the **exception_on** TSL function in the appropriate point in your test script.

➤ To instruct XRunner to stop exception detection, use the **exception_off** function. Use **exception_off_all** to stop detection of all active exceptions.

For more information on these functions, refer to the *TSL Reference Guide*.

Handling Object Exceptions

During testing, unexpected changes can occur to GUI objects in the AUT. These changes are often subtle but they can disrupt test execution and distort results.

One example is a change in the color of a button. Suppose that your application uses a green button to indicate that an electrical network is closed; the same button may turn red when the network is broken.

You could use exception handling to detect a different color in the button during the test run, and to recover test execution by calling a handler function that closes the network and restores the button's color.

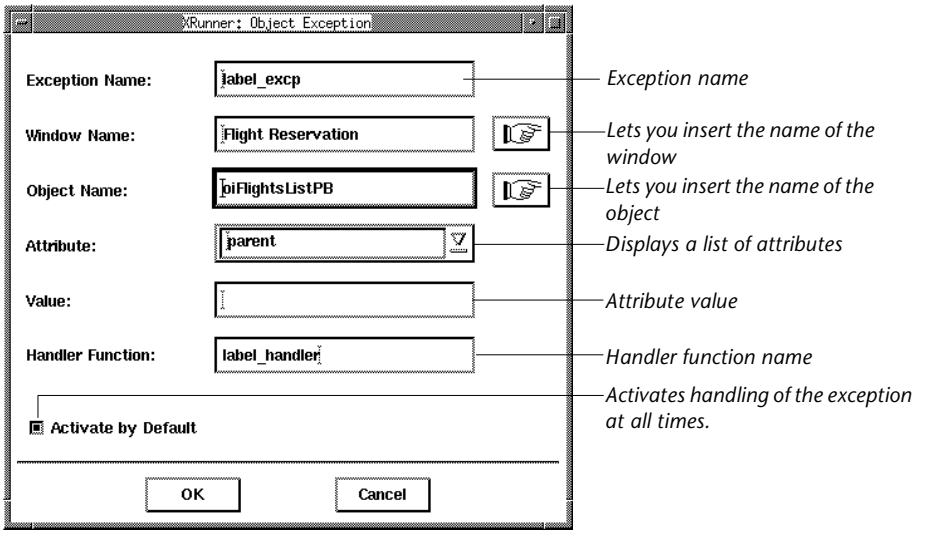
Defining Object Exceptions

You use the Object Exception form to define, modify and delete object exceptions.

To define an object exception:

1 Select Tools > Exception Handling to open the Exceptions form.

2 Select Object from the Exception Type list and click New to open the Object Exception form.



3 In the Exception Name field, type a new name.

4 In the Window Name field, type the name of the window in which the object is found.

5 In the Object Name field, enter the name of the object. Type the name of the window and the object, or press the pointing hand button and click on the object. The names of the object and its parent window appear in the field.

Note that for an object exception, the object and its parent window must be in the loaded GUI map when exception handling is activated.

6 From the Attribute list, select the attribute for which you are defining the object exception. You can also specify any Motif or Xt resource as an attribute.

7 In the Value field, type in a value for the attribute you have selected. If you do not specify a value, the exception will be defined for any change from the current attribute value.

Note that the attribute you specify for the exception cannot appear in the object's physical description. If you attempt to specify such an attribute,

XRunner will display an error message. To work around the problem, modify the object's physical description.

8 Enter the name of the handler function.

If you specify a handler function that is undefined or in an unloaded compiled module, the Handler Function form opens, displaying a handler function template. For more information on defining handler functions, see "Defining Handler Functions for Object Exceptions" on page 170.

9 To make exception handling active at all times, click the Activate by Default checkbox.

If you have not specified a value for the attribute, ensure that the object is displayed when you press the OK button. You can activate exception handling only if XRunner can learn the current value of the attribute.

10 Click OK to complete the definition and close the form. The new exception appears in the Exceptions List in the Exceptions form.

Once you have defined the exception, XRunner activates handling and adds the exception to the list of default object exceptions in the Exceptions form. Default exceptions are defined by the XR_EXCP_OBJ configuration variable in the .xrunner configuration file.

As an alternative to using the Object Exception form, you can create an object exception in a test script using the **define_object_exception** function. Note that exceptions you define using TSL are valid only for the current XRunner session. For more information on **define_object_exception**, refer to the *TSL Reference Guide*.

Defining Handler Functions for Object Exceptions

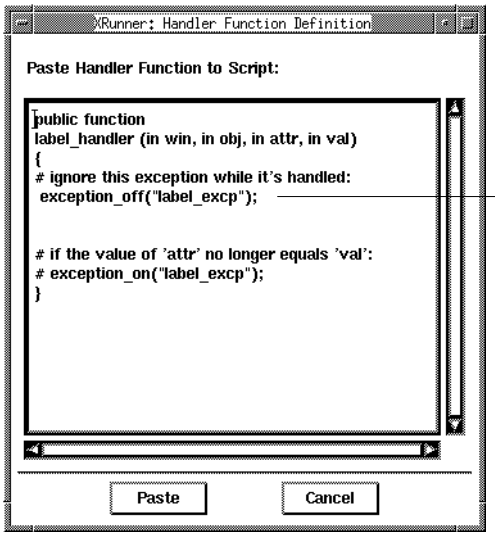
The handler function is responsible for recovering test execution. When XRunner detects a changed attribute, it calls the handler function. You implement this function to respond to the unexpected event in a way that meets your specific testing needs.

If you specify a handler function which is either undefined or in an unloaded compiled module, XRunner automatically displays a template in the Handler Function Definition form. You can use the template to help you create a handler function. The handler function must receive the *window, object, attribute* and *value* as parameters.

Note that the first command in the template is **exception_off.** This is because an object exception does not detect the actual change in the specified object attribute; rather, it detects a changed state in the specified object attribute. The handler function must deactivate exception handling as soon as the function begins to execute. If not, the exception will immediately reoccur, calling the handler function endlessly.

To define a handler function using the Handler Function Definition form:

1. Define an exception using the Object Exception form.
2. Press OK. The form closes and the Handler Function Definition form opens, displaying the handler function template.



3. Create a function that recovers test execution.
4. Press Paste to paste the statements into the XRunner editor. The form closes.
5. You can further edit the test script if necessary. When you are done, save the script in a compiled module. To enable exception detection, ensure that you load the compiled module before test execution.

   Handler functions must be stored in a compiled module. For XRunner to call the handler function, the compiled module must be loaded at the moment the exception occurs. For more information, see Chapter 21, "Creating Compiled Modules."

The labels of GUI objects can become distorted during testing, often as a result of memory management problems. You could, for instance, implement exception handling to take care of such irregularities in the Flights application.

The handler function called might send the unexpected event to a test report, close and restart your application, then exit the current test and continue to the next test in the batch. To do this, you would edit the handler function (label_handler) in the template as follows:

```
public function label_handler(in win, in obj, in attr, in val)
{

#ignore this exception while it is handled:
exception_off("label_except");
report_msg("Label has changed");
menu_select_item ("File;Exit");
system ("flights&");


#if the value of "attr" no longer equals "val":
exception_on("label_except");
texit;
}
```

Activating Object Exception Handling

To instruct XRunner to look for any occurrence of a changed object attribute, exception handling must be activated.

Handling of all exceptions defined by the XR_EXCP_OBJ configuration parameter is automatically activated when you start XRunner.

You can activate exception handling in a test script using TSL commands:

- To instruct XRunner to begin detecting exceptions, insert an **exception_on** statement in the appropriate point in your test script.
- To instruct XRunner to stop detecting exceptions, use the **exception_off** function. Use **exception_off_all** to stop detection of all active exceptions.

  For more information on these functions, refer to the *TSL Reference Guide*.

Part IV

Programming with TSL

17

Enhancing Your Test Scripts with Programming

XRunner test scripts are composed of statements coded in Mercury Interactive's Test Script Language (TSL). This chapter provides a brief introduction to TSL and shows you how to enhance your test scripts using a few simple programming techniques.

This chapter describes:

- Statements
- Comments and White Space
- Constants and Variables
- Performing Calculations
- Creating Stress Conditions
- Decision-making
- Sending Messages to a Report
- Starting Applications from a Test Script
- Defining Test Steps

About Enhancing Your Test Scripts

When you record a test, a test script is generated in Mercury Interactive's Test Script Language (TSL). Each TSL statement in the test script represents keyboard and mouse input to the application under test.

TSL is a C-like programming language designed for creating test scripts. It combines functions developed specifically for test creation with general purpose programming language features such as variables, control-flow statements, arrays, and user-defined functions. TSL is easy to use because you do not have to compile. You simply enhance a recorded test script by typing programming elements into the test window, and immediately execute the test.

TSL includes four types of functions:

- *Context Sensitive* functions perform specific tasks on GUI objects, such as pressing a button or selecting an item from a list. Function names reflect the purpose of the function, for example, **button_press** and **list_select_item**.
- *Analog* functions depict mouse clicks, keyboard input, and the exact coordinates traveled by the mouse.
- *Standard* functions perform general purpose programming tasks, such as sending messages to a report or performing calculations.
- *Customization* functions allow you to adapt XRunner to your testing environment.

XRunner includes a visual programming tool which helps you to quickly and easily add TSL functions to your tests. For more information, see Chapter 18, "Using Visual Programming."

This chapter introduces some basic programming concepts and shows you how to take advantage of a few simple programming techniques in order to create more powerful tests. For more information on TSL, refer to the *TSL Reference Guide*.

Statements

When XRunner records a test, each line it generates in the test script is a statement. A statement is any expression that is followed by a semicolon. A single statement may continue beyond one line in the test script.

For example:

```
if (button_check_state("Underline", OFF) == E_OK)
    report_msg("Underline checkbox is disabled.");
```

If you program a test script by typing directly into the test window, remember to include a semicolon at the end of each statement.

Comments and White Space

When programming, you can add comments and white space to your test scripts to make them easier to read and understand.

Comments

A comment is a line or part of a line in a test script that is preceded by a pound sign (#). When you run a test, the TSL interpreter does not process comments. Use comments to explain sections of a test script in order to improve readability and to make tests easier to update.

For example:

```
# Open the Open Order window in Flight Reservation application
set_window ("Flight Reservation", 17);
menu_select_item ("File;Open Order...");

# Select the reservation for James Brown
set_window ("Open Order", 18);
button_set ("Customer name", ON);
edit_set ("Value_0", "James Brown"); # Type James Brown
button_press ("OK");
```

White Space

White space refers to spaces, tabs, and blank lines in your test script. The TSL interpreter is not sensitive to white space unless it is part of a literal string. Use white space to make the logic of a test script clear.

Constants and Variables

Constants and variables are used in TSL to manipulate data. A constant is a value that never changes. It can be a number, character, or a string. A variable, in contrast, can change its value each time you run a test.

Variable and constant names can include letters, digits, and underscores (_). The first character must be a letter or an underscore. TSL is case sensitive, therefore y and Y are two different characters. Certain words are reserved by TSL and may not be used as names. For a complete list, refer to the *TSL Reference Guide*.

You do not have to declare variables you use outside of function definitions in order to determine their type. If a variable is not declared, XRunner determines its type (auto, static, public, extern) when the test is run.

For example, the following statement uses a variable to store text that appears in an edit field.

```
edit_get_text ("File Name:", text);
        report_msg ("The File Name is " & text);
```

XRunner reads the value that appears in the File Name field and stores it in the *text* variable. A **report_msg** statement is used to display the value of the text variable in a report. For more information, see "Sending Messages to a Report" in this chapter.

Performing Calculations

You can create tests that perform simple calculations using mathematical operators. For example, you can use a multiplication operator to multiply the values displayed in two fields in your application. TSL supports the following mathematical operators:

+	addition
-	subtraction (unary)
-	subtraction (binary)
*	multiplication
/	division
%	modulus
^ or **	exponent
++	increment (adds 1 to its operand - unary operator)
--	decrement (subtracts 1 from its operand - unary operator)

TSL supports five additional types of operators: concatenation, relational, logical, conditional, and assignment. It also includes functions that can perform complex calculations such as **sin** and **exp**. See the *TSL Reference Guide* for more information.

The following example uses the Flight Reservation application. XRunner reads the price of both an economy ticket and a business ticket. It then checks whether the price difference is greater than $100.

```
# Select Economy button
set_window ("Flight Reservations");
button_set ("Economy", ON);

# Get Economy Class ticket price from price field
edit_get_text ("price", economy_price);

# Select Business button
button_set ("Business", ON);

# Get Business Class ticket price from price field
edit_get_text ("price", business_price);

# Check whether price difference is greater than $100
if ((business_price - economy_price) > 100)
tl_step ("Price_check", 1, "Price difference is too large.");
```

Creating Stress Conditions

You can create stress conditions in test scripts that are designed to determine the limits of your application. You create stress conditions by defining a loop which executes a block of statements in the test script a specified number of times. TSL provides three statements that enable looping: *for*, *while*, and *do/while*. Note that you cannot define a constant within a loop.

For Loop

A *for* loop instructs XRunner to execute one or more statements a specified number of times. It has the following syntax:

for (*[expression1]***;** *[expression2]***;** [expression3] **)**
    statement

First, *expression1* is executed. Next, *expression2* is evaluated. If *expression2* is true, *statement* is executed and *expression3* is executed. The cycle is repeated as long as *expression2* remains true. If *expression2* is false, the *for* statement terminates and execution passes to the first statement immediately following.

For example, the *for* loop below selects the file UI_TEST from the File Name list in the Open window. It selects this file five times and then stops.

```
set_window ("Open");
for (i=0; i<5; i++)
    list_select_item ("File Name:_1", "UI_TEST"); # Item Number 2
```

While Loop

A *while* loop executes a block of statements for as long as a specified condition is true. It has the following syntax:

while (*expression* **)**
    statement ;

While *expression* is true, the statement is executed. The loop ends when the expression is false.

For example, the *while* statement below performs the same function as the *for* loop above.

```
set_window ("Open");
i=0;
while (i<5)
    {
    i++;
    list_select_item ("File Name:_1", "UI_TEST"); # Item Number 2
    }
```

Do/While Loop

A *do/while* loop executes a block of statements for as long as a specified condition is true. Unlike the *for* loop and *while* loop, it tests the conditions at the end of the loop rather than at the beginning. A *do/while* loop has the following syntax:

do
statement
while (*expression***);**

The statement is executed and then the *expression* is evaluated. If the expression is true, then the cycle is repeated. If the *expression* is false, the cycle is not repeated.

For example, the *do/while* statement below opens and closes the Order window in Flight Reservation five times.

```
set_window ("Flight Reservation");
i=0;
do
    {
    menu_select_item ("File;Open Order...");
    set_window ("Open Order");
    button_press ("Cancel");
    i++;
    }
while (i<5);
```

Decision-making

You can incorporate decision-making into your test scripts using *if/else* statements or *switch* statements.

If/Else Statement

An *if/else* statement executes a statement if a condition is true, otherwise it executes another statement. It has the following syntax:

```
if ( expression )
    statement1;
else
    statement2;
```

expression is evaluated. If *expression* is true, *statement1* is executed. If *expression1* is false, *statement2* is executed.

For example, the *if/else* statement below checks that the Flights button in the Flight Reservation window is enabled. It then sends the appropriate message to the report.

```
#Enter a value in the Fly From: edit field
set_window ("Flight Reservation");
menu_select_item ("File; New Order");
button_press ("down_arrow_0");
list_select_item ("Fly From:", "Portland");

#Enter a value in the Fly To: edit field
button_press ("down_arrow_1");
list_select_item ("Fly To:", "Denver");

#Check that the Flights button is enabled
button_get_state ("oiFlightsListPB", PB_value);
if (PB_value != ON)
    tl_step ( "Button_enabled", 0, "The Flights button was successfully enabled");
else
    tl_step ( "Button_enabled", 1, "Flights button was not enabled. Check that
values for Fly From and Fly To are valid");
```

Switch Statement

A *switch* statement enables XRunner to make a decision based on an expression that can have more than two values. It has the following syntax:

```
switch (expression)
{
    case case_1:
        statements
    case case_2:
        statements
    case case_n:
        statements
default: statement(s)
}
```

The *switch* statement consecutively evaluates each of the case expressions until one is found that equals the initial expression. If no case is equal to the expression, then the default statement(s) are executed. The default statement(s) are optional.

Note that the first time a case expression is found to be equal to the specified initial expression, no further case expressions are evaluated. However, all subsequent statements enumerated by these cases are executed, unless you use a *break* statement to pass execution to the first statement immediately following the *switch* statement.

The following script reads a configuration file and parses it. The configuration file consists of lines <variable> = <value>. If a line ends with a backslash, the next line is considered as the continuation of this line.

```
while (getline conf_line < conf_file) {

    # Parse the line of the configuration file.
    arg_num = 1;
    for (pos_in_str = 1; pos_in_str < length(conf_line); pos_in_str++)
        switch(cur_char = substr(conf_line, pos_in_str, 1)) {
        case " ":
            break;
        case "=":
            arg_num++;
```

```
            break;
        case "\\":
            if (getline conf_line < conf_file)
                pos_in_str = 0;
            break;
        default:
            arg[arg_num] = arg[arg_num] & cur_char;
    }
    # Verify that configuration file is correct
    switch(arg[1])
    {
    case "ORDER_DIR":
        if (arg[2] != getenv("HOME") & "/new_order_dir")
            report_msg("Error in default definition of ORDER_DIR");
        break;
    case "":
        break;
    default:
        report_msg("Unknown variable is defined: " & arg[1]);
    }
}
```

Sending Messages to a Report

You can define messages in your test script and have XRunner send the messages to the test report. To send a message to a report, add a **report_msg** statement to your test script. The function has the following syntax:

report_msg *(message);*

The *message* can be a string, a variable, or a combination of both.

In the following example, XRunner gets the value of the *label* attribute in the Flight Reservation window and enters a statement in the report containing the message and the label value.

```
win_get_info("Flight Reservation", "label", value);
report_msg("The label of the window is " & value);
```

Starting Applications from a Test Script

You can start an application from within an XRunner test script by using the TSL **system** statement. For example, you can add a **system** statement to a startup test that will automatically open the application under test each time you start XRunner. For more information, see Chapter 34, "Initializing Special Configurations."

The **system** function has the following syntax:

system ("*command [&]*"**);**

The parameter *command* designates the system command to be executed (including command options).

For example, by including the appropriate command line options within the **system** statement, you can specify the exact location at which the AUT window is to be brought up. For example, the TSL statement

```
system ("LD_LIBRARY_PATH=/usr/openwin/lib; export LD_LIBRARY_PATH;
        calctool -Wp 300 400&");
```

defines an environment variable LD_LIBRARY_PATH and invokes the calculator application so that the upper left corner of its window is located at screen coordinates 300, 400.

Note: The **system** statement is interpreted by a Bourne shell and therefore can include only Bourne shell commands.

Defining Test Steps

After you run a test, XRunner displays the overall result of the test (pass/fail) in the Report form. If you want to determine whether sections of a test pass or fail, add **tl_step** statements to the test script.

The **tl_step** function has the following syntax:

tl_step (*step_name, status, description***);**

The *step_name* parameter is the name of the test step. The *status* parameter determines whether the step passed (0), or failed (any value except 0). The *description* parameter describes the step.

For example, in the following test script segment, XRunner enters a two-letter password in the Flights Reservation application login window. The **tl_step** function is used to determine whether the application processes the illegal password.

```
set_window("Login");
edit_set("Agent_name", "Andy");
edit_set("Password", "mc");
button_press("OK");
if (!win_exists("Flights Reservation Message", 1));
    tl_step("Password_check", 0, Application successfully processed password
of fewer than3 characters.");
else
    tl_step("Password_check", 1, "Application failed to process password of
fewer than 3characters-expected message did not appear.");
```

When the test run is completed, you can view the test results in the XRunner Report form. The report displays a result (pass/fail) for each step you defined with **tl_step**.

18

Using Visual Programming

Visual programming helps you add TSL statements to your test scripts quickly and easily.

This chapter describes:

- Generating a Function for a GUI Object
- Selecting a Function from a List
- Assigning Argument Values
- Modifying the Default Function in a Category

About Visual Programming

When you record a test, XRunner generates TSL statements in a test script each time you click on a GUI object or use the keyboard. In addition to the recordable functions, TSL includes many functions that can increase the power and flexibility of your tests. You can easily add functions to your test scripts using XRunner's visual programming tool, the Function Generator.

The Function Generator provides a quick, error-free way to program scripts. You can:

- Add Context Sensitive functions that perform operations on a GUI object or get information from the application under test.
- Add Standard and Analog functions that perform non-Context Sensitive tasks such as synchronizing test execution or sending user-defined messages to a report.
- Add Customization functions that allow you to modify XRunner to suit your testing environment.

You can add TSL statements to your test scripts using the Function Generator in two ways: by pointing to a GUI object, or by selecting a function from a list. When you select the Insert Function command and point to a GUI object, XRunner suggests an appropriate Context Sensitive function and assigns values to its arguments. You can accept this suggestion, modify the argument values, or select a different function altogether.

By default, XRunner suggests a "get" function. This is a function that provides information about the object. For example, if you select Insert Function > Object from the Create menu and point to an OK button, XRunner opens the Function Generator form and generates the statement:

```
button_get_info ("OK", "enabled", value);
```

This statement examines the OK button and gets the current value of the enabled attribute. The *value* can be 1 (enabled), or 0 (disabled).

Once you have generated a statement, you can use it in two different ways, separately or in combination:

- *Paste* the statement into your test script. When required, a **set_window** statement is inserted automatically into the script before the generated statement.
- *Execute* the statement from the Function Generator.

  Note that if you point to an object that is not in the GUI map, the object is added automatically to the temporary GUI map file when the generated statement is executed or pasted into the test script.

Generating a Function for a GUI Object

With the Function Generator, you can generate a Context Sensitive function simply by pointing to a GUI object in your application. XRunner examines the object, determines its class, and suggests an appropriate function. You can accept this default function or select another function from a list.

Using the Default Function for a Window or Object

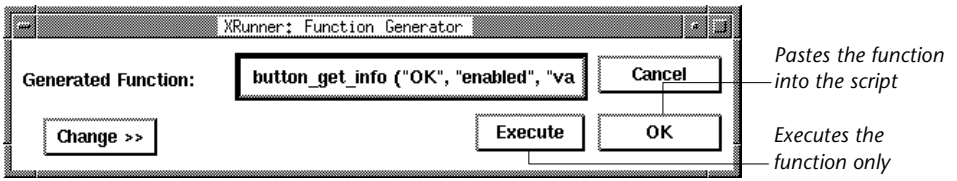
When you generate a function by pointing to a window or an object in your application, XRunner determines the class of the window or object and suggests a function. For most classes, the default function is a "get" function. For example, if you click on a list, XRunner suggests the **list_get_selected** function.

To use the default function for a window or object:

1. Select Create > Insert Function > Window or > Object. The mouse pointer changes into a pointing hand.
2. Point to a window or object in the application under test.
3. Click on an object with the left mouse button. The selected object or window is highlighted. The Function Generator form opens and presents the default function for the selected object. XRunner automatically assigns argument values to the function.

   To cancel the operation without selecting an object, press the right mouse button.
4. To *paste* the statement into the test script, click OK. The function is pasted into the test script at the insertion point and the Function Generator form closes.

   To *execute* the function, click Execute. The function is executed but is not pasted into the test script.



5 Click the Close button to close the form.

Selecting a Different Function for a GUI Object

If you do not want to use the default function suggested by XRunner, you can select a different function from a list.

To select a different function for a GUI object:

1 Select Create > Insert Function > Window or > Object. The mouse pointer changes into a pointing hand.

2 Point to a window or object in the application under test.

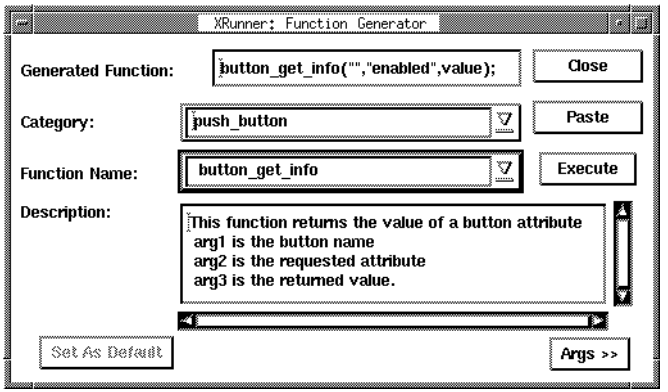
3 Click on an object with the left mouse button. The selected object or window is highlighted. The Function Generator form opens and presents the default function for the selected object. XRunner automatically assigns argument values to the function.

To cancel the operation without selecting an object, press the right mouse button.

4 In the Function Generator form, click the Change button. The form expands and displays a list of functions. The list includes only functions which can be used on the GUI object you selected. For example, if you select a list object, the Function Name list displays **list_activate_item**, **list_close**, **list_deselect_range** etc.

5 Select a function from the Function Name list. The generated statement is displayed at the top of the form. Note that XRunner automatically fills in

argument values. A description of the function appears at the bottom of the form.



6 If you want to modify the argument values, press the Args button. The form expands and displays a field for each argument. See "Assigning Argument Values" in this chapter to learn how to fill in the argument fields.

7 To *paste* the statement into the test script, click Paste. The function is pasted into the test script at the insertion point.

To *execute* the function, click Execute. The function is immediately executed but is not pasted into the test script.

8 You can continue to generate function statements for the same object by repeating the steps above without closing the form. The object you selected remains the active object and arguments are filled in automatically for any function selected.

9 Click Close to close the form.

Selecting a Function from a List

When programming a test, you may want the test to perform a particular task, but you do not know the exact function to use. The Function Generator helps you to quickly and easily locate the function you need and to insert the function into your test script. Functions are organized by category; you select the appropriate category and the function you need

from a list. A description of the function is displayed along with its parameters.

To select a function from a list:

1. Select Insert Function > From List from the Create menu to open the Function Generator form.
2. In the Category list, select a function category. For example, if you want to view menu functions, select menu. If you do not know which category you need, use the default *all_functions*.
3. Select a function from the Function Name list. If you select a category, only the functions that belong to the category are displayed in the list. The generated statement is displayed at the top of the form. Note that XRunner automatically fills in the default argument values. A description of the function appears at the bottom of the form.
4. To define or modify the argument values, press the Args button. The form expands and displays a field for each argument. See "Assigning Argument Values" in this chapter to learn how to fill in the argument fields.
5. To *paste* the statement into the test script, click Paste. The function is pasted into the test script at the insertion point.

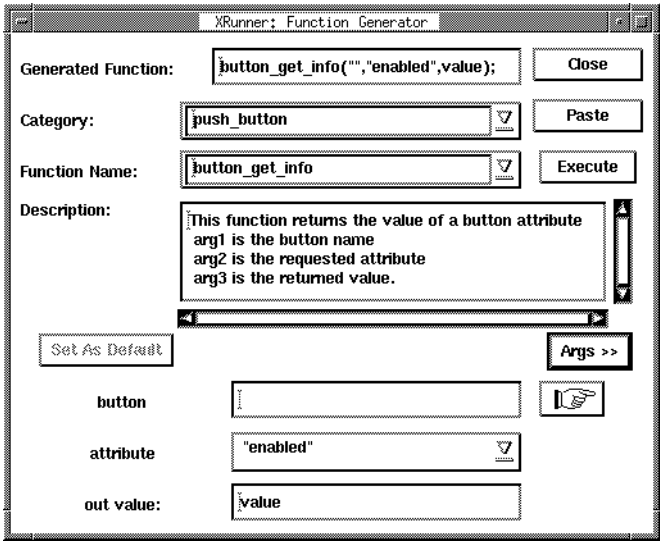
   To *execute* the function, click Execute. The function is immediately executed but is not pasted into the test script.
6. You can continue to generate additional function statements by repeating the steps above without closing the form.
7. Click Close to close the form.

Assigning Argument Values

When you generate a function using the Function Generator, XRunner automatically assigns values to the function's arguments. If you generate a function by clicking on a GUI object, XRunner evaluates the object and assigns the appropriate argument values. If you select a function from a list, XRunner fills in default values when possible, and you fill in the rest.

To assign or modify argument values for a generated function:

1. In the Function Generator form, select a category and a function name.
2. Click the Args button. The form expands according to the number of arguments in the function.



3. Assign values to the arguments. You can assign a value either manually or automatically.

   To *manually* assign values, type a value directly in an argument field. For some fields, you can choose a value from a dropdown list.

   To *automatically* assign values, click the pointing hand button and then click on an object in your application. The appropriate values appear in the argument fields.

   Note that if you click on an object that is not compatible with the selected function, a message states “Current function is invalid for the selected object. Change Category?” Click Yes to change to the correct category. Click No to cancel the operation.

Modifying the Default Function in a Category

In the Function Generator, each function category has a default function. When you generate a function by clicking on an object in your application, XRunner determines the appropriate category for the object and suggests the default function. In most Context Sensitive function categories this is a "get" function. For example, if you click on an edit field, the default function is **edit_get_info**. For Analog, Standard and Customization function categories, the default is the most commonly used function in the category. For example, the default function for the window category is **win_check_gui**.

If you find that you frequently use a function other than the default for the category, you can make it the default function.

To change the default function in a category:

1. Open the Function Generator form and select a function category.
2. In the Function Name list, select the function that you want to make the default.
3. Click the Set as Default button.
4. Click Close to close the form.

The selected function remains the default function in its category until it is changed or until XRunner is exited. To preserve changes to the default function setting, add a **generator_set_default_function** command to your startup test. For more information on startup tests, see Chapter 34, "Initializing Special Configurations."

The **generator_set_default_func** function has the following syntax:

generator_set_default_function *(category_name, function_name)**;***

For example:

```
generator_set_default_function ("push_button", "button_press");
```

sets the **button_press** function as the default for the push_button category.

19

Calling Tests

The tests you create with XRunner can call, or be called by, any other test. When XRunner calls a test, parameter values can be passed from the calling test to the called test.

This chapter describes:

- Using the Call Statement
- Returning to the Calling Test
- Setting the Search Path
- Defining Test Parameters
- Calling the check_file Test

About Calling Tests

By adding **call** statements to test scripts, you can create a modular test tree structure containing an entire test suite. A modular test tree consists of a main test that calls other tests, passes parameter values, and controls test execution.

When XRunner interprets a call statement in a test script, it opens and runs the called test. Parameter values may be passed to this test from the calling test. When the called test is completed, XRunner returns to the calling test and continues the test run. Note that a called test may also call other tests.

By adding decision-making statements to the test script and return values, you can use a main test to determine the conditions that enable a called test to run.

For example:

```
rc= call login ("John", Mercury);
if (rc == E_OK)
    {
    call insert_order();
    }
else
    }
    tl_step ("Call Login", 1, "Login test failed";
    call open_order ();
}
```

This test calls the login test. If login is executed successfully, XRunner calls the insert_order test. If the login test fails, the open_order test is executed.

You commonly use **call** statements in a batch test. A batch test allows you to call a group of tests and run them unattended. It suppresses all messages that are normally displayed during execution, such as a message reporting a bitmap mismatch. For more information, see Chapter 26, "Running Batch Tests."

Note: An XRunner call chain can contain a maximum of 100 called tests.

Using the Call Statement

A test is invoked from within another test by means of a **call** statement. This statement has the following syntax:

call *test_name* **(** *[parameter1, parameter2, ...parametern]* **);**

Parameters are optional. However, when one test calls another, the **call** statement should designate a value for each parameter defined for the called test. If no parameters are defined for the called test, the **call** statement must contain an empty set of parentheses.

Any called test must be stored in a directory specified in the search path, or be called with the full pathname enclosed within quotation marks.

For example:

```
call “/u/andy/qa/test_z” ();
```

While running a called test, you can pause execution and view the current call chain. To do so, select Calls from the Debug menu.

Returning to the Calling Test

The **treturn** and **texit** statements are used to stop execution of called tests.

- The **treturn** statement stops the current test and returns control to the calling test.
- The **texit** statement stops test execution entirely, unless tests are being called from a batch test. In this case, control is returned to the main batch test.

Both functions provide a return value for the called test. If **treturn** or **texit** is not used, or if no value is specified, then the return value of the **call** statement is 0.

treturn

The **treturn** statement terminates execution of the called test and returns control to the calling test. The syntax is:

treturn [(*expression***)];**

The optional *expression* is the value returned to the **call** statement used to invoke the test. For example:

```
# test_a
if (call test_b() == "success")
    report_msg("test_b succeeded");
```

```
# test_b
if
(win_check_bitmap ("Win_2", "Calc", 1))

    treturn("success");
else
    treturn("failure");
```

In the above example, test_a calls test_b. If the bitmap comparison in test_b is successful, then the string "success" is returned to the calling test, test_a. If there is a mismatch, then test_b returns the string "failure" to test_a.

texit

When tests are run interactively, the **texit** statement discontinues test execution. However, when tests are called from a batch test, **texit** ends execution of the current test only; control is then returned to the calling batch test. The syntax is:

texit [(*expression***)];**

The optional *expression* is the value returned to the call statement used to invoke the test. For example:

```
# batch_test
return_val = call help_test();
report_msg("help returned the value "& return_val);

# help_test
call select_menu(help, index);
msg = get_text(4,30,12,100);
if (msg == "Index help is not yet implemented")
    texit("index failure");
...
```

In the above example, batch_test calls help_test. In help_test, if a particular message appears on the screen, execution is stopped and control is returned to the batch test. Note that the return value of the help_test is also returned to the batch test, and is assigned to the variable *return_val*. If **texit** is not used, *return_val* is 0.

Note: If you started XRunner from the command line, and you are running a test in regular mode, **texit** causes XRunner to exit.

For more information on batch tests, see Chapter 26, "Running Batch Tests."

Setting the Search Path

The Search Path determines the directories where XRunner will search for a called test. To set the search path, select Search Path from the Options menu and define a directory path in the Search Path form. The directories are searched in the order of their appearance in the form.

- To add a directory to the search path, type the directory name in the Pathname field. Use the Add Before and Add After buttons to position this directory in the list.
- To change a directory, select the directory in the list, make the desired changes in the edit field, and click the Change button.
- To delete a directory, select its name in the list and click the Delete button.

You can also set a search path by adding a **setvar** statement to a test script. This search path is valid only for the current test run.

For example:

```
setvar ("XR_SEARCH_PATH", "/u/pearl/tests/ui_tests");
```

This statement tells XRunner to search the /u/pearl/tests/ui_tests directory for called tests. For more information on the **setvar** function, see Chapter 33, "Changing System Defaults."

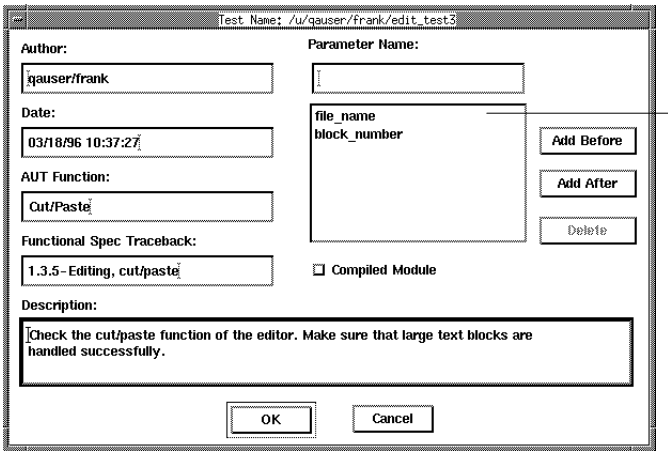
Defining Test Parameters

A parameter is a variable that is assigned a value from outside the test in which it is defined. You can define one or more parameters for a test; any calling test must then supply values for these parameters.

For example, suppose you define two parameters, *starting_x* and *starting_y* for a test. The purpose of these parameters is to assign a value to the initial mouse position when the test is called. Subsequently, two values supplied by a calling test will supply the x and y coordinates of the mouse pointer.

Defining Test Parameters in the Test Header Form

Parameters are defined in the Test Header form. To open this form select the Header command from the File menu.



To define a new parameter:

1. In the Test Header form, type the name of the parameter in the Parameter Name field.
2. Select one of the parameters in the list and then choose either Add After or Add Before.

   Note that since parameter values are assigned sequentially, the order in which parameters are listed determines the value that is assigned to a parameter by the calling test.
3. Click OK to close the form.

To delete a parameter from the parameter list:

1. In the Test Header form, select the name of the parameter to be deleted.
2. Press the Delete button.
3. Click OK to close the form.

Test Parameter Scope

The parameter defined in the called test is known as a *formal* parameter. Test parameters can be constants, variables, expressions, array elements, or complete arrays.

Parameters that are expressions, variables, or array elements are evaluated and then passed to the called test. This means that a copy is passed to the called test. This copy is local; if its value is changed in the called test, the original value in the calling test is not affected. For example:

```
# test 1 (calling_test)
i = 5;
call test 2(i);
print(i);  # prints "5"

# test 2 (called test), with formal parameter x
x = 8;
print (x);  # prints "8"
```

In the calling test (test_1), the variable *i* is assigned the value 5. This value is passed on to the called test (test_2) as the value for the formal parameter *x*.

Note that when a new value (8) is assigned to *x* in test_2, this change does not affect the value of *i* in test_1.

Complete arrays are passed by reference. This means that, unlike array elements, variables, or expressions, they are not copied. Any change made to the array in the called test influences the corresponding array in the calling test. For example:

```
# test q
a[1] = 17;
call test r(a);
print(a[1]); # prints "104"

# test_r, with parameter x
x[1] = 104;
```

In the calling test (test_q), element 1 of array *a* is assigned the value 17. Array *a* is then passed to the called test (test_r), which has a formal parameter *x*. In test_r, the first element of array *x* is assigned the value 104. Unlike the previous example, this change to the parameter in the called test does affect the value of the parameter in the calling test, because the parameter is an array.

All undeclared variables that are not on the formal parameter list of a called test are global and may be accessed and altered from another called or calling test. If a parameter list is defined for a test, and that test is not called but is run directly, then the parameters function as global variables for the test run. For more information about variables, refer to the *TSL Reference Manual*.

The test segments below illustrates the use of global variables. Note that test_a is not called, but is run directly.

```
# test a, with parameter k
i = 1;
j = 2;
k = 3;
call test_b(i);
print(j & k & l); # prints '256
```

```
# test_b, with parameter j
j = 4;
k = 5;
l = 6;
print (i & j & k); # prints '145'
```

Calling the check_file Test

XRunner allows you to evaluate the behavior of the application under test by capturing and comparing files. You do this by programming the following statement in your test script:

call check_file ("*filename***");**

Note that the full pathname must be included between quotes.

During test creation, XRunner captures the specified file and stores it as expected results. During test execution, XRunner compares the contents of the specified file with those stored as expected results. If a mismatch between these two files is detected, XRunner captures the current file.

The file comparison is performed using the UNIX *diff* utility. The **check_file** script is stored by default in the $M_ROOT/lib directory.

20

Creating User-Defined Functions

You can expand XRunner's testing capabilities by creating your own TSL functions. You can use these user-defined functions in a test or a compiled module. This chapter describes:

- Function Syntax
- Return Statement
- User-Defined Function Example

About User-Defined Functions

In addition to providing built-in functions, TSL allows you to design and implement your own functions. You can:

- create user-defined functions in a test script. You define the function once, and then call it from anywhere within the test (including called tests).
- create user-defined functions in a compiled module. Once you load the module, you can call the functions from any test. For more information, see Chapter 21, "Creating Compiled Modules."

User-defined functions are convenient in situations where you want to perform the same operation several times in a test script. Instead of repeating the code, you can write a single function that performs the operation. This makes your test scripts modular, more readable, and easier to debug and maintain.

A function can be called from anywhere in a test script. Since it is already compiled, execution time is accelerated. For instance, suppose you create a test that opens a number of files and checks their contents. Instead of

recording or programming the sequence that opens the file several times, you can write a function and call it each time you want to open a file.

Function Syntax

A user-defined function has the following structure:

[*class*] **function** name **(**[*mode*] *parameter...***)**
{
declarations;
statements;
}

Class

The class of a function can be either *static* or *public*. A static function is available only to the test or module within which the function was defined.

Once you execute a public function, it is available to all tests, as long as the test containing the function remains open. This is convenient when you want the function to be accessible from called tests. However, if you want to create a function that will be available to many tests, you should place it in a compiled module. The functions in a compiled module are available for the duration of the testing session.

If no class is explicitly declared, the function is assigned the default class, public.

Parameters

Parameters need not be explicitly declared. They can be of mode *in*, *out*, or *inout*. For all non-array parameters, the default mode is *in*. For array parameters, the default is *inout*. The significance of each of these parameter types is as follows:

in: A parameter which is assigned a value from outside the function.

out: A parameter which is assigned a value from inside the function.

inout: A parameter which can be assigned a value from outside or inside the function.

A parameter designated as out or inout must be a variable name, not an expression. When you call a function containing an *out* or an *inout* parameter, the argument corresponding to that parameter must be a variable, and not an expression. For example, consider a user-defined function with the following syntax:

```
function get_date (out todays_date) { ... }
```

Proper usage of the function call would be

```
get_date (todays_date);
```

Illegal usage of the function call would be

```
get_date (date[i]); or get_date ("Today's date is "& todays_date);
```

because both contain expressions.

Array parameters are designated by square brackets. For example, the following parameter list in a user-defined function indicates that variable *a* is an array:

```
function my_func (a[], b, c){ ... }
```

Array parameters can be either mode out or inout. If no class is specified, the default mode inout is assumed.

Declarations

Normally in TSL, declaration is optional. In functions, however, variables, constants, and arrays must all be declared. The declaration can be within the function itself, or anywhere else within the test script or compiled module containing the function. Additional information about declarations can be found in the *TSL Reference Guide*.

Variables

Variable declarations have the following syntax:

class variable [= init_expression];

The *init_expression* assigned to a declared variable can be any valid expression. If an *init_expression* is not set, the variable is assigned an empty string. The *class* defines the scope of the variable. It can be one of the following:

auto: An auto variable can only be declared within a function and is local to that function. It exists only as long as the function is running. A new copy of the variable is created each time the function is called.

static: A static variable is local to the function, test, or compiled module in which it is declared. The variable retains its value until the test is terminated by an Abort command.

public: A public variable can only be declared within a test or module, and is available for all functions, tests, and compiled modules.

extern: An extern declaration indicates a reference to a public variable declared outside of the current test or module.

Remember that you must declare all variables used in a function within the function itself, or within the test or module that contains the function. If you wish to use a public variable that is declared outside of the relevant test or module, you must declare it again as extern.

The extern declaration must appear within a test or module, before the function code. An extern declaration cannot initialize a variable.

For example, suppose that in Test 1 you declare a variable as follows:

```
public window_color=green;
```

In Test 2, you write a user-defined function which accesses the variable window_color. Within the test or module containing the function, you declare window_color as follows:

```
extern window_color;
```

With the exception of the auto variable, all variables continue to exist until the Abort command is executed. The following table summarizes the scope, lifetime, and availability (where the declaration can appear) of each type of variable:

Declaration	Scope	Lifetime	Declare the Variable in...
auto	local	end of function	function
static	local	until abort	function, test, or module
public	global	until abort	test or module
extern	global	until abort	function, test, or module

Note: In compiled modules, the Abort command initializes static and public variables. For more information, see Chapter 21, "Creating Compiled Modules."

Constants

The *const* specifier indicates that the declared value cannot be modified. The syntax of this declaration is:

[class] **const** *name [= expression];*

The *class* of a constant may be either public or static. If no class is explicitly declared, the constant is assigned the default class public. Once a constant is defined, it remains in existence until you exit XRunner.

For example, defining the constant MY_GUI_PATH using the declaration:

```
const MY_GUI_PATH = "/user/xrunner/gui";
```

means that the assigned value /user/xrunner/gui cannot be modified. (This value can only be changed by explicitly making a new constant declaration for MY_GUI_PATH.)

Arrays

The following syntax is used to define the class and the initial expression of an array. Array size need not be defined in TSL.

```
class array_name [ ] [=init_expression]
```

The array class may be any of the classes used for variable declarations (auto, static, public, extern).

An array can be initialized using the C language syntax. For example:

```
public hosts [ ] = {"lithium", "silver", "bronze"};
```

This statement creates an array with the following elements:

```
hosts[0]="lithium"
hosts[1]="silver"
hosts[2]="bronze"
```

Note that, as in C, arrays with the class *auto* cannot be initialized.

In addition, an array can be initialized using a string subscript for each element. The string subscript may be any legal TSL expression. Its value is evaluated during compilation. For example:

```
static gui_item [ ]={
    "class"="push_button",
    "label"="OK",
    "X_class"="XmPushButtonGadget",
    "X"=10,
    "Y"=60
    };
```

creates the following array elements:

```
gui_item ["class"]="push_button"
gui_item ["label"]="OK"
gui_item ["X_class"]="XmPushButtonGadget"
gui_item ["X"]=10
gui_item ["Y"]=60
```

Note that arrays are initialized once, that is the first time a function is run. If you edit the array's initialization values, the new values will not be reflected in subsequent test runs. To reset the array with the new initialization values, either interrupt test execution with the Abort command, or define the new array elements explicitly. For example:

Regular Initialization	**Explicit Definitions**

public number_list[]={1,2,3};	number_list[0] = 1;
	number_list[1] = 2;
	number_list[2] = 3;

Statements

Any valid statement used within a TSL test script can be used within a function, except for the **treturn** statement.

Return Statement

The **return** statement is used exclusively in functions. The syntax is:

return (*expression* **);**

This statement passes control back to the calling function or test. It also returns the value of the evaluated expression to the calling function or test. If no expression is assigned to the **return** statement, an empty string is returned.

User-Defined Function Example

The following user-defined function opens the specified text file in an editor. It assumes that the necessary GUI map file is loaded. The function verifies that the file was actually opened by comparing the name of the file with the label appearing in the window title bar after the operation is completed.

```
function open_file (file)
{
auto lbl;
set_window ("Editor");

# Open the Open form
menu_select_item ("File;Open...");

# Insert file name in the proper field and click OK to confirm
set_window ("Open");
edit_set("Open Edit", file);
button_press ("OK");

# Read window banner label
set_window ("Editor");
win_get_info("Editor","label",lbl);

#Compare label to file name
if ( file != lbl)
    return 1;
else
    return 0;
}
rc=open_file("/users/jon/dash/readme.txt");
pause(rc);
```

21

Creating Compiled Modules

Compiled modules are libraries of frequently-used functions. You can save user-defined functions in compiled modules and then call the functions from your test scripts.

- Contents of a Compiled Module
- Creating a Compiled Module
- Loading and Unloading a Compiled Module
- Incremental Compilation
- Compiled Module Example

About Compiled Modules

A compiled module is a script containing a library of user-defined functions that you want to call frequently from other tests. When you load a compiled module, its functions are automatically compiled and remain in memory. You can call them directly from within any test.

For instance, you can create a compiled module containing functions that:

- compare the size of two files
- check your system's current memory resources

Compiled modules can improve the organization and performance of your tests. Since you debug compiled modules before using them, your tests will require less error-checking. In addition, calling a function that is already compiled is significantly faster than interpreting a function in a test script.

You can compile a module in one of two ways:

- run the module script using the XRunner Run commands
- load the module from a test script using the TSL **load** function.

If you need to debug a module or make changes, you can use the Step command to perform incremental compilation. You only need to run the part of the module that was changed in order to update the entire module.

You can add **load** statements to your startup test. This ensures that the functions in your compiled modules are automatically compiled each time you start XRunner. For more information, see Chapter 34, "Initializing Special Configurations."

Contents of a Compiled Module

A compiled module, like a regular test you create in TSL, can be opened, edited, and saved. You indicate that a test is a compiled module by activating the Compiled Module radio button in the Test Header form (see "Creating a Compiled Module," in this chapter).

The contents of a compiled module differs from that of an ordinary test: it cannot include checkpoints or any analog input such as mouse tracking. The purpose of a compiled module is not to perform a test, but to store functions you use most so that they can be quickly and conveniently accessed from other tests.

Unlike an ordinary test, all data objects (variables, constants, arrays) in a compiled module must be declared before use. The structure of a compiled module is similar to a C program file, in that it may contain the following elements:

- function definitions and declarations for variables, constants and arrays. (For more information, see Chapter 20, "Creating User-Defined Functions.")
- prototypes of external functions (For more information, see the *TSL Reference Guide*.)
- **load** statements to other modules (For more information see "Loading and Unloading a Compiled Module" in this chapter.)

Note that when user-defined functions appear in compiled modules:

➤ A public function is available to all modules and tests, while a static function is available only to the module within which it was defined.

➤ The loaded module remains resident in memory even when test execution is aborted. However, all variables defined within the module (whether static or public) are initialized.

Creating a Compiled Module

Creating a compiled module is similar to creating a regular test script.

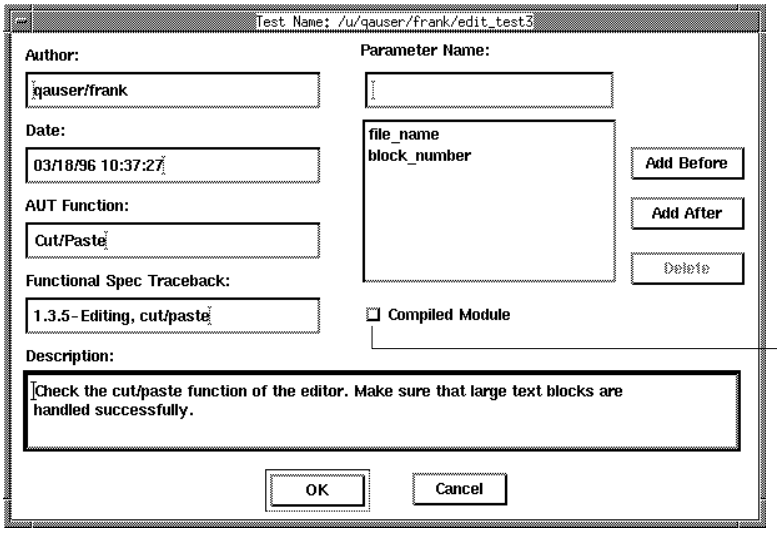
To create a compiled module:

1 Open a new test.

2 Write the user-defined functions.

3 Select Header from the File menu to open the Test Header form.

4 Select the Compiled Module button and click OK.



5 Choose Save from the File menu.

Save your modules in a location that is readily available to all your tests. When a module is loaded, XRunner locates it according to the Search Path

you define. For more information on defining a Search Path, see Chapter 19, "Calling Tests".

6 Compile the module using the **load** function. See "Loading and Unloading a Compiled Module" in the following section of this chapter for more information.

Loading and Unloading a Compiled Module

In order to access the functions in a compiled module you need to load the module. You can load it from within any test script using the **load** command; all tests will then be able to access the function until you quit XRunner or unload the compiled module.

If you create a compiled module that contains frequently-used functions, you can load it from your startup test. For more information, see Chapter 34, "Initializing Special Configurations."

You can load a module either as a *system* module or as a *user* module. A system module is generally a closed module that is "invisible" to the tester. It is not displayed when it is loaded, cannot be stepped into and is not stopped by a pause command. A system module is not unloaded when you execute an **unload()** statement with no parameters (global unload).

A user module is the opposite of a system module in these respects. Generally, a user module is one that is still being developed. In such a module you might want to make changes and compile them incrementally.

load

The **load** function has the following syntax:

load (*module_name [,1|0] [,1|0]* **);**

The *module_name* is the name of an existing compiled module.

Two additional, optional parameters indicate the type of module. The first parameter indicates whether the function module is a system module or a user module. 1 indicates a system module. 0 indicates a user module. (Default=0)

The second optional parameter indicates whether a *user* module appears in the Switch menu. 0 (the default) indicates that the module appears in the Switch menu. (Default=0).

When the **load** function is executed for the first time, the module is compiled and stored in memory. This module is ready for use by any test and does not need to be reinterpreted.

A loaded module remains resident in memory even when test execution is aborted. All variables defined within the module (whether static or public) are still initialized.

unload

The **unload** function removes a loaded module or selected functions from memory. It has the following syntax:

unload (*module_name* | *test_name [, "function_name"]* **);**

For example, the following statement removes all functions loaded within the compiled module named mem_test.

```
unload ("mem_test");
```

An **unload** statement with empty parentheses removes all modules loaded within all tests during the current session, except for system modules.

reload

If you make changes in a module, you should reload it. The **reload** function removes a loaded module from memory and reloads it (combining the functions of **unload** and **load**).

The syntax of the **reload** function is:

```
reload (module_name [,1|0] [,1|0] );
```

The *module_name* is the name of an existing compiled module.

Two additional optional parameters indicate the type of module. The first parameter indicates whether the module is a system module or a user module. 1 indicates a system module. 0 indicates a user module. (Default=0)

The second optional parameter indicates whether a *user* module appears in the Switch menu. 0 (the default) indicates that the module appears in the Switch menu. (Default=0).

Note: Do not load a module more than once. To recompile a module, use **unload** followed by **load**, or the **reload** function.

If you try to load a module that has already been loaded, XRunner does not load it again. Instead, it initializes variables and increments a *load counter*. If a module has been loaded several times, then the **unload** statement does not unload the module, but rather decrements the counter. For example, suppose that test A loads the module *math_functions*, and then calls test B. Test B also loads *math_functions*, and then unloads it at the end of the test. XRunner does not unload the function; it decrements the load counter. When execution returns to test A, *math_functions* is still loaded.

Incremental Compilation

In addition to using the **load** function to compile a module, you can also compile a module by executing the module script. This is especially useful when you are developing or modifying a module. If a module has already been loaded, and you modify it or add just a few lines, you can run those statements step by step. The compiled version of the module is automatically updated. Note that if you make a change within a function, you must run the entire function.

To perform incremental compilation:

1. Open the module.
2. In the Header form, deactivate the Compiled Module checkbox.

   Select OK to close the form.
3. To load an entire module, select Run from Top from the Run menu.

   To incrementally compile part of a module, run the necessary statements using the Step command.

4. Open the Header form and change the test type back to Compiled Module. Select OK to close the form.
5. Save the module if required and close it.

Compiled Module Example

The following module contains two simple, all-purpose functions that you can call from any test. The first function receives a pair of numbers and returns the number with the higher value. The second function receives a pair of numbers and returns the one with the lower value.

```
# return maximum of two values
function max (x,y)
{
if (x>=y)
    return x;
else
    return y;
}

# return minimum of two values
function min (x,y)
{
    if (x>=y)
        return y;
    else
        return x;
}
```

22

Using Dynamically Linked Libraries

XRunner allows you to call any function residing in an external dynamically linked library.

This chapter describes:

- Loading External Libraries
- Declaring External Functions in TSL
- Standard C Library Examples

About Calling External Functions

To extend the power of your automated tests, XRunner allows you to take advantage of functions in external dynamically-linked libraries. Your test scripts can call a function in any external library, provided you can interface with the library using C.

The **load_dll** TSL command allows you to access:

- standard function libraries such as libc, libll and libm
- custom libraries specific to your application

Before you can program calls to functions in external libraries, you perform two main steps:

1. Load the external library using a **load_dll** statement
2. Declare each function you need to call as an external type function

Loading External Libraries

You must load the external dynamically-linked libraries containing the functions you want to call, using the **load_dll** standard function. This function performs a runtime load of the required libraries. The syntax of the function is:

load_dll (*pathname***);**

The *pathname* is the full pathname of the dynamically-linked library to be loaded.

For example:

```
load_dll ("/usr/lib/libc.so.1.9");
```

To unload the dynamically-linked library, use the **unload_dll** function.

The syntax of the **unload_dll** function is:

unload_dll (*pathname***);**

Declaring External Functions in TSL

You must write an *extern* declaration for each C function you want to call. The extern declaration must appear before the function call. Normally, you store these declarations in an initialization test. (For more information, see Chapter 34, "Initializing Special Configurations.")

The syntax of the extern declaration is:

```
extern type function_name (param1, param2,...);
```

The *type* refers to the return value of the function. Type can be one of the following:

char (signed and unsigned)*float*
short (signed and unsigned)**double**
int (signed and unsigned)*string* (equivalent to char* in C)
long (signed and unsigned)*unsigned int*

Each parameter must include the following information:

```
[mode] type [name] [<size>]
```

- The *mode* can be either *in*, *out*, or *inout*. The default is *in*. Note that these values must appear in lowercase letters.
- The *type* can be any of the values listed for *type*, above, but not *float* or *double*.
- An optional *name* can be assigned to the parameter to improve readability.
- The <*size*> is required only for an *out* or *inout* parameter of type *string*. (See below.)

For example, suppose you want to call a function called set_clock that sets the time on a clock application. The function is part of an external DLL that you loaded with the **load_dll** statement. To declare the function, write:

```
extern int set_clock (string name, int time);
```

The set_clock function accepts two parameters. Since they are both input parameters, no mode is specified. The first parameter, a string, is the name of the clock window. The second parameter specifies the time to be set on the clock. The function returns an integer that indicates whether the operation succeeded.

Once the extern declaration is interpreted, you can call the set_clock function the same way you call a TSL built-in function:

```
result = set_clock ("clock v. 3.0", 3);
```

If an extern declaration includes an *out* or *inout* parameter of type *string*, you must budget the maximum possible string size by specifying an integer <*size*> after the parameter *type* or (optional) *name*. For example, the statement below declares the function get_clock_string, that returns the time displayed in a clock application as a string in the format of: "The time is..."

```
extern int get_clock_string (string clock, out string time <20>);
```

The *size* should be large enough to avoid an overflow. If no value is specified for *size*, the default is 128.

TSL identifies the function in your C code by its name only. You must pass the correct argument information from TSL to the C function. TSL does not check arguments. If the information is incorrect, the operation fails.

In addition, your C function must adhere to the following conventions:

- Any argument designated as a *string* in TSL must be associated with an argument of type *char** in C.
- Any argument of mode *out* or *inout* in TSL must be associated with a pointer in C. For instance, an argument *out int* in TSL must be associated with an argument *int** in the C function.

For example, the following declaration in TSL:

```
extern int set_clock (string name, inout int time);
```

must appear as follows in C:

```
int set_clock (char* name, int* time);
```

Standard C Library Examples

The following are several examples of tests that call functions in the standard C library, *libc*.

The gethostname Function

The first example uses the gethostname function to determine the current host machine. The program assigns a specific test to "HP3.0" if it is the current host. Otherwise it assigns a general test to the host.

```
# Load the appropriate file from the lib directory.
rc = load_dll ("/usr/lib/libc.so.1.8");

# Declare the external function.
extern int gethostname(out string name, in int len);

# Search for the first 10 characters of the host name.
gethostname(host_name, 10);
```

```
# Assign a specific test depending on the host machine.
if (host_name == “HP3.0”)
        call hp30_test();
else
        call general_test();
```

The time Function

The second example uses the rindex function to extract a filename from its full pathname. Unlike the TSL **index** function that indicates the position of one string within another, the rindex function returns a substring of a string, str. The substring begins from the last occurrence of the character c in the string and ends with the terminating character in the string.

```
# Load the appropriate file from the lib directory.
rc = load_dll ("/usr/lib/libc.so.1.9”);

# Declare the external function.
extern string rindex(string str, in char c);

# Assign the full pathname to an output variable
string = "/u/usr_name/dir_1/dir_2/dir_3/file_name”;

# Extract file_name from full pathname
a= substr (rindex (string, ascii (“/”)), 2);
```

The putenv Function

C library functions are not limited to read-only operations. This example illustrates how a library function can be used to assign a value to an environment variable. The TSL **getenv** function may be complemented with the putenv function.

```
# Load the appropriate file from the lib directory.
rc = load_dll ("/usr/lib/libc.so.1.8”);

# Declare the external function
extern int putenv();

# Assign a new value to an environment variable
putenv("ENV_VAR=my_value");
```

Customizing the TSL set_window Function

The next example uses the Xlib XBell and XRaiseWindow functions to customize the TSL **set_window** function. The customized version overrides the built-in function definition. In the built-in version, when a **set_window** statement fails, XRunner will return only an error code. In this version, whenever the statement fails, XRunner sounds a beep and raises the window to the top of the display.

```
# Load the X11.so.4.3 library.
rc = load_dll("/usr/openwin/lib/libX11.so.4.3");

# Declare the Xlib functions.
extern int XOpenDisplay(in string dpy);
extern int XFlush(in int dpy);
extern int XBell(in int dpy);
extern int XRaiseWindow(in int dpy, in int w);
public dpy;
extern int window_to_wid(in string window, in int x);

# Connect a client program to X server konishiki, server 0, display 0.
dpy = XOpenDisplay("konishiki:0.0");
if (!dpy)
    texit;
public function set_window(in winname, in time);
{
    autostatus = E_OK;
    auto oldwin, doagain;
    auto i=1, len;
    static last_wid;
    extern dpy;
    extern root;
    if (nargs() == 1)
        time = 0;

    # Save information about the set_window function call in case an error
occurs.
    save_report_info("set_window");
    len = length(winname);
    while (i<=len &&
        (substr(winname,i,1)==" "||
        substr(winname,i,1)=="\t"||
```

```
        substr(winname,i,1)==”\n “))
    i++;

    # If winname is description, return ILLEGAL_PARAMETER.
    if ((substr(winname,i,1)==”{“{
        status=E_ILLEGAL_PARAMETER;

        # Report the error description when an error occurs.
        process_return_value(status, winname, T_WINDOW, 0, doagain);
        return(status);
    }
    oldwin=GUI_get_window();

    # Identify the GUI object or window using the included input parameters.
    status = activate_function(T_WINDOW,winname,”_set_window”, time);
    if(status != E_OK)
        GUI_set_window(oldwin);
    else{
        status=GUI_set_window(winname);
        if(status == E_OK)
            win_find(winname, -1,-1,last_wid);

            # Raise the window to the top of the stacking order.
            XRAISEWindow(dpy,last_wid);

            # Ring the bell.
            XBell(dpy)
            XBell(dpy);

            # Flush the output buffer and display all queued requests
            XFlush(dpy);
        }
    return(status);
}
```

23

Using Regular Expressions

You can use regular expressions to increase the flexibility and adaptability of your tests. This chapter describes:

- When to Use Regular Expressions
- Regular Expression Syntax

About Regular Expressions

Regular expressions allow XRunner to identify objects with varying names or titles. You can use regular expressions in TSL statements or in object descriptions in the GUI map. For example, you can define a regular expression in the physical description of a pushbutton so that XRunner can locate the pushbutton if its label changes.

A regular expression is a string which specifies a complex search phrase. In most cases the string is preceded by an exclamation point (!). By using special characters such as a period (.), asterisk (*), caret (^), and brackets ([ ]), you define the conditions of the search. For example, the string "!windo.*" matches both "window" and "windows". See "Regular Expression Syntax" in this chapter for more information.

Note that XRunner regular expressions include options similar to those offered by the UNIX grep command. For additional information, see the UNIX manpages for ed(1).

When to Use Regular Expressions

Use a regular expression when the name of a GUI object can vary each time you run a test. For example, you can use a regular expression:

- In the physical description of an object in the GUI map, so that XRunner can ignore variations in the object's label. For example, the physical description:

  ```
  {
  class: push_button
  label: "!St.*"
  {
  ```

  allows XRunner to identify a pushbutton if its label toggles from "Start" to "Stop".

- In a GUI checkpoint, when evaluating the contents of an edit object or static text object with a varying name. You define the regular expression by creating a custom check for the object. For example, if you select a Check GUI command from the Create menu and double-click on a static text object, you can define a regular expression in the static text checks form:

  Note that when using a regular expression to perform a check on a static text object or edit object, it should *not* be preceded by an exclamation point.

- In a text checkpoint, to locate a varying text string using the **find_text** function. For example, the statement:

  ```
  find_text ("Edit", "win.*", coord_array, 640, 480, 366, 284);
  ```

  allows XRunner to find any text in the object named "Edit" that begins with "win".

  Note that when using a regular expression to perform a check on a static text object or edit object, it should *not* be preceded by an exclamation point.

  Since windows often have varying labels, XRunner defines a regular expression in the physical description of a window. For more information, see Chapter 5, "Editing the GUI Map."

Regular Expression Syntax

Regular expressions must begin with an exclamation point (!), except when defined in a GUI check form or in a text checkpoint. All characters in a regular expression are searched for literally, except for a period (.), asterisk (*), caret (^), and brackets ([ ]), as described below. When one of these special characters is preceded by a backslash (\), XRunner searches for the literal character.

The following options can be used to create regular expressions:

Matching Any Single Character

A period (.) instructs XRunner to search for any single character. For example,

```
welcome.
```

matches welcomes, welcomed, or welcome followed by a space or any other single character. A series of periods indicates a range of unspecified characters.

Matching Any Single Character within a Range

In order to match a single character within a range, you can use brackets ([ ]). For example, to search for a date that is either 1968 or 1969, write:

```
196[89]
```

You can use a hyphen (-) to indicate an actual range. For instance, to match a year in the 1960s, write:

```
196[0-9]
```

Brackets can be used in a physical description to specify the label of a static text object that may vary:

```
{
class: static_text,
label: "!Quantity[0-9]"
}
```

In the above example, XRunner can identify the static_text object with the label "Quantity" when the quantity number varies.

A hyphen does not signify a range if it appears as the first or last character within brackets, or after a caret (^).

A caret (^) instructs XRunner to match any character except for the ones specified in the string. For example:

```
[^A-Za-z]
```

matches any non-alphabetic character. The caret has this special meaning only when it appears first within the brackets.

Note that within brackets, the characters ".", "*", "[" and "\" are literal. If the right bracket is the first character in the range, it is also literal. For example:

```
[]g-m]
```

matches the "]" and g through m.

Matching One or More Specific Characters

An asterisk (*) instructs XRunner to match zero or more occurrences of the preceding character. For example:

```
Q*
```

causes XRunner to match Q, QQ, QQQ, etc. For example, in the following physical description, the regular expression enables XRunner to locate any pushbutton that starts with "O" (for example, On or Off).

```
{
class: push_button
label: "!O.*"
}
```

The above statement uses two special characters: "." and "*". Since the asterisk follows the period, XRunner locates any combination of characters. You can also use a combination of brackets and an asterisk to limit the label to a combination of non-numeric characters only:

```
{
class: push_button
label: "!O[a-zA-Z]*"
}
```

Part V

Running Tests

24

Running Tests

Once you have developed a test script, you run the test to check the behavior of your application.

This chapter describes:

- XRunner Test Execution Modes
- XRunner Run Menu Commands
- Running a Test to Check Your Application
- Running a Test to Debug Your Test Script
- Running a Test to Update Expected Results
- Controlling Test Execution by Modifying Configuration Parameters

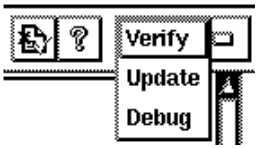
About Running Tests

When you run a test, XRunner interprets your test script, line by line. The execution arrow in the left margin of the test script marks each TSL statement as it is interpreted. As the test runs, XRunner operates your application as though a person were at the controls.

You can run your tests in three modes:

- Verify mode, to check your application
- Debug mode, to debug your test script
- Update mode, to update the expected results

You select the run modes from the dropdown list of modes on the toolbar or from the Run menu. The Verify mode is the default run mode.



Use XRunner's Run commands to run your tests. You can run an entire test, or a portion of a test. Before running a Context Sensitive test, load the necessary GUI map files. For more information, see Chapter 4, "Creating the GUI Map."

You can run individual tests, or use a batch test to run a group of tests. A batch test is particularly useful when your tests are long and you prefer to run them overnight or at other off-peak hours. For more information, see Chapter 26, "Running Batch Tests."

XRunner Test Execution Modes

XRunner provides three modes in which you run your tests—Verify, Debug, and Update. You use each mode during different phases of the testing process.

Verify Mode

Use the Verify mode to check your application. XRunner compares the *current* response of your application to its *expected* response. Any discrepancies between the current and expected responses are captured and saved as *verification results*. View the verification results in the Report form to determine the outcome of the test. For more information see Chapter 25, "Analyzing Test Results."

You can save as many sets of verification results as you require. To do so, continue to run your test, each time saving the results in a new directory. You specify the directory name for the results using the Set Results Directory form. This form appears every time you run a test in the Verify mode.

Debug Mode

Use the Debug mode to help you identify bugs in a test script. Running a test in the Debug mode is the same as running a test in the Verify mode, except that debug results are always saved in the *debug* directory. Because only one set of debug results is stored, the Set Results Directory form does not appear when you run a test in the Debug mode.

Once you run a test in the Debug mode, the Debug mode continues to be the default run mode for the current XRunner session—until you activate another mode.

Use XRunner's debugging facilities when you debug a test script:

- Control the execution of your tests using the Step commands in the Run menu. This allows you to execute a single line of a test script. For more information, see Chapter 30, "Debugging Test Scripts."
- Set breakpoints to pause test execution at specified points in the test script. For more information, see Chapter 31, "Using Breakpoints."
- Use the Watch List to monitor variables used in a test script as the test runs. For more information, see Chapter 32, "Monitoring Variables."

Update Mode

Use the Update mode to update the *expected results* of a test. For example, you might choose to update the expected results for a GUI checkpoint which checks a pushbutton, if the default status of the pushbutton changes from enabled to disabled.

Note that after a test has run in Update mode or has been aborted, Verify automatically becomes the default run mode, again.

By default, XRunner saves expected results in the *exp* directory, overwriting any existing expected results. Generally, only one set of expected results is stored, but you can create multiple sets if required. For more information, see "Generating Multiple Expected Results", on page243 in this chapter.

You can update the expected results for a test in one of two ways:

- Globally overwrite the full existing set of expected results by running the entire test using a Run command.

➤ Update the expected results for individual checkpoints and synchronization points using the Run from Arrow command or a Step command.

XRunner Run Menu Commands

You use the commands in XRunner's Run menu to execute your tests. When a test is running, the execution arrow in the left margin of the test script marks each TSL statement as it is interpreted.

Run from Top Command

The Run from Top command runs the active test from the first line in the test script. If the test calls another test, XRunner displays the script of the called test. Execution stops at the end of the test script.

Run from Arrow Command

The Run from Arrow command runs the active test from the line in the script marked by the execution arrow. In all other aspects, the Run from Arrow command is the same as the Run from Top command.

Quick Run Commands

The Quick Run commands deactivate the execution arrow during test execution and allow you to run tests more quickly. You can use the Quick Run commands to run a test either from the top of the test script, or from the execution arrow.

Step Commands

You use a Step command to run a single statement in a test script. For more information, see Chapter 30, "Debugging Test Scripts."

Abort Command

You can abort a test run immediately by selecting the Abort command. When you abort a test, test variables and arrays become undefined. Unsaved modifications to configuration parameters, however, are retained. After stopping a test, you can access only those functions that you loaded using the **load** command. You cannot access functions that you compiled using

the Run commands. Recompile these functions to regain access to them. For more information, see Chapter 21, “Creating Compiled Modules.”

Pause Command

The Pause command pauses test execution. Unlike the Abort command, which immediately terminates execution, a paused test continues running until all previously interpreted TSL statements are executed. When you pause a test, test variables and arrays maintain their values—as do unsaved modifications to configuration parameters. To resume execution of a paused test, select the appropriate Run command. Execution resumes from the point that you paused the test.

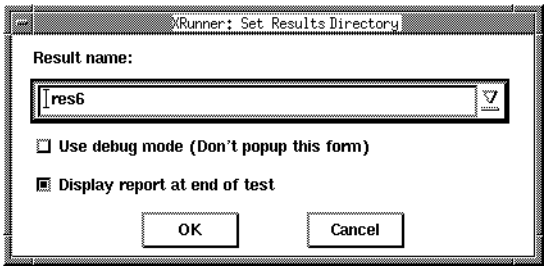
Running a Test to Check Your Application

When you run a test to check the behavior of your application, XRunner compares the current results with the expected results. You specify the directory in which the verification results for the test are saved.

To run a test to check your application:

1. Open the test if it is not already open.
2. Select Verify from the dropdown list of Run modes on the toolbar.
3. Choose the appropriate Run command.

   The Set Results Directory form opens, displaying a default directory name for the verification results, for example, res6.



4 You can save the test results under the default directory name. To use a different name, type in a new name or select an existing name from the dropdown list.

   To instruct XRunner to display the test report automatically following the test run (the default), select the "Display report at end of test" checkbox.

   Click OK. The Set Results Directory form closes and XRunner runs the test according to the Run command you selected.

5 Test results are saved in the directory you specified.

Running a Test to Debug Your Test Script

When you run a test to debug your test script, XRunner compares the current results with the expected results. Any differences are saved in the debug results directory. Each time you run the test in the Debug mode, XRunner overwrites the previous debug results.

To run a test to debug your test script:

1 Open the test if it is not already open.

2 Select Debug from the dropdown list of modes on the toolbar.

3 Choose the appropriate Run command.

   To execute the entire test, choose the Run from Top command. The test runs from the top of the test script and generates a set of Debug results.

   To execute a portion of the test, choose the Run from Arrow command, or one of the Step commands. The test runs according to the Run command you selected, and generates a set of Debug results.

Running a Test to Update Expected Results

When you run a test to update expected results, the new results replace the expected results created earlier and become the basis of comparison for subsequent test runs.

To run a test to update the expected results:

1. Open the test if it is not already open.
2. Select Update from the dropdown list of modes on the toolbar.
3. Choose the appropriate Run command.

   To update the entire set of expected results, choose the Run from Top command.

   To update only a portion of the expected results, choose the Run from Arrow command, or one of the Step commands.

   XRunner runs the test according to the Run command you select and updates the expected results. The default directory for the expected results is *exp*.

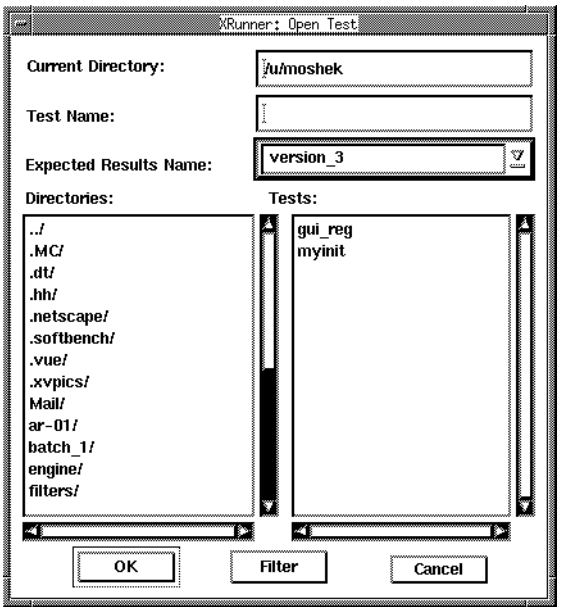
Generating Multiple Expected Results

You can generate more than one set of expected results for any test. You may want to generate multiple sets of expected results if, for example, the response of your application varies according to the time of day. In such a scenario, you would generate a set of expected results for each defined period of the day.

To create a different set of expected results for a test:

1. Choose Open from the File menu. The Open Test form opens. Note that if the test is already open, you must reopen it.

2 In the Open Test form, select the test for which you want to create multiple sets of expected results. Enter a unique directory name for the new expected results in the Expected field.



3 Click OK. The Open Test form closes.

4 Select Update from the dropdown list of Run modes on the toolbar.

5 Choose the Run from Top command to generate a new set of expected results.

XRunner runs the test and generates a new set of expected results, in the directory you specified.

Running a Test with Multiple Sets of Expected Results

If a test has multiple sets of expected results, you specify which expected results to use before running the test.

To run a test with multiple sets of expected results:

1 Choose Open from the File menu. The Open Test form opens.

Note that if the test is already open, but is accessing the wrong set of expected results, you must reopen the test.

2 In the Open Test form, select the test that you want to run. The Expected field displays all the sets of expected results for the selected test.

3 Select the required set of expected results from the Expected field, and click OK. The Open Test form closes.

4 Select Verify from the dropdown list of Run modes on the toolbar.

5 Choose the appropriate Run command. The Set Results Directory form opens, displaying a default directory name for the verification results—for example, RES_1.

6 Click OK to begin test execution, and to save the test results in the default directory. To use a different verification results directory, type in a new name or select an existing name from the dropdown list.

Click OK. The Set Results Directory form closes. XRunner runs the test according to the Run command you selected and saves the test results in the directory you specified.

Controlling Test Execution by Modifying Configuration Parameters

You can control how a test is run using XRunner's configuration parameters. For example, you can set the time XRunner waits at a bitmap checkpoint for a bitmap to appear, or the speed that a test is run.

You modify parameter settings from the Configuration form. To display the form choose Configure from the Options menu. You can also modify configuration parameter settings from within a test script using the **setvar** function.

For example, the default for the XR_MIN_DIFF parameter (that defines the minimum number of pixels that constitute a bitmap mismatch) is 0. If you assign a new value to a configuration parameters, you can choose to make this the default value when you exit XRunner.

For a more comprehensive discussion of controlling test execution with configuration parameters, refer to Chapter 33, "Changing System Defaults."

25

Analyzing Test Results

After you execute a test, you can view a report of all the major events that occurred during the run and analyze the success or failure of the test.

This chapter describes:

- Test Results Summary
- Test Results Log
- The Test Tree
- Viewing All Captures
- Viewing the Results of a Test
- Viewing the Results of a GUI Checkpoint
- Viewing the Results of a Bitmap Checkpoint
- Controlling How Bitmaps are Displayed
- Filtering Results
- Updating Expected Results
- Printing Results

About Viewing Test Results

The Report form presents all the information related to the results of test execution. You can view both textual and graphical representations of the test run. The type of information displayed depends on whether the test was run in Verify mode and on the types of checkpoints performed (bitmap or GUI).

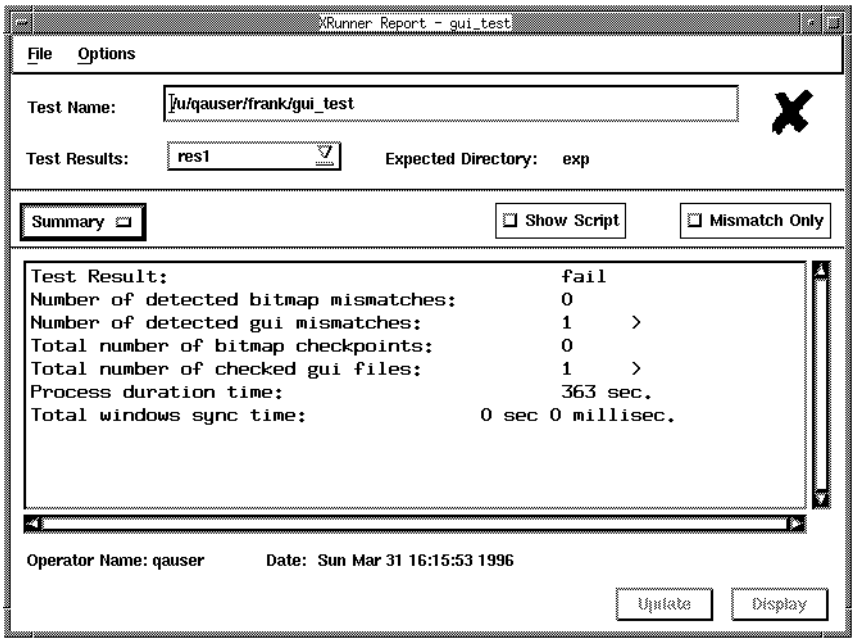
XRunner provides the following information:

- The *test summary* tells you whether the test passed or failed and lists all checkpoints that were performed. Technical details include the test name; the expected results directory (for verification runs); the time and date of execution; and the name of the test operator.
- The *test log* details the major events that occurred during the test run. These include the start and termination of the test; GUI and bitmap checkpoints; changes in the progress of the test flow; changes to configuration parameters; displayed report messages; and run-time errors.
- The color-coded *test tree* shows all tests executed during the run (including called tests) and the result for each. Select a test from the tree to "jump" into the results for that test.
- The *All Captures* option displays all captured GUI and bitmap data for the current test.

Depending on the test performed, you can view the following:

- *Bitmaps* captured during the run: for a test run containing no mismatches, this will be the bitmap(s) constituting the *expected* results; following a verification mismatch, this will be the bitmap(s) constituting the *difference* between the expected and the actual bitmap.
- For a captured *GUI*, the results for each check are displayed in table form: for all runs, the *expected* results for each check; following a verification mismatch, you can see the *actual* results as well as the expected results for each check.
- For all tests, any *error message*, *user message* or *system message* generated during the run.

Test Results Summary



When you open a report for a test run, XRunner displays a summary. The following information can appear:

Test Name: The name of the test.

A check or a cross: Depicts the overall result of the test run—either pass or fail.

Results Directory: The directory containing the verification results of the last test run.

Expected Results Directory: The name of the expected results directory used for the test.

Summary: Displays Test Results Summary. Also toggles between other displays: Test Log and All Images.

Test Result: Indicates whether the test passed or failed. For a batch test, this refers to the batch test and not to the main tests that it called.

Number of detected bitmap mismatches: Number of bitmaps that did not match expected results.

If you double-click on a bitmap checkpoint line displaying a bitmap, XRunner displays a detailed list of bitmap checkpoints for the test. When the list is displayed, the '>' sign changes to '<'.

Bitmap checkpoints are displayed in the following format: *Img_1 clock(3)* indicates: the first bitmap (_1) captured for the clock test. The number in parentheses indicates the line in the test script that contains the **check_window**, **win_check_bitmap** or **obj_check_bitmap** statement associated with this bitmap, in this case line 3.

The letters E, A or D following the bitmap checkpoint indicate whether there is an expected, actual of difference available stored on the disk.

To display available bitmaps, double click on the checkpoint, or highlight the checkpoint and press Display. By highlighting several entries, you can display the bitmaps captured for several checkpoints.

Number of detected GUI mismatches: Total number of verified GUI captures that did not match expected results.

If you double-click on a GUI mismatches line displaying a '>', XRunner displays a detailed list of GUI checkpoints. If the list is displayed, the '>' changes to a '<.'

GUI checkpoints are displayed in the following format: *gui_1 clock (4)* indicates: GUI checkpoint number 1, captured for the clock test. The fourth line in the test script contains the **obj_check_gui** or **win_check_gui** statement associated with this GUI capture.

To display the GUI data, double-click on the GUI checkpoint, or highlight the checkpoint and press Display. By highlighting several entries, you can display the GUI data captured for several checkpoints.

Total number of bitmap checkpoints: Number of captured bitmaps.

Total number of GUI checkpoints: Number of GUI captures.

Total run time: Total time (in hr:min:sec) that elapsed from start to finish of the current test run.

Total windows sync time: Total time (in seconds and milliseconds) spent on synchronizing window events.

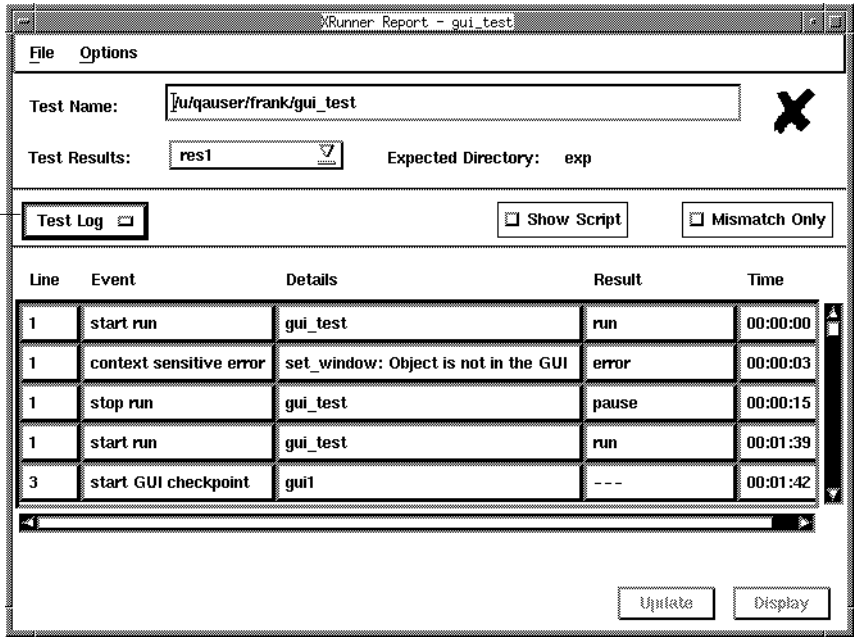
Date: The date and time of the test run.

Operator Name: The login name of the user who ran the test.

Display: Displays captured GUI data or bitmaps, depending on the selected line in the Report.

Update: Replaces the previously expected GUI data or bitmaps with those in the actual results. If several GUI or bitmap checkpoints are selected when you press Update, the combined actual results for all selected checkpoints are stored as expected results.

Test Results Log



To display the test log, select Test Log from the Display options. The log provides detailed information on every major event that occurred during

the specified execution of the selected test. The row describing a successful checkpoint appears, by default, in green. The row describing a mismatch or failure appears, by default, in red. You can change the default report color settings from the Configuration form:

- The XR_PASS_COLOR configuration parameter defines the color indicating a successful test run.
- The XR_FAIL_COLOR configuration parameter defines the color indicating a failed test run.

If XRunner performed GUI and bitmap checks or transmitted messages during test execution, you can click on the event in the log to see the relevant information. The log can include the following:

Show Script: When activated, highlights the TSL command corresponding to the line in the report.

Mismatch Only: When activated, displays only mismatches for the test run.

Bitmap checkpoints: double-click on the row to view the selected bitmap. In the case of a mismatch, you can view the Expected bitmap, the Actual bitmap captured and a Difference bitmap showing the discrepancy between the two. You can view the bitmaps captured for several checkpoints by clicking on several rows.

GUI checkpoints: double-click on the row to bring up the Verification Results form for the selected checkpoint.

System, User, or Error messages created during the test run: double-click on the row to see the message.

The Test Tree

The test tree opens automatically when you select Test Log. You can also open it by selecting Test Tree from the Options menu of the Results form.

When you view the results of a test that calls other tests, the test tree shows all the tests associated with this test run.

Each test is color-coded to indicate whether it passed or failed and to show the currently selected test:

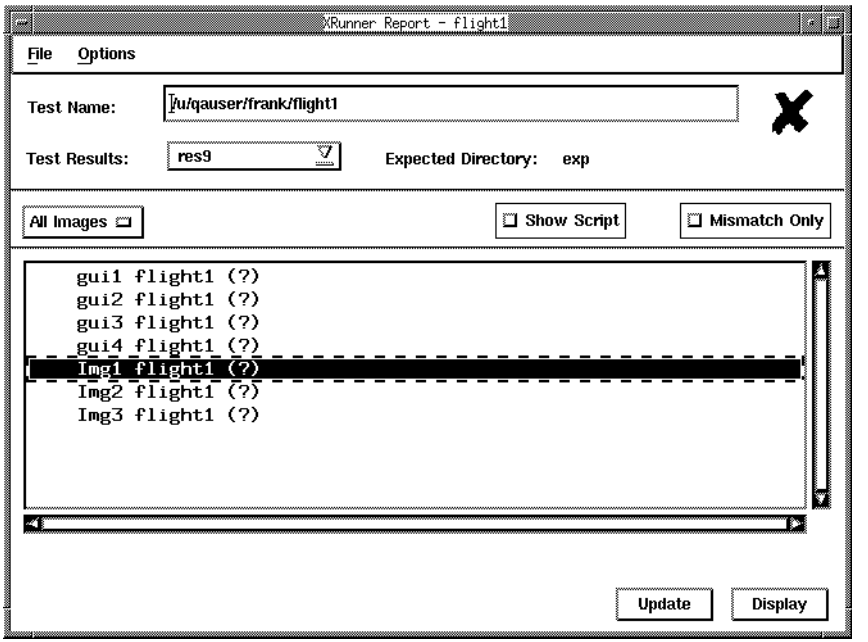
- Shading indicates the currently selected test.
- Green indicates a test that passed.
- Red indicates a test that failed.

Red and green are the default colors for failed and successful test runs. Report colors are configurable. For more details, see “Test Results Log”, on page251 in this chapter.

Before viewing test results, make sure that all tests called during the run are in the XRunner search path. For more information, see Chapter 19, “Calling Tests.”

Viewing All Captures

To display a complete list of GUI and bitmap data stored in a specific results directory, select All Captures from the Display options.



All Captures displays all existing expected, actual and difference data for all GUI and bitmap checkpoints in the selected results directory. For example, suppose that a test bitmap_chk_2 contains five bitmap checkpoints. If, during the last test run, you executed a test from the line containing the third checkpoint, All Captures would allow you to view the data captured during all previous runs, as well as those captured for points 3,4 and 5 in the last run.

To display one or more files in the list, double-click the appropriate lines, or highlight the lines and press Display.

To update expected results, highlight the line and press Update.

Viewing the Results of a Test

When a test run is completed, you can view detailed test results in the Reports form. To open the form, select Reports from the Tools menu or click on the Reports icon. If you ran a test in Verify mode and selected the "Show Results" option in the Results form, the Report automatically opens when a test run is completed. For more information, see Chapter 24, "Running Tests."

The Report form opens and displays the results of the current test. You can view both expected, debug, and verification results in the Report form. By default, the Report form displays the results of the most recently executed test run. To view other results, select a results directory from the Test Results list. Examine the results in the test summary and test log. When available, click on called tests in the test tree to view their results. In order to place the report file automatically into the test directory, insert the following line when performing the XRunner configuration:

```
XR_AUTO_REPORT = TRUE
```

To close the Report form, select the Quit command from the File menu.

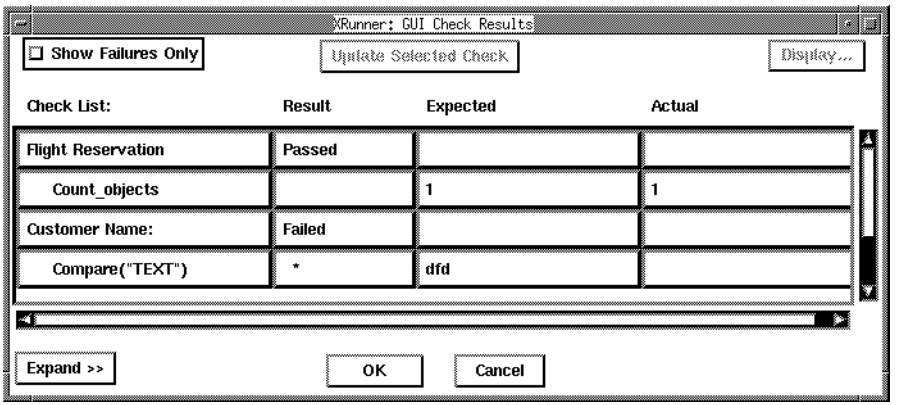
Viewing the Results of a GUI Checkpoint

A GUI checkpoint helps you to identify changes in the look and performance of GUI objects in your application. The results of a GUI checkpoint are presented in a table opened from the Report form. The table lists every object checked during the GUI check and the type of check performed. Each check is listed as either passed of failed and the expected and actual results are shown. If one or more objects fail, the entire GUI checkpoint is marked as failed in the test log.

To display the results of a GUI checkpoint:

1 Open the Report form. In the test log, look for entries that contain the end check GUI statement in the Event column. Failed GUI checks appear in red; passed GUI checks appear in green.

2 Double-click on a GUI checkpoint entry in the test log or click on the GUI checkpoint entry and press the Display button. The GUI Check Results form opens:



Checklist: Displays the complete GUI checklist. Each object in the checklist is followed by its check(s). Checks in the Checklist are indented.

Result: Indicates the result for each check, either "Passed" or "Failed." An asterisk next to the check indicates a mismatch.

Expected: Displays the expected results for each check.

Actual: Displays the actual results for each check (in the case of GUI mismatch only).

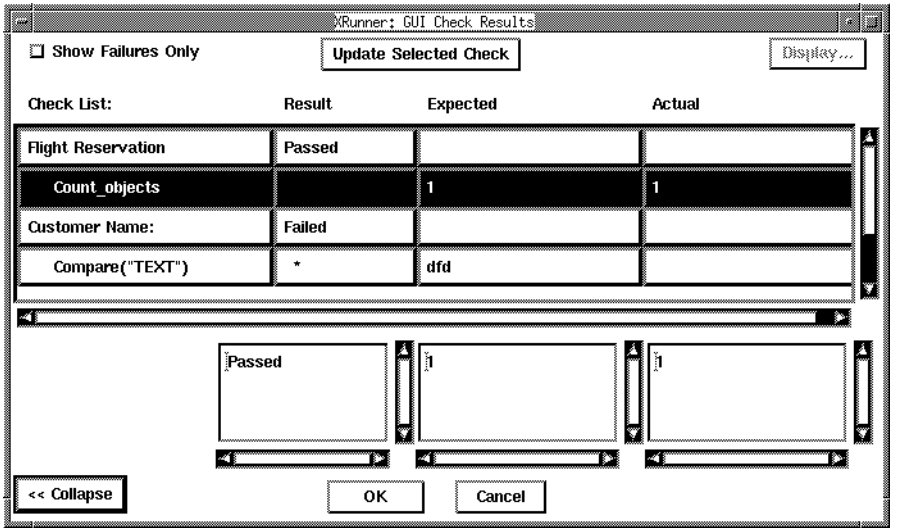
Show failures only: When checked, only the checks that failed are displayed.

Update selected check: Replaces the expected results with the actual results for a selected check.

Expand: When a particular check is selected, displays values that are too long to fit into the main list box.

3 The form lists every object checked and the type of check performed. Each check is listed as either passed or failed and the expected and actual results are shown.

To view results that are too long to fit into the main table, click the Expand button to expand the form. Press the Collapse button to collapse the form to its original size.



4 To display only failed checks, select the Show Failures Only checkbox.

5 Click OK to close the form.

Viewing the Results of a Bitmap Checkpoint

A bitmap checkpoint compares expected and actual bitmaps in your application. In the Report form you can view pictures of the expected and actual results. If a mismatch is detected by a bitmap checkpoint during a verification run, you can also view bitmaps showing the differences between the expected and actual results.

To view the results of a bitmap checkpoint:

1 In the test log, look for entries that contain the check bitmap statement in the Event column.

2 To display the results of one or more bitmap checkpoint, double-click on their entries in the log or select the entries and press the Display button. For

a mismatch, the expected, actual, and difference bitmaps are displayed; for all other runs, only the bitmaps constituting the expected results are displayed.

Expected *Actual* *Difference*

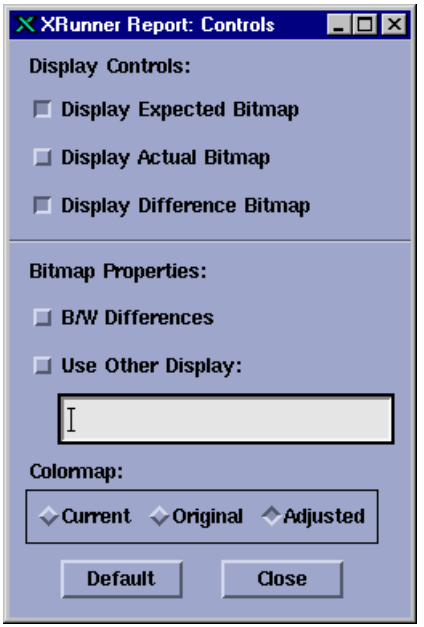
3 To remove a bitmap from the screen, click anywhere inside the window.

Controlling How Bitmaps are Displayed

For bitmap checkpoints, you can control which types of bitmaps are displayed. You can also control whether to use the colormap stored with the bitmap, or the currently installed colormap.

To control how bitmaps are displayed:

1 In the report Options menu, select the Controls command. The Controls form opens.



2 Click on checkboxes to select options.

3 Click OK to save the display configuration and close the form.

The Controls form includes the following buttons and fields:

Display Expected Bitmap: Displays the expected bitmap

Display Actual Bitmap: Displays the actual bitmap captured during the verification run (available only when a mismatch occurs).

Display Difference Bitmap: Displays the discrepancy between the expected and actual bitmaps (available only when a mismatch occurs).

B/W Differences: When activated, difference bitmaps are displayed in black and white instead of color. This may make it easier to discern the recorded discrepancy.

Use Other Display: Sends the display of the specified bitmap to the screen of another workstation. Use the X DISPLAY environment variable format to specify the destination display. For example, type in mercury:0, where mercury is a server name for which you have the appropriate authorization.

Current: Uses the colormap currently installed to display the selected bitmap. No colormap changes occur on the screen. This means that if the AUT uses a different colormap than the one currently installed, the colors in the displayed bitmap may be different from those used when you captured the bitmap.

Original: Installs the colormap stored with the captured bitmap. This colormap affects all windows currently displayed on the screen and may distort the colors used in the current environment.

Adjusted: Modifies the current colormap in order to approximate the correct colors in the displayed bitmap window. This may be useful, for example, for displaying color bitmaps on a monochrome display. Selecting this option affects all other windows on the screen.

Note that only one colormap can be installed at any given time. If your tested application uses a different colormap than other windows on the screen, you may have to compromise while displaying bitmaps: either the captured AUT window bitmap or some other windows opened on the screen may be displayed with the wrong colors.

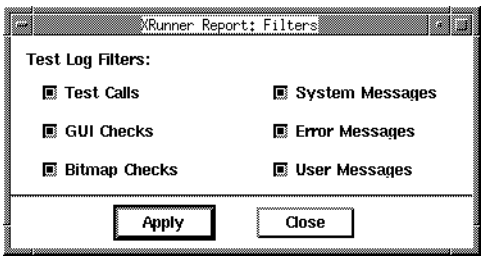
If your window manager is set to change the colormap manually, then in addition to selecting the Original or Adjusted options, you may need to "unlock" the colormap focus using the appropriate menu command or softkey. For further information, see your window manager documentation.

Filtering Results

You can choose to view only specific types of results by filtering the events in the test log.

To filter results:

1 Select Filters from the report Option menu. The Filters form opens.



2 Select the types of results that you want to view in the test log. Deselect any result types that you do not want to view. The default filter configuration displays all result types.

3 Click Apply to save the filter configuration for the current XRunner session and to close the form.

4 Click Close to close the form.

Updating Expected Results

If a bitmap or GUI checkpoint fails, you can update the expected results directory (*exp*) with data in the verification results directory. The next time you run the test, the new expected results will be compared to the actual results in the application.

To update the expected results of a bitmap checkpoint:

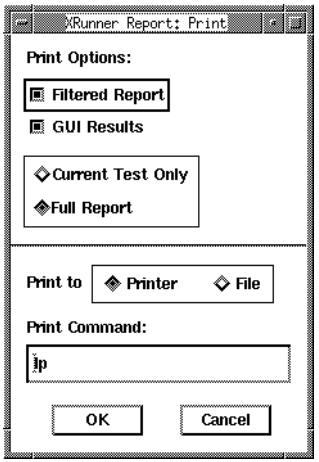
1. Select one or more mismatched bitmap checkpoint from the test log.
2. Press the Update button.
3. A message states that overwriting expected results cannot be undone. Select Yes to update the results.

To update the expected results of a GUI checkpoint:

1. In the test log, select one ore more mismatched GUI checkpoint.
2. To update the results for the entire GUI checkpoint, click the Update button.
3. To update the results for a specific check within the GUI checkpoint, double-click on the GUI checkpoint entry in the log, or press the Display button. The GUI Check Results form opens.
4. Select a failed check and click the Update Selected Check button.

Printing Results

You can print test results directly from the Report form. Simply select the Print command from the File menu in the form.



In the Print form that is displayed, you can select

- a filtered report or GUI results
- the results for the current test or for a full test run (for example a test call chain)
- to send test results to a printer or to a disk file

Click OK to print the results.

26

Running Batch Tests

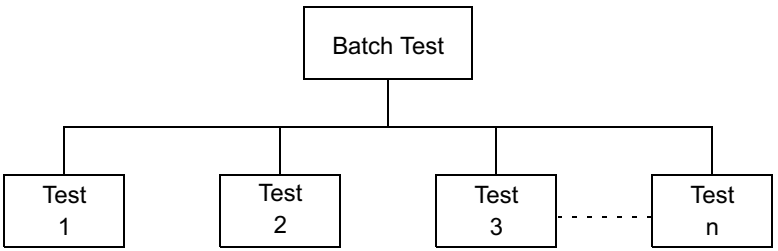
XRunner allows you to execute a group of tests unattended. This can be particularly useful when you want to run a large group of tests overnight or at other off-peak hours.

This chapter describes:

- Creating a Batch Test
- Executing a Batch Test
- Storing Batch Test Results
- Viewing Batch Test Results

About Running Batch Tests

You can run a group of tests unattended by creating and executing a single batch test. A batch test is a test script which contains call statements to other tests. It opens and executes each test, and saves the test results.



At first glance, a batch test appears to be a regular test that includes call statements. A test only becomes a “batch test” when the batch test flag has

been set to ON before you execute the test. When running in Batch mode, XRunner suppresses all messages that would ordinarily be displayed during execution, such as a message reporting a bitmap mismatch. XRunner also suppresses all **pause** statements and any halts in execution resulting from runtime errors.

By suppressing all messages, XRunner can run a batch test unattended. This differs from a regular, interactive test run in which messages appear on the screen and prompt you to click a button in order to resume test execution. A batch test enables you to run tests overnight or during off-peak hours, so that you can save time while testing your application.

When a batch test run is completed, the results can be viewed in the XRunner Report form. The form displays the results of all the major events that occurred during the run.

Note that you can also execute a group of tests from the command line. For more information, see Chapter 27, "Running Tests from the Command Line."

Creating a Batch Test

A batch test is a test script that calls other tests. You program a batch test by typing call statements directly into the test window and setting the value for the XR_BATCH_MODE parameter in the Configuration form to ON.

A batch test may include programming elements such as loops and decision-making statements. Loops enable a batch test to run called tests a specified number of times. Decision-making statements such as *if/else* and *switch* condition test execution on the results of a test called previously by the same batch script. For more information, see Chapter 17, "Enhancing Your Test Scripts with Programming."

For example, the following batch test executes three tests in succession, then loops back and calls the tests again. The loop specifies that the batch test should call the tests ten times.

```
for (i=0; i<=10; i++)
    {
    call"/u/andy/pbtests/open" ();
    call"/u/andy/pbtests/save" ();
    call"/u/andy/pbtests/setup" ();
    }
```

Executing a Batch Test

You execute a batch test in the same way that you execute a regular test. Choose a mode (Verify, Update, or Debug) from the dropdown list in the icon bar or the Run menu and then select the Run from Top command from the Run menu. For more information, see Chapter 24, "Running Tests."

When you run a batch test, XRunner opens and executes each called test. All messages are suppressed so that the tests are run without interruption. If you are running the batch test in Verify mode, the current test results are compared to the expected test results saved earlier. If you are running the batch test in order to update expected results, new expected results are created in the expected results directory for each test. See the section "Storing Batch Test Results" in this chapter for more information. When the batch test run is completed, the test results can be viewed in the Report form.

Note that if your tests contain TSL **texit** statements, XRunner interprets these statements differently for a batch test run than for a regular test run. During a regular test run, **texit** terminates test execution. During a batch test run, **texit** halts execution of the current test only and control is returned to the batch test.

You can enable a batch test to run unattended using either of two methods:

➤ **Using the Configuration form:** Before running the batch test from XRunner, ensure the setting for the XR_BATCH_MODE configuration parameter is ON. The default setting is OFF. Click Apply to apply the change for the current session or Save to apply the change for current and future sessions.

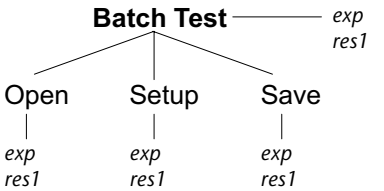
➤ **From the command line:** Run the batch test using the -batch option. For example, to run the batch test /u/bert/qa/batch1, enter the following at the UNIX prompt:

```
xrun -t /u/bert/qa/batch1 -batch ON
```

Storing Batch Test Results

When you run a regular, interactive test, results are stored in a subdirectory under the test. The same is true when a test is called by a batch test. XRunner saves the results for each called test separately in a subdirectory under the test. A subdirectory is also created for the batch test that contains the overall results of the batch test run.

For example, suppose you create three tests, *Open*, *Setup*, and *Save*. For each test, expected results are saved in a *exp* subdirectory under the test directory. Suppose you also create a batch test that calls the three tests. Prior to executing the batch test in Verify mode, you tell XRunner to save the results in a directory called *res1*. When the batch test is run, it compares the current test results to the expected results saved earlier. Under each test directory, XRunner creates a subdirectory called *res1* in which it saves the verification results for the test. A *res1* directory is also created under the batch test to contain the overall verification results for the entire run.



If you run the batch test in Update mode in order to update expected results, XRunner overwrites the expected results in the *exp* subdirectory under each test and under the batch test.

Note that if you run the batch test from XRunner, without modifying the XR_BATCH_MODE configuration parameter or using the -batch option, XRunner saves results only in a subdirectory under the batch test. This can cause problems later, if you choose to run the tests independently, since

XRunner will not know where to look for the previously saved expected and verification results.

Viewing Batch Test Results

When a batch test run is completed, you can view information about the events that occurred during the run in the XRunner Report form. If one of the called tests fails, then the batch test is marked as failed.

The test log section of the Report form lists all the events that occurred during the batch test run. Each time a test is called, a *call_test* entry is listed in the log. To view the results of the called test, double-click on its *call_test* entry. For more information on viewing test results in the Report form, see Chapter 25, "Analyzing Test Results."

27

Running Tests from the Command Line

You can run tests directly from your UNIX command line according to the options that you define in advance.

This chapter describes:

- Using the Command Line with XRunner
- Command Line Options

About Running Tests from the Command Line

Running tests from the command line enables you to use XRunner together with UNIX system facilities and other processes and applications. This is particularly useful for performing a large batch run of tests requiring much system initialization and administration. You can use the command line to prepare a script that:

- recompiles the new AUT release
- loads the relevant tests
- creates a temporary work space for running the tests
- invokes XRunner
- executes the tests
- moves the test results to a storage area

Using the Command Line with XRunner

Most of the functional options that you can set within XRunner can also be set from the command line. For example, the following command invokes XRunner, loads a batch test, sets the results directories and the delay option, and runs the test.

```
xrun -E -t /v1_3/newclock -run -batch -exp ER1 -verify res5 -delay 3&
```

An Execution license copy of XRunner is invoked. The test, newclock (located in directory v1_3), is loaded and then run. The expected results for the run are stored under the subdirectory ER1. The verification results are written to subdirectory res5. The delay between consecutive screen samplings is set to three seconds.

Command Line Options

The following is a description of each command line option.

-auto_load {on | off}

Activates or deactivates automatic loading of the temporary GUI map file. (Default = **on**)

-auto_load_dir pathname

Determines the directory in which the temporary GUI map file (temp.gui) resides. This option is applicable only when auto load is on.

-batch {on | off}

Runs the loaded test as a batch test.
(Default = **off**)

-beep {on | off}

Activates or deactivates the XRunner system beep.
(Default = **on**)

-click_delay *non-negative integer*

Defines the delay (in tenths of a second) that XRunner waits after interpreting a single click of a mouse button.
(Default = **10** [tenths of a second])

-compress {on | off}

Activates or deactivates the compressed storage of captured bitmaps.
(Default = **off**)

-cycle

Specifies the cycle name. (Used for TestDirector integration.)

-D | -E

Specifies what type of XRunner license is to be invoked: Development (-D) or Execution (-E).
(Default = **-D**)

-dblclk_time *non-negative integer*

Defines the longest possible interval (in tenths of a second) that can elapse between two clicks for them to still constitute a double-click.
(Default = **30** [tenths of a second])

-delay *non-negative integer*

Defines the time (in seconds) that XRunner waits between consecutive samplings of the screen.
(Default = **1**[seconds])

-display *display name*

Displays the XRunner user interface on a remote workstation. Although XRunner and the AUT both run on the same local machine, the XRunner window will appear on the designated remote machine.

The display name must specify the name of the remote machine on which you want the XRunner interface to appear and the number of the display.

-exp *expected_results_name*

Assigns a name to the subdirectory in which expected results are stored. In a verification run, specifies the set of expected results used as the basis for the verification comparison.
(Default = exp)

-fast_replay {on | off}

Activates or deactivates fast test execution.
(Default = **off**)

-focus_delay *non-negative integer*

Defines the delay (in tenths of a second) that XRunner waits from the time the mouse is moved to a new window until input is entered.
(Default = **10** [tenths of a second])

-fontgrp *group_name*

Specifies the active font group when XRunner is invoked.

-kbd_delay *non-negative integer*

Defines the delay (in tenths of a second) that XRunner waits after interpreting a keyboard entry.
(Default = **0**)

-min_diff *non-negative integer*

Defines the number of pixels that constitute the threshold for a bitmap mismatch.
(Default = **0** [pixels])

-mismatch_break {on | off}

Activates or deactivates Break on Mismatch before a verification run.

The functionality of Break on Mismatch is different than when running a test interactively. In an interactive run, the test is paused. For a test invoked from the command line, the first occurrence of a comparison mismatch terminates test execution.
(Default = **off**)

-move_windows {on | off}

Activates or deactivates the automatic relocation of windows.
(Default = **on**)

-raise_windows {on | off}

Activates or deactivates the automatic raising of moved windows.
(Default = **on**)

-redraw *non-negative integer*

Defines the time (in seconds) that XRunner waits for the screen to be redrawn after it moves a window during test execution.
(Default = **3**[seconds])

-run

Instructs XRunner to execute the loaded test.

-script_font *font name*

Specifies the font in the XRunner window.

-search_path *pathname*

Defines the directories to be searched for tests to be opened and/or called. Note that the search_path can specify multiple directories. The search path is given as a string enclosed in quotation marks; a space serves as the delimiter between adjacent directory names.
(Default = Invocation directory and $M_ROOT/lib)

-server *name*

Accesses an X server on a remote workstation. The server name should specify the name of the remote display on which the AUT is running.

-sync_mode {on | off}

Activates or deactivates synchronization for test execution.
(Default = **on**)

-sync_time

Defines the maximum time (in seconds) thast XRunner waits for a synchronization event.

-t testname

Specifies the name of the test to be loaded in the XRunner window. This can be the name of a test stored in a directory specified in the search path or the full pathname of any test stored in your system.

-td_log_dirname

Determines the directory in which the XRunner logfile resides.

-test_director {on | off}

Activates or deactivates TestDirector integration support.

-timeout *non-negative integer*

Defines the global *timeout* variable (in seconds) used by XRunner.
(Default = **1**[second])

-user_name

Used for TestDirector integration.

-verify *verification_results_name*

Specifies that the test is to be run in Verify mode and assigns a name to the subdirectory in which the test results are stored.

-window_frames {on | off}

Includes or excludes application window frames from bitmap comparison. (Default = **on**)

-wm_borders {on | off}

Determines whether bitmaps are captured with their window frame. (Default = **on**)

28

Running Tests in the Background

XRunner lets you execute tests in the background while you use your mouse and keyboard to develop tests or to work with other applications.

This chapter describes:

- Running a Background Test
- Setting the Background XRunner Startup Options
- Setting the Background Environment Options
- Running Background Tests from the Command Line
- Stopping a Background Run

About Background Testing

Background testing lets you execute a test and continue to use your workstation for other purposes. For example, you can execute a test in background mode while developing a new test in the XRunner window.

Background tests are run in the Background Test window. This is a self-contained testing environment based on Mercury Interactive's Virtual X Server technology. The Background Test window can be iconized while executing a test. Your mouse and keyboard are free, so you can continue working in other windows.

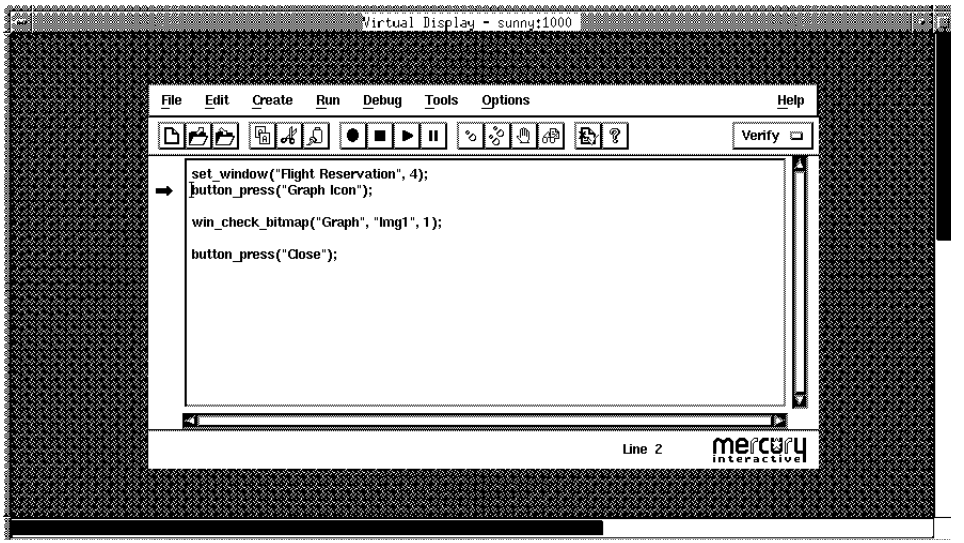
The Background Test form is used to activate and set options for background testing. There are two types of background test parameters: background XRunner start-up options and background environment options. Background XRunner start-up options include the pathname of the test to be executed, the test execution mode, and the command line options.

Background environment options include the window manager and the virtual display ID.

Running a Background Test

To run a test automatically in background mode:

1. Select the Background Run command from the Tools menu. The Background Run form appears.
2. Enter the full pathname of the test you want to open in the Test Pathname field.
3. Select the Run or Quick Run test execution modes.
4. Set additional Startup and Environment options (see the following sections.)
5. Click OK. The Background Run window opens and the test is executed. When the test is completed, the Background Test window closes.



If you select the Idle test run mode from the Background Run form, the Background Test window opens with a background XRunner inside. Use the File and Run menus of the background XRunner to open and execute tests.

You can invoke the AUT by using the **system** function or by typing the following at the UNIX command line:

```
aut -display <displayname>
```

Where *aut* is the application under test and *-displayname* is the name in the banner of the Background Run window.

For more information on using the **system** command to start your application, see Chapter 34, "Initializing Special Configurations."

Setting the Background XRunner Startup Options

Follow the steps below to set the desired start-up options in the Background Run form:

- To select the test run mode, activate the Idle, Quick Run, or Run checkbox.
- For a Verification run, click the Verify check box and enter the name of a results directory in the Name field.
- Enter the command line parameters you want to set in the Command line options field. The command line options **-server** and **-display** cannot be set from this field. For more information about command line options see Chapter 27, "Running Tests from the Command Line."
- Check the Quit Foreground XRunner checkbox if you want to quit XRunner and run only the Background Run window.

Setting the Background Environment Options

The following background environment options can be set in the Background Run form:

- Enter the name of the window manager you want to use in the Background Run window in the Window Manager field. The default option is the Motif window manager.
- Click the Lock Screen checkbox to lock the Background Run window to mouse and keyboard input. This prevents accidental keyboard and mouse

input that may disturb test execution. Tests are executed in batch mode to prevent popup messages.

➤ If you want to run more than one session you must enter a unique display number for each window. Enter the ID number of the Background Run window in the Display ID field. The default is zero.

Running Background Tests from the Command Line

The Background Test window can be opened using the command line interface. Type at the UNIX command line:

```
bg_xrun [-id display_number -wm window manager -lock
        -server -display -command_line_options]
```

-id *display_number*

Defines the display number of the Background Test window. (The actual display number is the ID plus the 1000 base address).
(Default = 0)

-wm *window manager*

Defines the name of the window manager.
(Default = mwm)

- lock

Sets the Lock Screen option.
(Default = no lock)

-server *name*

The name of the workstation on which the Background Run virtual (xgate) window will run and will execute a selected test.

-display *display name*

The display on which the Background Run XRunner window will appear.

- command_line_options *command_line_options*

Sets the command line options.
(Default = no command line options set.)

For more information see Chapter 27, "Running Tests from the Command Line."

The Background Run window opens in Idle mode. To execute a test in Run or Quick Run mode the testname and command line options *-animate (*for a Run) or *-run* (for a Quick Run) must be included in the script.

For example, the following command line would load XRunner and the newclock test on *metal* using the *olwm* window manager. The test would be run on *copper* in virtual window ID *1002*. The test run would lock out all keyboard and mouse input.

```
bg_xrun -id 2 -wm olwm -lock -server copper:0 -display metal:0 -t/v1_3/newclock
-run
```

Stopping a Background Run

If the Lock Screen option was not selected, close the test and then exit XRunner from the File menu of the background XRunner. This will close the Background Run window.

Click the Kill button to stop a background run during execution if the Lock Screen option is set. Test execution stops and the Background Run window closes. The Kill command might not close the AUT.

29

Running Tests on Remote Hosts

This chapter describes how to perform Context Sensitive testing when XRunner and the application under test (AUT) are running on different machines, or on different displays.

This chapter describes:

- Connecting XRunner to a Remote AUT
- Disconnecting XRunner from Applications

About Running Tests on Remote Hosts

XRunner can be connected simultaneously to more than one remote machine. You can also test your application using several local or remote XRunners.

XRunner automatically begins a name server process called *mc_svc* on each host you use to connect XRunner and your AUT. The *mc_svc* process registers current connections between the AUT and XRunner.

To connect XRunner to applications running on a remote host or display, you use an **aut_connect** statement in a test script. To disconnect XRunner from an application running on a remote host or display, use an **aut_disconnect** statement.

There are two additional functions that are useful when XRunner and your AUT are running on different hosts or displays: **aut_set** sets the application with which XRunner will communicate and **aut_get** returns information on the remote AUT connected to XRunner. For more information on the **aut_set** and **aut_get** functions, refer to the *TSL Reference Guide*.

Connecting XRunner to a Remote AUT

To connect XRunner to a remote application, run an **aut_connect** statement in a test script.

The **aut_connect** function has the following syntax:

aut_connect (*host*, *app_name*, *instance_id*, *display*);

For example, to connect XRunner to all applications running on the host called *metal*, and the display *xterm8:0*, run **aut_connect** as follows:

```
aut_connect("metal","AUT","*","xterm8:0");
```

The connection is valid for all AUTs matching the description in the **aut_connect** statement (whether they are already running or not). You do not need to re-execute the **aut_connect** statement each time you start the AUT.

For more information on the **aut_connect** function, refer to the *TSL Reference Guide*.

Disconnecting XRunner from Applications

The **aut_disconnect** function disconnects XRunner from an application running on a remote host.

The **aut_disconnect** function has the following syntax:

aut_disconnect (*[host, app_name, instance_id, display]* **);**

Specifying arguments for the **aut_disconnect** function is optional. If no arguments are specified, **aut_disconnect** disconnects XRunner from *all* applications running on remote hosts.

For example, to disconnect XRunner from an application "WinApp", running on a host called *sunny*, and the display *xterm4:0*, run **aut_disconnect** as follows:

```
aut_disconnect ("sunny","WinApp","*","xterm4:0");
```

In this example, "WinApp" was defined by the MC_AUT_NAME environment variable before starting the application under test. This made is possible to then specify a hypothetical "WinApp" for the *app_name* parameter instead of the default *"AUT"*.

For more details on the **aut_disconnect** function, refer to the *TSL Reference Guide*.

Part VI

Debugging Tests

30

Debugging Test Scripts

Controlling test execution can help you to identify and eliminate bugs in your test scripts.

This chapter describes:

- Running a Single Line of a Test Script
- Pausing Test Execution

About Debugging Test Scripts

After you create a test script you should check that it runs smoothly, without errors in syntax or logic. In order to detect and isolate bugs in a script, you can use the Step and Pause commands to control test execution.

Three Step commands are available:

- The *Step* command lets you run a single line of a test script.
- The *Step Into* command lets you call and display another test or user-defined function.
- The *Step Out* command—used in conjunction with the Step Into command—lets you complete the execution of a called test or user-defined function.

In addition, you can use the Pause command or the **pause** function to temporarily suspend test execution.

You can also control test execution by setting breakpoints. A breakpoint pauses a test run at a predetermined point, allowing you to examine the

effects of the test on your application. For more information, see Chapter 31, "Using Breakpoints."

To help you debug your tests, XRunner allows you to monitor variables in a test script. You define the variables you want to monitor in a Watch List. As the test runs, you can view the values that are assigned to the variables. For more information, see Chapter 32, "Monitoring Variables."

When debugging a test script, run the test in the Debug mode. The results of the test are saved in a *debug* directory. Each time you run the test, the previous debug results are overwritten. Continue to run the test in the Debug mode until you are ready to run the test in Verify mode. For more information on using the Debug mode, see Chapter 24, "Running Tests."

Running a Single Line of a Test Script

You can run a single line of a test script using the Step, Step Into and Step Out commands in the Run menu.

Step Command

The *Step* command executes only the current line of the active test script—the line marked by the execution arrow.

When the current line calls another test or a user-defined function, the called test or function is executed in its entirety but the called test script is not displayed in the XRunner window.

Step Into Command

The *Step Into* command executes only the current line of the active test script. However, in contrast to the Step command, if the current line of the executed test calls another test or a user-defined function:

- The test script of the called test or function is displayed in the XRunner window.
- The called test or function is not executed. Use the Step or Step Out commands to continue test execution.

Step Out Command

You use the *Step Out* command only after entering a test or a user-defined function using the Step Into command. The Step Out command executes to the end of the called test or user-defined function, returns to the calling test, and then pauses test execution.

Pausing Test Execution

You can temporarily suspend test execution by selecting the Pause command or by adding a **pause** statement to your test script.

Pause Command

You can suspend the execution of a test by choosing the Pause command in the Run menu or pressing the PAUSE softkey. A paused test stops running when all previously interpreted TSL statements have been executed. Unlike the Abort command, the Pause command does not initialize test variables and arrays.

To resume execution of a paused test, select the appropriate Run command in the Run menu. The test run continues from the point that you invoked the Pause command, or from the execution arrow if you moved it while the test was suspended.

The pause Function

When XRunner processes a **pause** statement in a test script, test execution halts and a message box is displayed. If the **pause** statement includes an expression, the result of the expression appears in the message box. The syntax of the **pause** function is:

pause (*[expression]***);**

In the following example, the **pause** function suspends test execution and displays the time that elapsed between two points.

```
t1=get_time();
t2=get_time();
pause ("Time elapsed is" & t2-t1);
```

For more information on the **pause** function, see the *TSL Reference Guide*.

31

Using Breakpoints

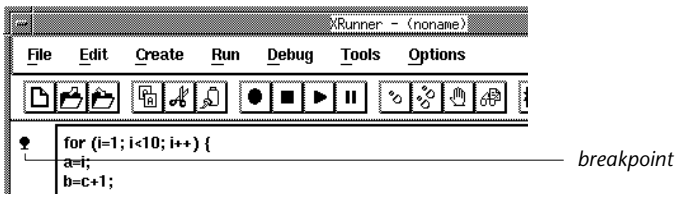
A breakpoint marks a place in the test script where you want to pause a test run. Breakpoints help to identify bugs in a script.

This chapter describes:

- Breakpoint Types
- Setting Break at Line Breakpoints
- Setting Break in Function Breakpoints
- Modifying Breakpoints
- Deleting Breakpoints

About Breakpoints

By setting a breakpoint you can stop a test run at a specific place in a test script. A breakpoint is indicated by a breakpoint marker in the left margin of the test window.



XRunner pauses test execution when it reaches a breakpoint. You can examine the effects of the test run up to the breakpoint, make any necessary changes, and then restart the test from the breakpoint. Use the Run from

Arrow command to restart the test run. Once restarted, the test continues until the next breakpoint is encountered, or the test is completed.

You can use breakpoints to:

- suspend test execution and inspect the state of your application.
- monitor the entries in the Watch List. For more information, see Chapter 32, "Monitoring Variables."
- mark a point from which to begin stepping through a test script using the Step commands. For more information, see Chapter 30, "Debugging Test Scripts."

Two types of breakpoints are available: Break at Line and Break in Function. A Break at Line breakpoint stops a test at a specified line number in a test script. A Break in Function breakpoint stops a test when it calls a specified user-defined function in a loaded compiled module.

You set a pass count for each breakpoint you define. The pass count determines the number of times the breakpoint is passed before it stops the test run. For example, suppose you program a loop that performs a command twenty-five times. By default the pass counter is set to zero, so test execution stops after each loop. If you set the counter to 25, execution stops only after the twenty-fifth iteration of the loop.

Note: The breakpoints you define are active only during your current XRunner session. If you terminate your XRunner session, you have to redefine breakpoints to continue debugging the script in another session.

Breakpoint Types

XRunner allows you to set two types of breakpoints: Break at Line and Break in Function.

Break at Line

A Break at Line breakpoint is defined by a test name and a test script line number. The breakpoint marker appears in the left margin of the test script, opposite the specified line. A Break at Line breakpoint might, for example, appear in the Breakpoints form as:

```
ui_test[137] : 0
```

This means that the breakpoint marker appears in the test named ui_test at line 137. The number after the colon represents the pass counter, here set to zero (the default). This means that the test will stop every time the breakpoint is passed.

Break in Function

A Break in Function breakpoint is defined by the name of a user-defined function and the name of the compiled module in which the function is located. When you define a Break in Function breakpoint, the breakpoint marker appears in the left margin of the XRunner window, opposite the first line of the function. XRunner halts the test run every time the specified function is called. A Break in Function breakpoint might, for example, appear in the Breakpoints form as:

```
ui_func [ui_test : 25] : 10
```

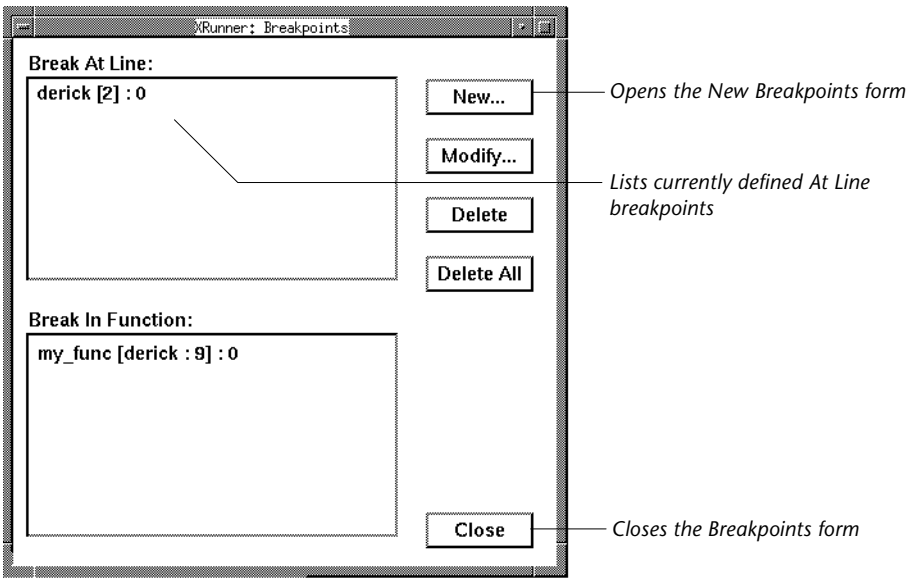
This indicates that a breakpoint has been defined for the line containing the function ui_func, in the compiled module ui_test: in this case line 25. The pass counter is set to 10, meaning that the test will be stopped each time the function has been called ten times.

Setting Break at Line Breakpoints

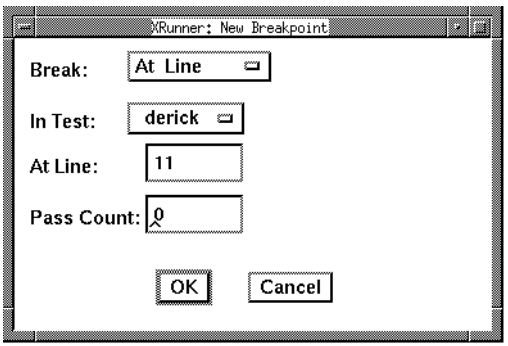
You set Break at Line breakpoints using the Breakpoints form, the mouse, or the Toggle Breakpoint command.

To set a Break at Line breakpoint using the Breakpoints form:

1 Select Breakpoints from the Debug menu to open the Breakpoints form.



2 Click New to open the New Breakpoint form.



3 Select At Line from the Break toggle options. Select the test name from the In Test list. Modify the line number and pass count as required.

4 Click OK to set the breakpoint and close the form. The new breakpoint is displayed in the Break at Line list in the Breakpoints form.

The breakpoint marker appears in the left margin of the test script, opposite the specified line.

5 Click Close to close the Breakpoints form.

To set a Break at Line breakpoint using the mouse:

1 Move the mouse pointer to the left margin of the XRunner window, to the line in the test script at which you want test execution to stop.

2 Click the right mouse button. The breakpoint symbol appears in the left margin of the XRunner window.

To remove the breakpoint, click on the breakpoint symbol with the right mouse button or select the Toggle Breakpoint command.

To set a Break at Line breakpoint using the Toggle Breakpoint command:

1 Move the insertion point to the line of the test script where you want test execution to stop.

2 Activate the Toggle Breakpoint command by selecting it from the Debug menu. The breakpoint symbol appears in the left margin of the XRunner window.

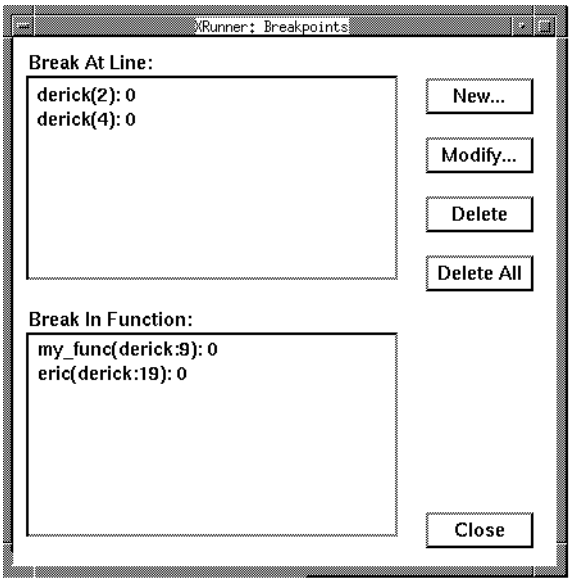
To remove the breakpoint, click on the breakpoint symbol with the right mouse button or select the Toggle Breakpoint command again.

Setting Break in Function Breakpoints

A Break in Function breakpoint stops test execution at the user-defined function that you specify. You can set a Break in Function breakpoint using either the Breakpoints form or the Break in Function command.

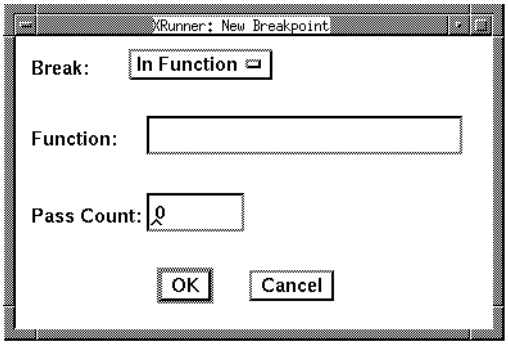
To set a Break in Function breakpoint using the Breakpoints form:

1 Select Breakpoints from the Debug menu to open the Breakpoints form.



2 Click the New button to open the New Breakpoint form.

3 Select In Function from the Break toggle options. The form changes so that you can type in a function name and a pass count value.



4 Enter the name of a user-defined function in the Function field. The function must be compiled by XRunner. For more information, see Chapter 20, “Creating User-Defined Functions” and Chapter 21, “Creating Compiled Modules.”

5 Enter a value in the Pass Count field.

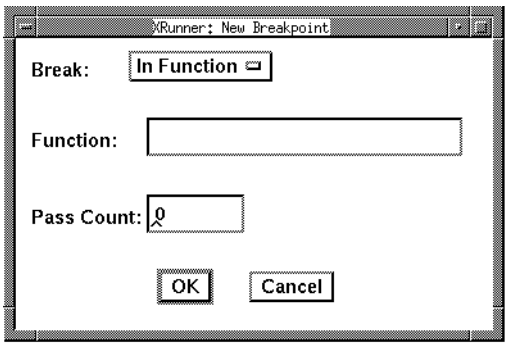
6 Click the OK button to close the form.

7 The new breakpoint is displayed in the Break in Function list of the Breakpoints form. Click Close to close the form.

The breakpoint symbol appears in the left margin of the XRunner window.

To set a Break in Function breakpoint using the Break in Function command:

1 Choose the Break in Function command from the Debug menu. The New Breakpoint form opens.



2 Enter the name of a user-defined function in the Function field. The function must be compiled by XRunner. For more information, see Chapter 20, "Creating User-Defined Functions" and Chapter 21, "Creating Compiled Modules."

3 Enter a value in the Pass Count field.

4 Close the form by clicking OK. The breakpoint symbol appears in the left margin of the XRunner window.

5 To close the Breakpoints form, click Close.

Modifying Breakpoints

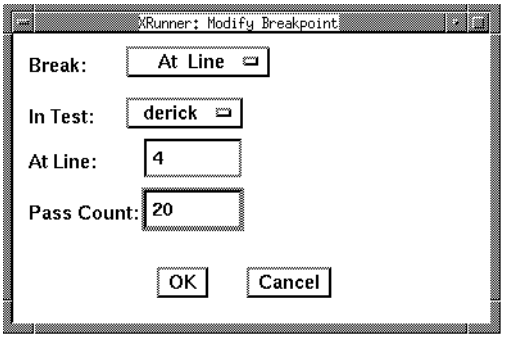
You can modify the definition of a breakpoint using the Breakpoints form. You can change the breakpoint's type, the test or line number for which it is defined, and the value of the pass counter.

To modify a breakpoint:

1 Select Breakpoints from the Debug menu to open the Breakpoints form.

2 Choose a breakpoint in one of the lists by clicking on it.

3 Click Modify to open the Modify Breakpoint form.



4 To change the type of Breakpoint, select a breakpoint type from the Break toggle options.

To select another test, select the name of a test from the In Test option list.

To change the line number in which the breakpoint will appear, enter a new value in the At Line field.

To change the Pass Count, enter a new value in the Count field.

5 Click OK to close the form.

Deleting Breakpoints

You can delete a single breakpoint or all breakpoints defined for the current test using the Breakpoints form.

To delete a single breakpoint:

1. Select Breakpoints from the Debug menu to open the Breakpoints form.
2. Select a breakpoint in either the Break at Line or the Break in Function field.
3. Click Delete. The breakpoint is removed from the list.
4. Click Close to close the Breakpoints form.

   Note that the breakpoint symbol is removed from the left margin of the XRunner window.

To delete all breakpoints:

1. Open the Breakpoints form.
2. Click the Delete All button. All breakpoints are deleted from both lists.
3. Click Close to close the form.

   Note that all breakpoint symbols are removed from the left margin of the XRunner window.

32

Monitoring Variables

The Watch List displays the values of variables, expressions and array elements during a test run. You use the Watch List to speed up the debugging process.

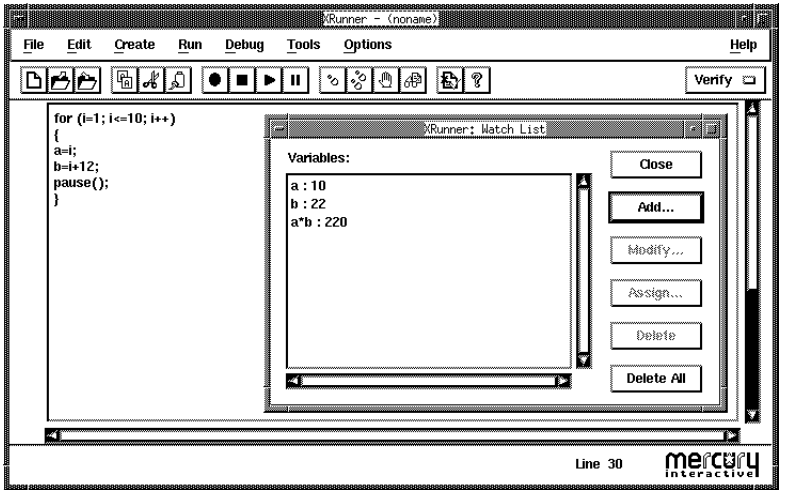
This chapter describes:

- Adding Variables to the Watch List
- Viewing Variables in the Watch List
- Modifying Variables in the Watch List
- Assigning a Value to a Variable in the Watch List
- Deleting Variables from the Watch List

About Monitoring Variables

The Watch List lets you monitor the values of variables, expressions and array elements while you debug a test script. You simply add the elements that you want to monitor to the Watch List. During a test run, you can view

the current values at each break in execution—such as after a Step command, at a breakpoint, or at the end of a test.



For example, in the following test, the Watch List is used to measure and track the values of variables *a* and *b*. After each loop is executed, the test pauses so that you can view the current values.

```
for (i = 1; i <= 10; i++)
{
a = i;
b = i+12;
pause ();
}
```

After XRunner executes the first loop, the test pauses and the variables and their values are displayed in the Watch List as follows:

```
a:1
b:13
```

When XRunner completes the test run, the Watch List shows the following results:

```
a:10
b:22
```

If a test script has several variables with the same name but different scopes, the variable is evaluated according to the current scope of the interpreter. For example, suppose both *test_a* and *test_b* use a static variable *x*, and *test_a* calls *test_b*. If you include the variable *x* in the Watch List, the value of *x* displayed at any time depends on whether XRunner is interpreting *test_a* or *test_b*.

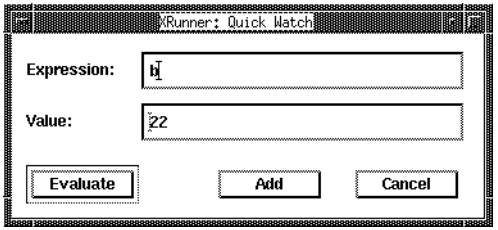
If you select a test from the Calls list (Debug > Calls), the context of the variables and expressions in the Watch List changes. XRunner automatically updates their values in the Watch List.

Adding Variables to the Watch List

You add variables, expressions, and arrays to the Watch List using the Quick Watch form.

To add a variable, an expression, or an array to the Watch List:

1 Select Debug > Quick Watch to open the Quick Watch form.



2 In the Expression field, enter the variable, expression, or array that you want to add to the Watch List.

3 Click Evaluate to see the current value of the new entry. If the new entry contains a variable or an array that has not yet been initialized, the message "<cannot evaluate>" is displayed in the Value field. The same message is displayed if you enter an expression that contains an error.

4 Press Add. The Quick Watch form closes and the new entry appears in the Watch List.

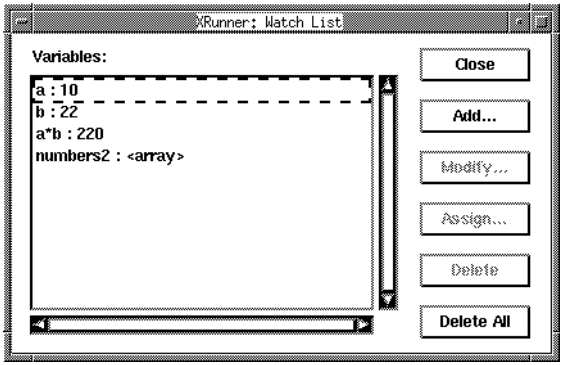
Note: Do not add expressions to the Watch List that assign or increment the value of variables; this can affect test execution.

Viewing Variables in the Watch List

Once variables, expressions, and arrays have been added to the Watch List, you can use the Watch List to view their values.

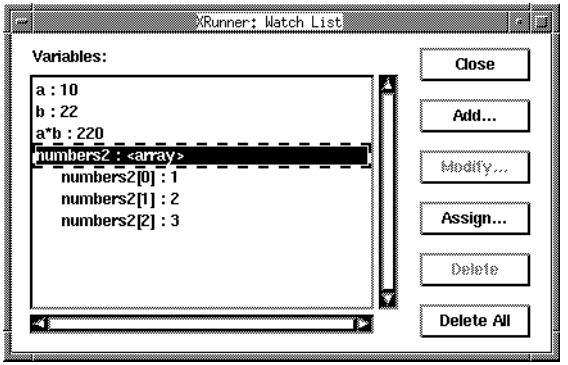
To view the values of variables, expressions and arrays in the Watch List:

1 Select Debug > Watch List to open the Watch List form.



2 The variables and expressions and arrays are displayed; current values appear after the colon.

3 To view values of array elements, double-click on the array name. The elements and their values are displayed below the array name. Double-click on the array name to hide the elements.



4 Click Close to exit the form.

Modifying Variables in the Watch List

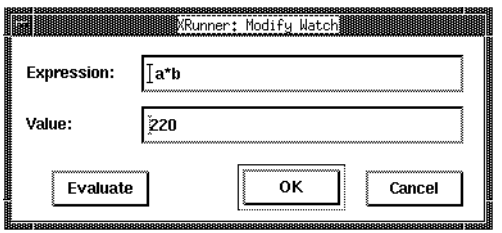
You can modify variables and expressions in the Watch List using the Modify Watch form. For example, you can turn variable *b* into the expression *b + 1*, or you can change the expression *b + 1* into *b * 10*. When you close the Modify Watch form, the Watch List is automatically updated to reflect the new value for the expression.

To modify an expression in the Watch List:

1 Select Debug > Watch List to open the Watch List form.

2 Select the variable or expression that you want to modify.

3 Click Modify to open the Modify Watch form.



4 Change the expression in the Expression field as needed. Click the Evaluate button to see the value of the modified expression. The new value of the expression appears in the Value field.

5 Click OK to close the Modify Watch form. The modified expression and its new value appear in the Watch List.

Assigning a Value to a Variable in the Watch List

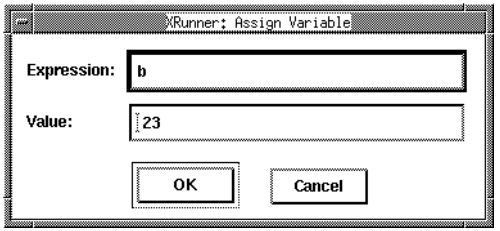
You can assign new values to variables and array elements in the Watch List. For example, variable *b* with the value 2 can be assigned the value 10. Values can be assigned only to variables and array elements, not to expressions.

To assign a value to a variable or an array element:

1 Select Debug > Watch List to open the Watch List form.

2 Click on the variable or array element to which you want to assign a value.

3 Click Assign to open the Assign Variable form.



4 Enter the new value for the variable or array element in the New Value field.

5 Click OK to close the form. The new value is displayed in the Watch List.

Deleting Variables from the Watch List

You can delete selected variables, expressions and arrays from the Watch List, or you can delete all the entries in the Watch List.

To delete a variable, an expression, or an array:

1. Select Debug > Watch List to open the Watch List form.
2. Click on the variable, expression or array that you want to delete.

Note: An array can be deleted only if its elements are hidden. To hide the elements of an array, double-click on the array name in the Watch List.

---

3. Click Delete to remove the entry from the list.
4. Click Close to close the Watch List form.

To delete all entries in the Watch List:

1. Open the Watch List form.
2. Click Delete All. All entries are deleted.
3. Click Close to close the form.

Part VII

Configuring XRunner

33

Changing System Defaults

This chapter lists XRunner configuration parameters and describes how you can change their default values.

This chapter describes:

- Configuration Files
- Modifying Configuration Settings from the Configuration Form
- Modifying Configuration Settings from a Test Script
- Environment Variables
- Configuration Parameters
- Configuration File Contents

About Changing System Defaults

XRunner system configuration parameters are defined in the *xrunner.cfg* file, which is located under */dat* in your installation directory. Each parameter affects a specific XRunner function.

You can adapt XRunner to your testing environment by modifying the default values of the system parameters. There are three methods for changing system defaults:

- **Using the XRunner Configuration form:** display XRunner configuration parameters by selecting Configure from the Options menu. You can apply changes you make to the current session, or save them for the current and future sessions.

- Using a setvar function in a test script: program a **setvar** TSL command, passing the configuration parameter and modified value as arguments. The new setting takes effect immediately but is not automatically saved for future sessions.
- Modifying settings in your .xrunner file: open the file in any text editor, make the necessary changes and save the file. Changes take effect the next time you start XRunner.

Configuration Files

XRunner has three levels of configuration files:

- The bottom-level file, xrunner.cfg is a system-wide configuration file. It is created by the XRunner installation program, and is normally maintained by the system administrator. The xrunner.cfg file resides in the /dat subdirectory of the XRunner installation directory.

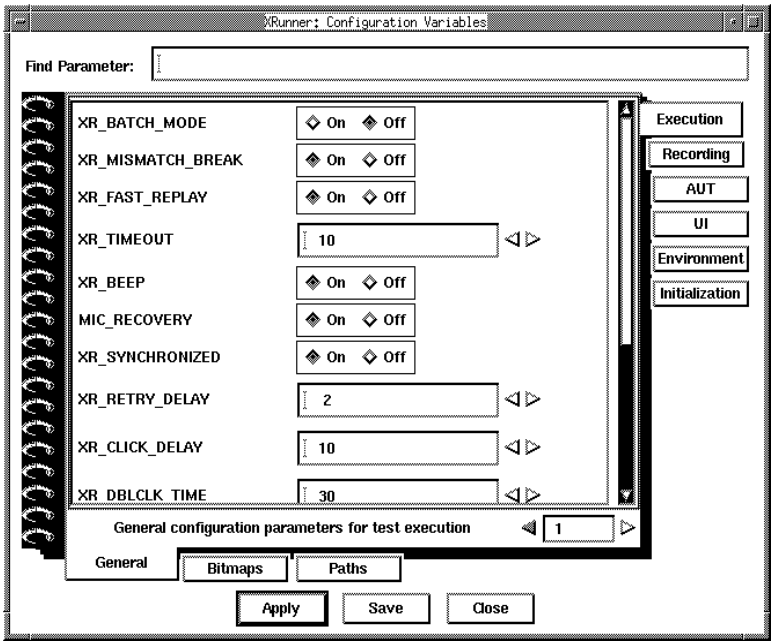
  Included with xrunner.cfg is the machine.cfg file. This file holds all platform-dependent items (such as softkey configuration). It is automatically copied to the correct machine version by the XRunner installation script.
- The middle-level configuration file is optional. An environment variable, XR_CFG_FILE, designates the name and location of this file. This file can be used to set values specific to a group of users testing the same application.
- The top-level configuration file (optional) is the *.xrunner* file stored in your home directory. This file can be used to tailor XRunner to your individual needs.

When you invoke XRunner, these configuration files are read sequentially. If the same configuration parameter is assigned a value by more than one of these files, the value set for this variable is the one specified in the highest level configuration file.

System configuration files assign values to parameters which affect specific XRunner functions. This chapter describes the configuration parameters and their default values, and provides an overview of the syntax and data types of the configuration files.

Modifying Configuration Settings from the Configuration Form

The Configuration form displays all the settings for the configuration parameters in the .xrunner file. The form provides you with a convenient method for making changes to system defaults, without having to open the .xrunner configuration file. Parameters are arranged in the form by category and subcategory to make your work easier. The form also has a built-in search facility for finding parameters.



Using the Configuration form, you can apply configuration changes for the current or future sessions.

- Click Apply to apply a change for the current XRunner session and to leave the form open.
- Click Save to save the changes to the .xrunner configuration file. Changes apply for the current and future sessions. The Configuration form closes.

➤ Click Close to close the Configuration form without applying any changes.

Note: XRunner records changes saved to the .xrunner file in a special reserved area appearing below the line:

Reserved: Please do not write beyond this line

Since previous values are overwritten each time the .xrunner file is updated, it is recommended you insert personal comments above the line.

Finding Parameters

There are three ways to locate a specific parameter:

➤ The Configuration form is organized according to categories and subcategories. Select a category (from among the major tabs displayed on the right side of the notebook) and a subcategory (from among the minor tabs displayed at the bottom of the notebook), then scroll down the page if necessary until you find the desired parameter.

➤ Use the spinbox at the lower right corner to scroll the pages until you find the parameter you need.

➤ Perform a quick search by entering a string or part of a string in the Find Parameter field at the top of the form. Press Enter. XRunner opens the notebook at the relevant page and highlights the requested parameter. If there is more than one parameter that matches the string, you can continue to press Enter until XRunner displays the parameter you need.

Parameter Categories

There are 12 pages to the Configuration Notebook. There are six categories and 10 subcategories of parameters:

Execution

This category includes the parameters controlling the way XRunner runs tests. These are the parameters you generally modify the most. They include the following subcategories:

- General test execution parameters, for example, XR_TIMEOUT for setting the global timeout (in seconds) used by XRunner
- Bitmap parameters control the display of bitmaps during test execution, for example, XR_RAISE_WINDOWS for bringing the bitmap to be checked to the front of the screen display during the test run.
- Path parameters, for example, XR_SHARED_CHECKLIST_DIR which sets the directory in which XRunner stores shared checklists for GUI checkpoints. Note that changes made to path parameters take effect only from the next time you invoke XRunner.

Recording

This category includes:

- Class Record Attribute parameters, for example, XR_WINDOW_REC_ATTR that defines the way XRunner records and learns objects of the window class.
- Class Record Method parameters, for example, XR_PBUTTON_REC_METHOD that defines the attributes XRunner records and learns for the push_button class of objects.

AUT

This category includes parameters that control aspects of your application.

They include:

- AUT Customization parameters, for example, MIC_BUTTON_CLICK_LOCATION that defines the location of the mouse-click performed on button objects during test execution Analog mode.
- Text Checkpoint parameters, for example, XR_FONT_GROUP that sets the default active font group for defining checkpoints in a test script.
- Keyboard parameters, for example, MIC_CLICK_BUTTON that defines which mouse button is used to perform mouse clicks.

UI

These parameters define aspects of the XRunner User Interface, for example, XR_HIDE_BUBBLE_HELP which you use to display or hide XRunner's bubble help.

Environment

This category contains parameters which control your test execution environment, for example, Expected Directory which defines the directory XRunner is currently using to save expected results for the next test run.

Initialization

Initialization parameters could also be called "hardware parameters". They define the configuration of sofkeys and input devices and are the least likely to change from one session to another.

They include:

- Softkey parameters, for example, XR_SOFT_PAUSE which defines the key combination that executes a Pause command during a test run.
- Input Device parameters, for example, XR_INP_KBD_NAME which designates the path and name of the keyboard definition file.

Modifying Configuration Settings from a Test Script

The **setvar** function allows you to modify configuration settings from within a test script. This enables you to control when the value of a configuration parameter affects the test run. For example, you can set different run speeds for different sections of the script.

You can retrieve the current value of a configuration parameter using the **getvar** function. By using a combination of **setvar** and **getvar** statements in a test script, you can control how XRunner executes a test.

setvar

You use the **setvar** function to modify a configuration setting from within the test script. This function has the syntax:

setvar (*parameter, value* **);**

In this function, *parameter* may specify any configuration parameter.

For example:

```
setvar ("XR_MISMATCH_BREAK", "Off");
```

disables the break on mismatch mechanism. The new setting remains in effect during the testing session until it is changed again, either from the Configuration form or with another **setvar** statement.

getvar

You use the **getvar** function to retrieve the current value of a configuration parameter. The syntax of this statement is:

user_variable = **getvar (***parameter* **);**

In this function, *parameter* may specify any configuration parameter.

For example:

```
nowdelay = getvar ("XR_CLICK_DELAY");
```

Assigns the current value of the click delay (the interval that XRunner waits after inputting a single click during a test run) to the user-defined variable nowdelay.

Note that some configuration parameters are set by XRunner and cannot be changed using **setvar** or the Configuration form. For example, the value of the Test Name parameter is always the name of the current test. Use the **getvar** function to retrieve these read-only values.

Controlling Test Execution with setvar and getvar

You can use a combination of **getvar** and **setvar** statements to control test execution. For example, in the following test script fragment, XRunner checks the bitmap Img1. The **getvar** and **setvar** functions are used to control the value of the XR_TIMEOUT and XR_RETRY_DELAY configuration parameters. The **getvar** statement is used to retrieve their current values of these parameters and **setvar** is used to assign them new values for this particular **win_check_bitmap** statement. After the window is checked, **setvar** is used to return the *parameters* to their original values.

```
t = getvar ("XR_TIMEOUT");
d = getvar ("XR_RETRY_DELAY");
setvar ("XR_TIMEOUT", 30);
setvar ("XR_RETRY_DELAY", 3);
win_check_bitmap ("calculator", Img1, 2, 261,269,93,42);
setvar ("XR_TIMEOUT", t);
setvar ("XR_RETRY_DELAY", d);
```

Environment Variables

You define environment variables before invoking XRunner. You must define the variables in the same shell that you invoke XRunner. They are not part of the XRunner configuration and are not saved from session to session.

M_ROOT = ***installation directory***

Determines the directory in which your XRunner installation files reside.

XR_TSL_INIT = *startup test pathname*

Designates the path and name of a startup test file. Use a startup test to configure recording, load compiled modules, and load GUI map files when invoking XRunner.

(Default= $M_ROOT/dat/tslinit)

MC_AUT_NAME

Defines an application name for the Mercury Communication Server (mc_svc). When the variable has been defined, you can use the **aut_connect** and **aut_set** functions with any name you define for your application, instead of using the default "AUT". This parameter should be defined in the same shell from which you run your application.

MIC_TOOLKIT

Defines the type of toolkit used by the tested application. By default this variable is set to *Motif*. If your application uses XView, you should set this variable to *XView*; if your application uses Olit, set it to *Olit;* for Open Interface set it to *Open_interface*.

XRUN_TEST_EXT

Defines the extension to be used by XRunner for record and test execution. By default, XRunner searches for one of the following extensions (in the following order) and uses the first one that is found: Record (R6), XTrap, XTestExtension1, and XTEST.

XRUN_UI

Activates XRunner's user-defined user interface feature. In order to modify the XRunner menu bar or to program a dialog box to appear during interactive test execution, XRUN_UI must be defined and set to any value (such as 1 or On) before XRunner is started.

XRUN_WM

Defines the window manager used by the application. This environment variable should be defined only if you are using one of the following window managers: *xuwm*, *dxwm*, *decmwm* or *fvwm*.

Configuration Parameters

The configuration parameters described in this section are organized according to the categories in the Configuration form. Note, however, that a few configuration parameters are not included in the form and must be modified using a **setvar** command, or manually in your .xrunner file. Parameters that do not appear in the form are indicated where applicable.

If a value for a parameter is not specified in the middle- or top-level configuration file, then XRunner uses system default values defined in the *xrunner.cfg* file created by the XRunner installation program.

Test Execution Parameters

XR_AUTO_LOAD = {On|Off}

Activates or deactivates automatic loading of the temporary GUI map file when XRunner is invoked.

(Default = On)

[Note that the value of this parameter cannot be set using the Configuration form.]

XR_BEEP = {On|Off}

Activates or deactivates the system beep that is produced whenever a checkpoint or an error occurs.

(Default = On)

XR_BATCH_MODE = {On|Off}

Activates or deactivates XRunner's batch mode. When batch mode is activated, XRunner suppresses messages during a test run, so that a test can run unattended.

(Default = Off)

XR_CLICK_DELAY = *integer*

Sets the interval, in tenths of a second, that XRunner waits after inputting a single click during a test run. During a fast (default) test run, using a longer setting ensures that two consecutive single clicks are not misinterpreted as a double-click. Note that this does not apply to double-clicks. If a double-click is recorded, it executes as a double-click regardless of the value of the XR_CLICK_DELAY configuration parameter.

(Default = 10 [tenths of a second])

XR_DBLCLK_TIME = *integer*

Defines the maximum permitted interval, in tenths of a second, that can elapse between two clicks, and still constitute a double-click. It is advised to make the setting consistent with your system default. The minimum value is 10 (tenths of a second).

(Default = 30 [tenths of a second])

XR_FAST_REPLAY = {On|Off}

Sets the default test run speed.

(Default = On, fast run speed)

XR_FOCUS_DELAY = *integer*

Defines the amount of time, in tenths of a second, that XRunner waits during execution from the time the mouse is moved to a new window until input is entered. This is particularly important when XR_FAST_REPLAY is set to

On, since it ensures that XRunner does not send keystrokes to a new window before it is ready to receive them.

(Default = 10 [tenths of a second])

XR_KBD_DELAY = *integer*

Sets the interval, in tenths of a second, that XRunner waits after inputting a single keyboard event during a test run.

(Default = 0)

XR_KEY_EDITING = {On|Off}

Activates or deactivates key editing. When activated, XRunner generates more concise **type** statements, representing only the net result of pressing and releasing input keys. This makes your test script easier to read. Whenever the exact order of keystrokes is important for your test, you should disable key editing.

For example, typing the letter "A" with key editing off produces the following statement:

```
type ("<kShift>-a-<kShift>+a+");
```

With key editing on, the statement is:

```
type ("A");
```

For more information on key editing, see the **type** function in the *TSL Reference Guide*.

(Default = On)

XR_MISMATCH_BREAK = {On|Off}

Activates/deactivates the break on mismatch mechanism. When active, XRunner pauses execution and displays a message whenever a mismatch occurs or a Context Sensitive function fails during a verification run. This parameter should be used only when working interactively.

(Default = On)

XR_WM_OFFSET_X and XR_WM_OFFSET_Y = *integer*

Specifies an offset to be applied whenever a window is moved by a **wait_window** or **check_window** command. If the upper left corner of the window is designated in the command by the coordinates x,y specifying an offset causes the window to be moved to the position x+wm_offset_x, y+wm_offset_y. (Negative offsets are also accepted.)

This parameter is useful only for applications running under the twm window manager.

(Default = 0)

[Note that the value of this parameter cannot be set using the Configuration form.]

XR_RETRY_DELAY = *integer*

Instructs XRunner to determine whether a window is stable before capturing it for a bitmap checkpoint or a synchronization point during a test run. To be declared stable, a window must not change between two consecutive samplings. For example, when XR_DELAY is two seconds and XR_TIMEOUT is ten seconds, XRunner checks the window in the application under test every two seconds until two consecutive checks produce the same results or until ten seconds have elapsed. Setting the value to 0 disables all image checking.

(Default = 1[second])

XR_SCR_REDRAW_TIME = *integer*

Sets the time interval (in seconds) that XRunner waits for the entire screen to be redrawn after it moves a window during a verification run.

(Default = 10 [seconds])

XR_SYNC_TIME = *integer*

Determines the maximum amount of time (in seconds) that the system waits for an expected synchronization event. When an event is not received, XRunner waits up to the predefined XR_SYNC_TIME (in seconds) and then continues execution. If this time is exceeded, the run continues after a slight delay.

(Default = 10 [seconds])

[Note that the value of this parameter cannot be set using the Configuration form.]

XR_SYNCHRONIZED = {On|Off}

Determines whether XRunner monitors X server messages and sends input to the AUT only when it is ready to receive it. If you set this parameter to Off, the reliability of test execution may be impaired.

(Default = On)

XR_TIMEOUT = *integer*

Sets the global timeout (in seconds) used by XRunner. This value is added to the time parameter imbedded in GUI checkpoint or synchronization point statements to determine the maximum amount of time that XRunner searches for the specified window.

The maximum time is calculated by adding the *time* parameter of the statement to the value set for the XR_TIMEOUT parameter.

For example, in the statement:

```
win_check_bitmap ("calculator", Img1, 2, 261, 269, 93, 42);
```

when the XR_TIMEOUT parameter is set to 10 seconds, this operation takes a maximum of 12 (2+10) seconds.

(Default = 10[seconds])

Bitmap Parameters

XR_COMPRESS = {On|Off}

Activates or deactivates compression of captured bitmaps. Capture and display of compressed bitmaps is slightly slower, yet file size is significantly reduced.

(Default = On)

XR_WINDOW_FRAMES = {On|Off}

Determines whether the window frame is included in bitmap comparison. When activated, window frames are compared. When deactivated, only the window contents are compared. (In both cases, window frames are included

in the captured window.) Note that this parameter has no influence on bitmap comparison when the XR_WM_BORDER parameter is set to Off.

(Default = On)

XR_WM_BORDER = {On|Off}

Determines whether windows are captured with the window frame. When activated, the window frame is captured.

(Default = On)

XR_MOVE_WINDOWS = {On|Off}

Causes XRunner to automatically return the opened window to the location specified in any GUI checkpoint or synchronization point statement that references this window. Note that if you set this parameter to Off, you should take measures to ensure that during a test run, windows open in the correct, previously-recorded position.

(Default = On)

XR_RAISE_WINDOWS = {On|Off}

Brings the window to be checked during a bitmap checkpoint or synchronization point to the front of the screen display during a test run.

(Default = On)

XR_MIN_DIFF = *integer*

Defines the number of pixels that constitute the threshold for bitmap mismatch comparison. When this value is set to 0, a single pixel mismatch constitutes a window mismatch.

(Default = 0 [pixels])

XR_IMAGE_MODE = *image mode*

Defines the mode that XRunner uses to capture, verify and display bitmaps. By default, XR_IMAGE_MODE is set to xwd. The two other valid values are gl and external. If XR_IMAGE_MODE is set to either xwd or gl, XRunner uses its own built-in code to capture, verify and display bitmaps. If XR_IMAGE_MODE is set to external, XRunner uses all three external utilities that you provide to capture, verify and display bitmaps. For more information, see Appendix C, "External Utilities for Bitmap Capture/Check/Display."

(Default = xwd)

XR_CAPTURE_UTIL = *external utility pathname*

Defines the path of the external utility that XRunner uses to capture bitmaps. Note that XR_CAPTURE_UTIL is effective only if the XR_IMAGE_MODE configuration parameter is set to "external".

XR_DISPLAY_UTIL = *external utility pathname*

Defines the path of the external utility that XRunner uses to display bitmaps. Note that XR_DISPLAY_UTIL is effective only if the XR_IMAGE_MODE configuration parameter is set to "external".

XR_COMPARE_UTIL = *external utility pathname*

Defines the path of the external utility that XRunner uses to compare bitmaps during test runs. Note that XR_COMPARE_UTIL is effective, only if the XR_IMAGE_MODE configuration parameter is set to "external".

XR_VERIFY_UTIL = *external utility pathname*

Defines the path of the external utility that XRunner uses to verify bitmaps. Note that XR_VERIFY_UTIL is effective only if the XR_IMAGE_MODE configuration parameter is set to "external".

Path Parameters

XR_AUTO_LOAD_DIR = *directory pathname*

Determines the directory in which the temporary GUI file (*temp.gui*) resides. When you activate automatic loading of the temporary GUI file, this is the directory from which the file is loaded.

(Default is your home directory)

XR_AUTOMOUNT_MAP = *map file pathname*

Points to the automount map file.

(Default = $M_ROOT/dat/automount.map)

XR_GLOB_FILTER_LIB = *library pathname*

Identifies the directory in which the Global Filter Library resides. When you invoke XRunner, this library becomes the active Global Filter Library.

(Default = $M_ROOT/filters)

XR_SEARCH_PATH = *directory pathname(s)*

Sets the directory paths that XRunner searches for called tests. Any test stored in a directory that is specified by this parameter can be opened or called.

Identify each directory by its full logical pathname. A space delimits between the pathnames of two different directories. The order in which directories are listed determines the order in which they are searched for the specified test.

You can view the directory paths currently defined for XR_SEARCH_PATH by choosing Search Path from the Options menu.

(Default = Current Directory and installation directory\lib)

[Note that the value of this parameter cannot be set using the Configuration form.]

XR_SHARED_CHECKLIST_DIR = *pathname*

Sets the directory in which XRunner stores shared checklists for GUI checkpoints. These are checklists that you define as "shared" when you create them using the Check GUI form. In the test script, shared checklist files are designated by SHARED_CL before the file name in a **win_check_gui**, **obj_check_gui**, or **check_gui** command.

(Default = $M_ROOT/chklist)

XR_TMPDIR = *pathname*

Sets the directory in which XRunner stores temporary tests. Each temporary test is assigned a name having the format: *nonam***, where each asterisk is a digit. These tests are created when you choose New from the File menu, and are automatically deleted unless explicitly saved. The directory designated by this parameter should have at least five megabytes of storage available for these temporary tests. Note that this parameter takes precedence over the TMP environment variable.

(Default = /tmp)

XR_FILE_LOCKING = {On|Off}

Parameter to be used when running XRunner on SunOS 4.1. SunOS 4.1 has a documented bug with file-locking over the NFS. Locking a file on an NFS-mounted partition may hang the file server. If you are running on this version of the SunOS, you MUST set XR_FILE_LOCKING to Off. This bug was fixed in SunOS 4.1.1

Dec workstations might have the same problem. In the event of file-locking, set XR_FILE_LOCKING to Off.

(Default = On)

[Note that the value of this parameter cannot be set using the Configuration form.]

Class Record Method

The following parameters allow you to specify how mouse clicks/drags or keystrokes on objects of a particular class are recorded. For a list of the different record methods, see Chapter 6, "Configuring the GUI Map."

XR_WINDOW_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on windows.

(Default = MIC_RECORD_CS)

XR_CBUTTON_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on check buttons.

(Default = MIC_RECORD_CS)

XR_RBUTTON_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on radio buttons.

(Default = MIC_RECORD_CS)

XR_PBUTTON_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on push buttons.

(Default = MIC_RECORD_CS)

XR_MENU_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on menus.

(Default = MIC_RECORD_CS)

XR_SCROLL_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on scroll bars.

(Default = MIC_RECORD_CS)

XR_LIST_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on lists.

(Default = MIC_RECORD_CS)

XR_SPIN_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on spinboxes.

(Default = MIC_RECORD_CS)

XR_EDIT_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on edit objects.

(Default = MIC_RECORD_CS)

XR_NOTEBOOK_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on notebooks.

(Default = MIC_RECORD_CS)

XR_STATIC_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on static text objects.

(Default = MIC_RECORD_CS)

XR_OBJ_REC_METHOD = *record_method*

Defines the record method for mouse clicks/drags or keystrokes on an object in the general Mercury object class.

(Default = MIC_RECORD_CS)

Class Record Attributes

The following parameters define the set of attributes XRunner identifies while learning a specific class of GUI objects.

➤ When modifying settings for these parameters in the .xrunner file, use the following syntax:

parameter = obligatory_attribute$_1$ obligatory_attribute$_2$ ... obligatory_attribute$_n$,
optional_attribute$_1$ optional_attribute$_2$ ... optional_attribute$_n$,
selector.

For example:

XR_STATIC_REC_ATTR = class X_name, label class_index, location

➤ To modify settings for these parameters from a test script, use the **set_record_attr** TSL function. Refer to the TSL Reference Guide for more details.

For a full description of the default obligatory attributes, optional attributes and selectors for each class, see Chapter 6, "Configuring the GUI Map."

XR_WINDOW_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning windows.

(Default = class label, X_name class_index, index)

XR_OBJ_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning objects in the general Mercury object class.

(Default = class X_class X_name, class_index, location)

XR_PBUTTON_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning push buttons.

(Default = class label, X_name class_index, location)

XR_CBUTTON_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning check buttons.

(Default = class label, X_name class_index, location)

XR_RBUTTON_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning radio buttons.

(Default = class label, X_name class_index, location)

XR_MENU_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning menus.

(Default = class label, X_name class_index, location)

XR_SCROLL_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning scroll bars.

(Default = class attached_text orientation, X_attached_name X_name class_index, location)

XR_LIST_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning lists.

(Default = class attached_text, X_name class_index, location)

XR_EDIT_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning edit objects.

(Default = class attached_text, X_name class_index, location)

XR_SPIN_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning spinboxes.

(Default = class attached_text, X_name class_index, location)

XR_STATIC_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning static text objects.

(Default = class X_name, label class_index, location)

XR_NOTEBOOK_REC_ATTR = *attribute list*

Defines the attributes XRunner identifies when learning a notebook object.

(Default = class, X_name class_index, location)

Context Sensitive Customization

The parameters in this category allow you to customize XRunner to meet the specific testing requirements of your application.

MIC_BUTTON_CLICK_LOCATION = {CENTER|CORNER}

Defines the location of the mouse-click performed on button objects during test execution in the analog mode.

Note that this adjustment can be made only for applications running under Motif.

(Default = CENTER)

MIC_EDIT_RECORD_MULTIPLE = {0|1}

Defines the method for recording single-line edit fields. Normally, single-line edit fields are recorded as single-lines. The alternative method is to record them as multi-line edit-fields.

Note that this adjustment can be made only for applications running under Motif.

(Default = 0, single-line)

MIC_EDIT_REPLAY_BY_CHAR = {0|1}

Defines the method for executing edit_ TSL commands. Normally, entire strings are inserted into edit fields as a single event. The alternative method is to insert the string a single character at a time.

Note that this adjustment can be made only for applications running under Motif.

(Default = 0, inserts entire string as a single event)

MIC_ATT_TEXT_DISTANCE = *integer*

Defines the radius of the area XRunner searches between a GUI object and possible static text tag while learning/recording the object.

(Default = 50 [pixels])

MIC_ATT_TEXT_CORNER = {UPPER_LEFT | UPPER_RIGHT | LOWER_LEFT | LOWER_RIGHT | ATT_DISABLE}

Defines the point from which XRunner begins to search for a possible static text tag.

The options are:

UPPER_LEFT	Begins search in upper-left corner.
UPPER_RIGHT	Begins search in upper-right corner.
LOWER_LEFT	Begins search in lower-left corner.
LOWER_RIGHT	Begins search in lower-right corner.
ATT_DISABLE	Disables the search for attached_text.

(Default = MIC_ATT_DISABLE)

MIC_MAX_LIST_ITEM_LENGTH = *integer*

Defines the maximum number of characters XRunner records in list item objects. By default, XRunner records the *name* of the selected item in a list, for instance:

```
(list_select_item(<list>, "my_name");
```

If the selected item's length is longer than the default 128 characters, XRunner records its *position* in the list, instead, for instance:

```
(list_select_item(<list>, "#7");
```

Use this parameter to set the maximum length of a list item for which XRunner should record a name and not a number.

The maximum setting is 256.

(Default = 128 [characters])

MIC_LIST_TAG = {TAG_BY_ATT_TEXT|TAG_BY_X_NAME}

Defines how XRunner suggests logical names when learning list objects.

The options are:

TAG_BY_ATT_TEXT Name suggested according to attached text.

TAG_BY_X_NAME Name suggested according to X_name.

This adjustment can be made only for applications running under Motif.

(Default = TAG_BY_ATT_TEXT)

MIC_COMBO_OPEN = {0|1}

Defines the manner in which XRunner operates on combobox widgets during test execution. XRunner can select a list item from the displayed list by opening the combobox, then closing it. Alternatively, the list item is selected without opening and closing the combobox.

This adjustment can be made only for applications running under Motif.

(Default = 1 , opens and closes comboboxes)

MIC_EDIT_TAG = {TAG_BY_ATT_TEXT|TAG_BY_X_NAME}

Defines how XRunner suggests logical names when learning edit objects.

The options are:

TAG_BY_ATT_TEXT Name suggested according to attached text.

TAG_BY_X_NAME Name suggested according to X_name.

This adjustment can be made only for applications running under Motif.

(Default = TAG_BY_ATT_TEXT)

MIC_SPIN_TAG = {TAG_BY_ATT_TEXT|TAG_BY_X_NAME}

Defines how XRunner suggests logical names when learning spin objects.

The options are:

TAG_BY_ATT_TEXT Name suggested according to attached text.

TAG_BY_X_NAME Name suggested according to X_name.

This adjustment can be made only for applications running under Motif.

(Default = TAG_BY_ATT_TEXT)

MIC_SPIN_RECORD = {On|Off}

Defines the method by which XRunner records operations on spin objects. When recording the resultant value in the spin-box only, without performing mouse-clicks, a **spin_set** command is generated. When recording the mouse-clicks that scroll the spin box, a **spin_next** or **spin_prev** command is generated.

This adjustment can be made only for applications running under Motif.

Default = On (records **spin_set**)

MIC_SPIN_MAX_EVENTS = *integer*

Defines the maximum number of mouse-clicks to be performed on spin objects. If the operation requires a number of clicks greater than that set, XRunner automatically sets the spin object to the specific value directly.

This adjustment can be made only for applications running under Motif or using CDE Dt Widgets.

(Default = 200 [mouse-clicks])

MIC_TAG_CREATE = {TAG_AS_IS | TAG_LOCATION | TAG_CLASS_INDEX | TAG_BY_X_NAME | TAG_BY_ATT_TEXT}

Defines the way XRunner creates a logical name from the physical description in the GUI map.

The options are:

TAG_AS_IS	Configures the Context Sensitive libraries not to perform a name-check on existing objects; they do not create a unique tag for the object, either.
TAG_CLASS_INDEX	Configures the Context Sensitive libraries to add a suffix of type <class_index> to the logical name if the name is not unique.
TAG_LOCATION	Configures the Context Sensitive libraries to add a suffix of type <tag_location> to the logical name if the name is not unique.

(Default = TAG_LOCATION)

MIC_EDIT_ACTIVATE_LOW_LEVEL = {On|Off}

Executes the **edit_activate** command on low level.

In OLIT, most test execution actions are performed on low level, that is, using the server extensions.

In Motif, most test execution actions are performed directly through messages to the widgets. If you are using Motif, you may specify that certain actions be performed on low level instead of by messages.

(Default = Off)

MIC_LIST_ACTIVATE_LOW_LEVEL = {On|Off}

Executes the **list_activate_item** and **list_select_item** commands on low level.

In OLIT, most test execution actions are performed on low level, that is, using the server extensions.

In Motif, most test execution actions are performed directly through messages to the widgets. If you are using Motif, you may specify that certain actions be performed on low level instead of by messages.

(Default = Off)

MIC_MENU_SELECT_LOW_LEVEL = {On|Off}

Executes the **menu_select_item** command on low level.

In OLIT, most test execution actions are performed on low level, that is, using the server extensions.

In Motif, most test execution actions are performed directly through messages to the widgets. If you are using Motif, you may specify that certain actions be performed on low level instead of by messages.

(Default = Off)

MIC_BUTTON_SET_LOW_LEVEL = {On|Off}

Executes the **button_set** command on low level.

In OLIT, most test execution actions are performed on low level, that is, using the server extensions.

In Motif, most test execution actions are performed directly through messages to the widgets. If you are using Motif, you may specify that certain actions be performed on low level instead of by messages.

(Default = Off)

MIC_BUTTON_PRESS_LOW_LEVEL = {On|Off}

Executes the **button_press** command on low level.

In OLIT, most test execution actions are performed on low level, that is, using the server extensions.

In Motif, most test execution actions are performed directly through messages to the widgets. If you are using Motif, you may specify that certain actions be performed on low level instead of by messages.

(Default = Off)

MIC_XPATH_ONLY = {On|Off}

Forces XRunner to learn the X_name attribute only when creating physical descriptions for GUI objects. This parameter should be defined as On only if your application uses unique X_names.

Note, this adjustment can be made only for applications running under Motif.

(Default = Off, uses regular attributes)

MIC_NO_IN_PARENT = {On|Off}

Enables/disables an XRunner check that scans the entire hierarchy of a given widget to verify if the widget is clipped out of its parent area.

Note that for applications running under Olit, this parameter must be set to On at all times.

(Default = On, does not perform the check)

MIC_ICON_RECORD = {REC_ICON_PATH|REC_ICON_LABEL| REC_ICON_INDEX}

Defines the method in which XRunner records operations performed on icons inside container objects.

The options are:

REC_ICON_PATH	The icon is identified by all the labels in the path of the icon hierarchy.
REC_ICON_LABEL	The icon is identified by its label only.
REC_ICON_INDEX	The icon is identified by its serial number in relation to the container, as assigned by XRunner.

Note that these adjustments can be made only for applications running under Motif.

(Default = REC_ICON_PATH)

MIC_RECOVERY= {On|Off}

Activates/disables the recovery facility that prevents the application under test from crashing in the event of certain Context Sensitive operations.

(Default = On, recovery enabled)

MIC_CACHE_EXCP = {0|1}

Enables/disables caching for objects.

(Default = 0)

[Note that the value of this parameter cannot be set using the Configuration form. Use a **setvar** statement in a test script]

MIC_EXACT_RGB = 1

Enables/disables RGB values to color strings when creating the physical description in the GUI Map.

Note, this adjustment can be made only for applications running under Motif.

(Default is undefined, conversion enabled)

[Note that the value of this parameter cannot be set using the Configuration form. Use a **setvar** statement in a test script.]

MIC_RGB_DATABASE = *pathname*

Defines the location of a user-defined RGB database file.

Note, this adjustment can be made only for applications running under Motif.

(Default is the RGB database file in the /dat subdirectory of your home directory.)

[Note that the value of this parameter cannot be set using the Configuration form. Use a **setvar** statement in a test script.]

Learn Window Parameter

XR_LEARN_TIMEOUT = *integer*

Defines the timeout during which XRunner learns windows.

(Default = 180[seconds]

Recording Level Parameter

XR_REC_LEVEL = *integer*

Sets the default recording mode. 0 sets Analog and 1 sets Context Sensitive.

(Default = 1)

Text Checkpoint Parameters

XR_FONT_GROUP = *font group name*

Sets the default active font group. The designated font group must be defined in a font library designated by the XR_GLOB_FONT_LIB parameter. The value assigned to this parameter can be overridden from within a test script, using a **setvar** function, for example:

```
setvar ("XR_FONT_GROUP", "my_fonts");
```

(Default = stand)

XR_GLOB_FONT_LIB = *library pathname*

Identifies the directory in which the Global Font Library resides.

(Default = $M_ROOT/fonts)

XR_TEXT_PREVIEW_FONT = *font name*

Specifies the name of the font to be used when previewing text captured by the GET TEXT or WAIT BITMAP AREA command.

(Default = 9x15)

XR_TEXT_RECOGNITION_TIMEOUT = *integer*

Sets the maximum interval that XRunner waits when searching for text.

Default = 500 [milliseconds]

XR_TEXT_REMARKS = {On|Off}

Determines whether the text that is captured in a text checkpoint is inserted into the test script as a remark during test creation.

(Default = On)

[Note that the value of this parameter cannot be set using the Configuration form.]

XR_TEXT_REMOVE_BLANKS = {On|Off}

Determines whether leading and trailing blanks found in the recognized text are removed.

(Default = On)

XR_TEXT_SEARCH_RADIUS = *integer*

Sets the radius (in pixels) of the search for text around a point, when no parameters are specified in the **get_text** function.

(Default = 15 [pixels])

XR_UCFG_TEXT = {On|Off}

Enables or disables text checkpoint features.

(Default = On)

Application Mouse and Keyboard Parameters

MIC_CLICK_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to perform mouse clicks.

(Default = Left)

MIC_DBL_CLICK_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to perform double clicks.

(Default = Left)

MIC_DRAG_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to perform drags.

(Default = Left)

MIC_WIN_ACTIVATE_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to activate a window.

(Default = Left)

MIC_EDIT_ACTIVATE_KEY = *key*

Defines which key is pressed at the end of writing to an edit field to activate it. Any of the following keys can be defined: Return, Ctrl, Home, Esc.

(Default = Return)

MIC_EDIT_ACTIVATE_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to activate an edit item.

(Default = Left)

MIC_EDIT_CLICK_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to move the focus to an edit field before writing into it.

(Default = Left)

MIC_SCROLL_CLICK_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to scroll.

(Default = Left)

MIC_LIST_ACTIVATE_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to activate a list item.

(Default = Left)

MIC_LIST_ACTIVATE_KEY = *key*

Defines which key is used to activates a list item. Any of the following keys can be defined: Return, Ctrl, Home, Esc.

(Default = Return)

MIC_LIST_OPEN_BUTTON = {Left|Middle|Right}

Defines which mouse-button opens a list box.

(Default = Left)

MIC_LIST_CLOSE_BUTTON = {Left|Middle|Right}

Defines which mouse-button closes a list box.

(Default = Left)

MIC_ADD_OR_DESELECT_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to add or deselect items in a list box.

(Default = Left)

MIC_LIST_SELECT_BUTTON = {Left|Middle|Right}

Defines which mouse button is used to select items in a listbox.

(Default = Left)

MIC_SPIN_CLICK_BUTTON = {Left|Middle|Right}
Defines which mouse button is used to select a spin object.

(Default = Left)

MIC_SPIN_MIN_KEY = *key*
Defines which key is used to go to the beginning of a spin object.

(Default = Home)

MIC_SPIN_MAX_KEY = *key*
Defines which key is used to go to the end of a spin object.

(Default = End)

MIC_BUTTON_PRESS_BUTTON = {Left|Middle|Right}
Defines the mouse-button used to perform **button_press** commands during test execution.

(Default = Left)

MIC_MENU_OPEN_BUTTON = {Left|Middle|Right}
Defines which mouse button is used to open a menu.

(Default = Left)

MIC_MENU_CLOSE_BUTTON = {Left|Middle|Right}
Defines which mouse button is used to close a menu.

(Default = Left)

MIC_POPUP_MENU_POPUP_BUTTON = {Left|Middle|Right}
Defines which mouse button is used to open a popup menu.

(Default = Right)

MIC_POPUP_MENU_POPUP_SELECT = {Left|Middle|Right}
Defines which mouse button is used to select an item from a popup menu.

(Default = Left)

XRunner User Interface

XR_HIDE_BUBBLE_HELP = {On|Off}

Displays/hides icon bubble help.

(Default = Off, bubbles visible)

XR_INSERT_NEWLINES = {On|Off}

Specifies if XRunner inserts a blank line before and after GUI checkpoint and Synchronization point statements.

(Default = On)

XR_EDITOR_MAX_CHARS = *integer*

Determines the maximum number of characters that can be written per line in the XRunner window. When a generated script statement extends beyond this maximum length during a record session, the script line is split between two or more lines.

(Default = 80)

XR_REPORT_PASS_COLOR = *string*

Defines the color used in the test report to indicate a successful test run. The color is used in the "check" on the upper right corner of the test report summary and in lines detailing specific successful events in the test report log.

(Default = green4)

XR_REPORT_FAIL_COLOR = *string*

Defines the color used in the test report to indicate a failed test run. The color is used in the "cross" on the upper right corner of the test report summary and in lines detailing specific failed events in the test report log.

(Default = red)

Execution Environment Parameters

The following parameters appear in the Configuration form, but are read-only.

Current Directory = *pathname*

Indicates the current working directory for the test. There is no default value for this parameter.

Expected Results Directory = *pathname*

Defines the full pathname of the expected results directory associated with the current execution of the test. There is no default value for this parameter.

Line Number = *integer*

Displays the current line of the execution marker in the test script. There is no default value for this parameter.

Results Directory = *integer*

Sets the full pathname of the results directory during a verification run. There is no default value for this parameter.

System Mode = {verify|update|debug}

Defines the current mode: either Verify, Debug, or Update.

Test Name = *pathname*

Defines the full pathname of the current test. There is no default value for this parameter.

Softkey Parameters

The following parameters define XRunner's softkeys. Note that any modifications you make to softkey settings from the Configuration form are invalid for the current session. Changes take effect when you restart XRunner.

XR_SOFT_ANIMATE = *softkey*

Defines the RUN FROM ARROW softkey. Pressing this key is equivalent to choosing the Run from Arrow command from the Run menu.

(Default = F8)

XR_SOFT_MARKLOCATOR = *softkey*

Defines the MARK LOCATOR softkey used to record the absolute coordinate position (in pixels) of the screen pointer.

(Default = Ctrl Left F5)

XR_SOFT_PAUSE = *softkey*

Defines the PAUSE softkey. Pressing this softkey is equivalent to choosing the Pause command from the Run menu.

(Default = F8)

XR_SOFT_RECORD = *softkey*

Defines the RECORD softkey. Pressing this softkey is equivalent to choosing the Record-Context Sensitive command from the Run menu.

(Default = F5)

XR_SOFT_STEP = *softkey*

Defines the STEP softkey. Pressing this softkey is equivalent to choosing the Step command from the Run menu.

(Default = F7)

XR_SOFT_STEP_INTO = *softkey*

Defines the STEP INTO softkey. Pressing this softkey is equivalent to choosing the Step Into command from the Run menu.

(Default = Ctrl Left F7)

XR_SOFT_STOP = *softkey*

Defines the STOP softkey. Pressing this softkey is equivalent to choosing the Stop command from the Run menu (when recording) or the Abort command from the Run menu (when running a test).

(Default = F6)

XR_SOFT_CHECK_WINDOW = *softkey*

Defines the CHECK WINDOW softkey. Pressing this softkey is equivalent to choosing the Check Bitmap > Window command from the Create menu. (Default = F2)

XR_SOFT_CHECK_PARTIAL_WINDOW = *softkey*

Defines the CHECK WINDOW (AREA) softkey. Pressing this softkey is equivalent to choosing the Check Bitmap > Area command from the Create menu.

(Default = Ctrl Left F2)

XR_SOFT_GET_TEXT = *softkey*

Defines the GET TEXT softkey.

(Default = Ctrl Left F1)

XR_SOFT_WAIT_WINDOW = *softkey*

Defines the WAIT WINDOW softkey. Pressing this softkey is equivalent to choosing the Wait Bitmap > Window command from the Create menu.

(Default = F4)

XR_SOFT_WAIT_PARTIAL_WINDOW = *softkey*

Defines the WAIT WINDOW (AREA) softkey. Pressing this softkey is equivalent to choosing the Wait Bitmap > Area command from the Create menu.

(Default = Ctrl Left F4)

XR_SOFT_WAIT_REDRAW = *softkey*

Defines the WAIT REDRAW softkey. Pressing this softkey instructs XRunner to wait for a specific window to be redrawn, without evaluating its contents.

(Default = F3)

XR_SOFT_WAIT_REDRAW_PARTIAL_WINDOW = *softkey*

Defines the WAIT REDRAW AREA softkey. Pressing this softkey instructs XRunner to wait for a specific window area to be redrawn, without evaluating its contents.

(Default = Ctrl Left F3)

XR_SOFT_CHECK_GUI = *softkey*

Defines the CHECK GUI softkey. Pressing this softkey is equivalent to choosing the Check GUI > Checklist command from the Create menu.

(Default = Alt Left F2)

XR_SOFT_OBJ_FUNC_CALL = *softkey*

Defines the INSERT FUNCTION softkey. Pressing this softkey is equivalent to choosing the Insert Function > Object/Window command from the Create menu.

(Default = Ctrl Left F9)

XR_SOFT_GEN_FUNC_CALL = *softkey*

Defines the INSERT FUNCTION FROM LIST softkey. Pressing this softkey is equivalent to choosing the Insert Function > From List command from the Create menu.

(Default = F9)

Sun NeWS Server Parameter

XR_NEWS_COMPAT = {On|Off}

Instructs XRunner to use the lower left corner of the screen as the coordinate system origin, as in previous versions (version 1.6 and earlier of XRunner). This parameter should be set to On only if you are using Sun's NeWS server and have tests that were created on previous versions of XRunner.

(Default = Off)

[Note that the value of this parameter cannot be set using the Configuration form.]

Exceptions

The configuration parameters for exceptions do not appear in the Configuration form.

XR_EXCP_POPUP = *exception list*

Defines popup exceptions. Exceptions in the list are separated by blank spaces.

There is no default value for this parameter.

XR_EXCP_OBJECT= *exception list*

Defines object exceptions. Exceptions in the list are separated by blank spaces.

There is no default value for this parameter.

XR_EXCP_TSL = *exception list*

Defines TSL exceptions. Exceptions in the list are separated by blank spaces.

There is no default value for this parameter.

Input Device Parameters

XR_INP_CAPS_POLICY = {0|1}

Sets the event that XRunner records for the CapsLock key. This parameter can be set to one of two values:

0	Normal mode. Pressing the CapsLock key generates both press and release events.
1	Shift-release mode (common on European keyboards). Pressing the CapsLock key generates only a press event. Pressing the Shift key generates the release event for the CapsLock key. (Until the Shift key is pressed, additional presses on the CapsLock key have no effect.)

(Default = 0)

XR_INP_KBD_NAME = *file pathname*

Designates the path and name of the keyboard definition file. This file specifies the language that appears in the test script when you type on the keyboard during recording. For XRunner 4.0 and higher, all necessary keyboard information is acquired by default from the XServer, instead of from a keyboard file. For XRunner 4.0 or higher, set this parameter to NONE.

(Default = NONE)

XR_KBD_ALIAS_FILE = *file pathname*

Defines the file containing global key aliases.

(Default file located in $M_ROOT/dat)

XR_INP_KBD_DEV_ID = *keyboard device id*

Specifies the keyboard device id used by XTestExtension1. Most machines work with the default value. For some IBM X Terminal and DEC Alpha machines this parameter must be set to 2. You may ignore this parameter if you are running the Sun NeWS server.

(Default = 0)

XR_INP_MKEYS = *mouse_button_code string*

Assigns a unique name (*string*) to each of the mouse buttons. When a test is recorded in Analog mode, this name is the expression enclosed in the TSL **type** statement generated whenever the specified button is activated. For example, the default name assigned to each of the three mouse buttons (when pressed alone or in conjunction with the Shift key) are as follows:

```
XR_INP_MKEYS=0x01 Right    S_Right \
              0x02  Middle   S_Middle \
              0x04  Left     S_Left\
              0x08  Aux1     S_Aux1
```

Note that button codes are specified here in hexadecimal notation. When defining your mouse keys, be sure to use hexadecimal notation.

XR_INP_MOUSE_DEV_ID = *mouse device id*

Specifies the mouse device id used by XTestExtension1.

(Default = 1)

Configuration File Contents

System configuration files are text files that can include the following types of data:

- assignment statements
- directives
- blank lines

- comments

Assignment Statements

The main purpose of the configuration file is to allow the assignment of values to XRunner configuration parameters. The values assigned to these parameters determine how the program will run. Most of these values will be defined by the top-level (*~/.xrunner*) configuration file following system installation.

Other parameters point to system locations. For example, the parameter XR_GLOB_FILTER_LIB identifies the directory in which the Global Filter Library is stored.

In addition to assignment statements used to set values for parameters, you can assign values to user-defined variables. A typical use would be to assign an arbitrary shorthand name to a path which may appear any number of times in the configuration file. For example, the line

```
P153T = /project_15/ver_3/tests
```

assigns the specified pathname to the variable P153T. Whenever the name of this variable subsequently appears in the configuration file, the associated path will be understood. Thus if the location of the Global Filters Library is specified by the line

```
XR_GLOB_FILTER_LIB = $(P153T)/.
```

XRunner will understand that this logical folder resides under the pathname /project_15/ver_3/tests.

Note the following points:

- The equal sign (=) is always used to assign a value to a variable, whether this variable is a system parameter or a user-defined variable.
- Whenever a user-defined variable is initiated in the file, the name of this variable must be enclosed within parentheses; the enclosed variable name is preceded by a dollar sign character ($). Note that environmental variables may also be accessed using the same convention.

➤ If the same item appears more than once in the configuration file, the *last* value assigned to this item will be used by the system.

Regarding case sensitivity of names, note the following points:

➤ Names of variables and system parameters appearing in the configuration file are *not* case sensitive.

➤ Boolean values assigned to variables or parameters are *not* case sensitive.

➤ Values assigned to certain system parameters *may* be case sensitive, depending on the nature of the parameter.

Directives

The configuration file can contain one or more include-directives. An include-directive is used to integrate the entire contents of the specified file in the configuration data processed by XRunner. An include-directive is composed of:

➤ the character @ in the first column of the file line

➤ the label include

➤ the @ character, followed by the name of the file (enclosed between quotation marks) to be integrated at this point

For example:

```
@include "@loc_file"
.
.

@include "@/file2"
```

Note that each file to be integrated in the configuration data must be specified by its own include entry.

Include-directives can be nested: A file that is referenced by an include entry in the configuration file may in turn contain its own include directives to other files. Such nesting is supported up to ten levels. When a *relative* name is used to specify the file to be integrated, the specified name must express

the location of this file relative to the file in which the calling include-directive appears.

Blank Lines and Comments

As it processes, a configuration file ignores blank lines and comments. Comments may be inserted in a file using the number sign (#). All text that appears between a number sign and the end of a line is understood to be a comment.

Line Format

When a record in the configuration file extends beyond a single line, the backslash character (\) indicates that the record continues on the next line.

```
XR_INP_MKEYS=0x01 Right    S_Right \
              0x02  Middle   S_Middle \
              0x04  Left     S_Left\
              0x08  Aux1     S_Aux1
```

In the above example, the XR_INP_MKEYS parameter is used to assign a unique name (string) to each of the mouse buttons when pressed alone as well as in conjunction with the Shift key.

When more than one value is assigned to the same parameter or user-defined variable, the delimiter between the values is a blank space.

Special Characters

The backslash character (\) can also be used within a configuration file as an escape character. If a record must include a special character which has a different, reserved function, precede the special character with a backslash. The character that follows will then be read literally by the XRunner interpreter. (For example, in order that a backslash be understood as a literal backslash, type in a double backslash [\\].)

The backslash character is also used to indicate literal carriage return (ENTER) and tab characters in the configuration file.

Quotation marks (") can be used to indicate that two or more string segments constitute a single value.

34

Initializing Special Configurations

By creating *startup tests*, you can automatically initialize special testing configurations each time you invoke XRunner.

This chapter describes:

- Creating Startup Tests
- Sample Startup Test

About Initializing Special Configurations

A startup test is a test script that is automatically executed each time you start XRunner. You can create startup tests that load GUI map files and compiled modules, configure recording, and invoke the application under test.

You can create startup tests on two levels:

- On the first level, you define the XR_TSL_INIT environment variable so it points to the full path of the startup test. This test can be used to customize XRunner for a group of users testing the same application. For example

  ```
  setenv XR_TSL_INIT /qa/share/qinit
  ```

- The second level startup test is the *tslinit* test stored in your home directory. This test can be used to tailor XRunner to your individual needs.

Creating Startup Tests

It is recommended that you add the following types of statements to your startup test:

- **load** statements, which load compiled modules containing user-defined functions that you frequently call from your test scripts.
- **GUI_load** statements, which load one or more GUI map files. This ensures that XRunner recognizes the GUI objects in your application when you execute tests.
- **a system statement that invokes the application under test.**
- statements that enable XRunner to generate custom record TSL functions when you operate on custom objects, such as **add_cust_record_class**.

By including the above elements in a startup test, XRunner automatically compiles all designated functions, loads all necessary GUI map files, configures the recording of GUI objects, and loads the application under test.

Note that you can use the Test Wizard to create a basic startup test called *mytest* which loads a GUI map file and the application under test.

You can create a startup test for a group of users, or you can create startup tests for each individual user.

Sample Startup Test

The following is an example of the types of statements that might appear in a startup test:

```
# Load the compiled module "qa_funcs"
load ("qa_funcs", 1, 1);

# Load the GUI map file "bank.gui"
GUI_load ("/qa/gui/bank.gui");
```

```
# Invoke the sample Flights application
system ("cd" & getenv("M_ROOT") & "/samples/bin/frs/";
            set M_ROOT= "& getenv("M_ROOT") & "; export M_ROOT;
./airspace1b &");
```

Part VIII

Working with LoadRunner

35

Testing Client/Server Systems

Today's applications are run by multiple users over complex, client/server systems. With LoadRunner, Mercury Interactive's client/server testing tool, you can simulate the load of real users interacting with your server and measuring system performance. Note that LoadRunner is an independent testing tool and must be purchased separately.

This chapter describes:

- Simulating Multiple Users
- Virtual User Technology
- GUI Vusers
- Developing and Running Scenarios
- Creating Scripts for XRunner GUI Vusers
- Measuring Server Performance
- Synchronizing Virtual User Transactions
- Creating a Rendezvous
- A Sample Vuser Script

About Testing Client/Server Systems

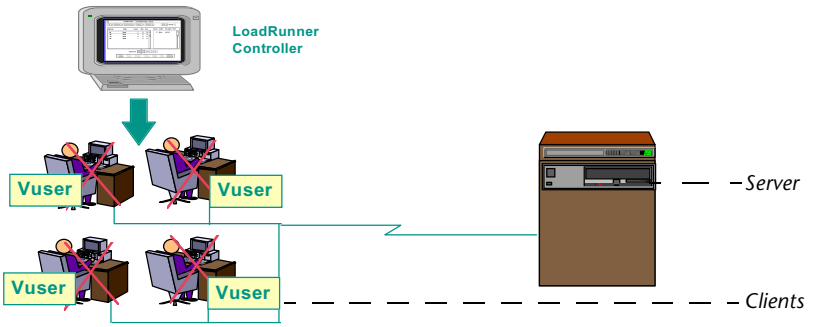
Software testing is no longer confined to testing applications that run on a single, stand-alone workstation. Applications are run in a network environment where multiple client PCs or UNIX workstations interact with a central server.

Modern client/server architectures are complex. While they provide an unprecedented degree of power and flexibility, these systems are difficult to test. LoadRunner simulates server load and then accurately measures and analyzes server performance and functionality. This chapter provides an overview of how to use LoadRunner and XRunner to test your server. For detailed information about how to test a client/server system refer to your LoadRunner documentation.

Simulating Multiple Users

With LoadRunner, you simulate the interaction of multiple users (clients) with the server by creating *scenarios*. A scenario defines the events that will occur during each client/server testing session, such as the number of users, the actions they perform, and the machines they use. For more information about scenarios, refer to the *LoadRunner Controller User's Guide*.

In the scenario, LoadRunner replaces the human user with a *virtual user* (Vuser). A Vuser simulates the actions of human users and submits input to the server. A scenario can contain tens, hundreds, or thousands of Vusers.



LoadRunner replaces human users with virtual users

Virtual User Technology

There are three types of Vusers, each designed to handle different aspects of today's client/server architectures:

- DB Vusers
- RTE Vusers
- GUI Vusers

DB Vusers submit input to the server without relying on client software. Instead, they directly access the server through API calls. This lets you simulate large numbers of users accessing the server simultaneously.

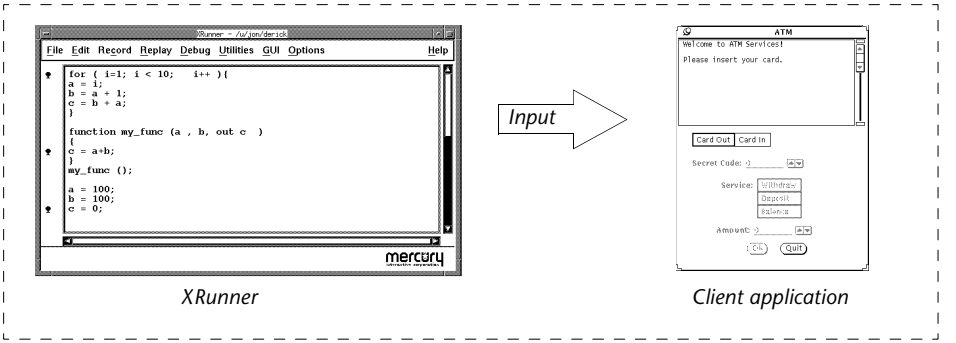
You describe the actions DB Vusers will perform by writing C programs. These include functions that control test execution, specify the input to submit to the server, and measure server performance. For additional information, refer to the *LoadRunner DB Vuser User's Guide*.

RTE Vusers operate character-based client applications in Mercury Interactive's remote terminal emulator. With RTE Vusers, you create C programs that type input and wait for output from a remote terminal emulator. For additional information, refer to the *LoadRunner RTE Vuser User's Guide*.

GUI Vusers operate graphical user interface applications in environments such as Motif. Each GUI Vuser simulates a real user submitting input to and receiving output from a client application.

A GUI Vuser replaces a human user that operates a client application. The client application can be any application used to access the server, such as a database client. Each GUI Vuser executes a Vuser script, a test that describes

the actions it performs during the scenario. It includes statements that measure and record the performance of the server.



A GUI Vuser

GUI Vusers

LoadRunner works with two types of GUI Vusers:

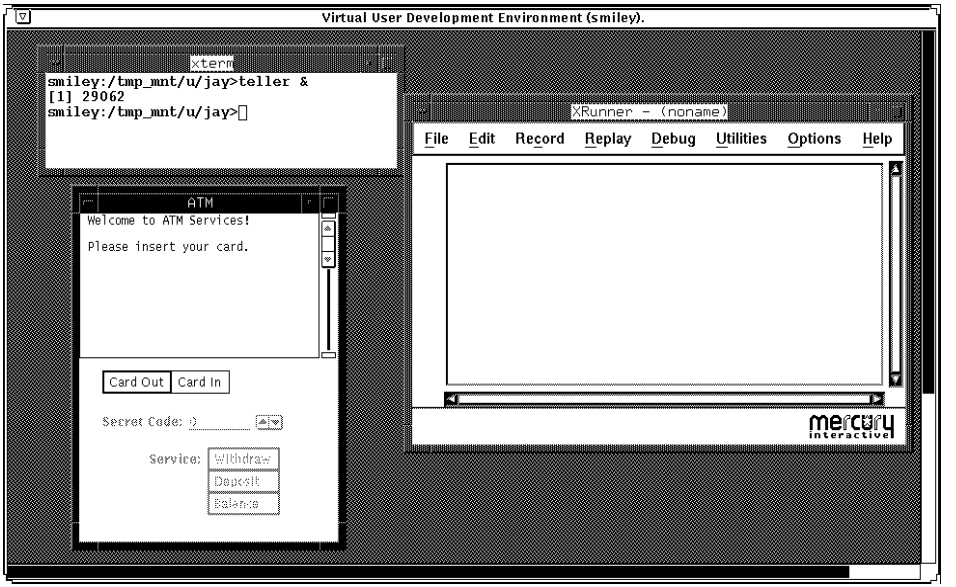
- XRunner
- VXRunner

An XRunner GUI Vuser uses an existing XRunner installation as a Vuser. LoadRunner invokes XRunner on a remote host and runs existing XRunner scripts through its controller. This type of GUI Vuser has full XRunner capabilities, but is limited to a single copy of XRunner per host; it can only simulate a single X Windows user per host.

A VXRunner GUI Vuser uses an enhanced version of XRunner, VXRunner, to incorporate additional LoadRunner capabilities. VXRunner allows you to develop and simulate multiple X Windows users on a single host. VXRunner does not support Context Sensitive testing or verification, but it is ideal for simulating load testing with X Window client applications. For more information, refer to the *LoadRunner GUI Vuser User's Guide*

LoadRunner is equipped with a Virtual User Development Environment (VUDE). The VUDE consist of VXRunner, an xterm window (from which you invoke an application), and a client application. You develop

VXRunner scripts through LoadRunner inside the VUDE, the Virtual User Development Environment.



You create your Vuser script with VXRunner, the same way you create tests in XRunner, with recording and programming. Vuser scripts created by VXRunner can be assigned to multiple GUI Vusers on a single host. This is ideal for load testing and environments where a limited number of host machines are available. You create Vusers and assign tests through a scenario script. For more details about the VUDE and scenario scripts, refer to the *LoadRunner GUI Vuser User's Guide*.

Developing and Running Scenarios

You develop a scenario script in which you create Vusers, assign tests to Vusers, and designate which workstations will host Vusers during the scenario. With the controller, you run the scenario. After you run a scenario, you analyze server performance with the controller's performance analysis graphs and reports.

The following procedure summarizes the process of testing your server with LoadRunner and XRunner GUI Vusers. For more detailed information, refer to the *LoadRunner GUI Vuser User's Guide*.

1 Create Vuser scripts.

A Vuser script describes the actions a Vuser will perform during the scenario. You create GUI test scripts with XRunner. A GUI Vuser script contains TSL statements specially designed for client/server testing. For example, a GUI Vuser script contains transaction statements to measure server performance and rendezvous statements to synchronize the actions of Vusers on multiple hosts.

2 Create the scenario with a scenario script.

A scenario consists of many Vusers. To create Vusers, assign Vuser scripts, and designate host machines, you write a script in the Scenario Script window using LoadRunner statements. You must replay the scenario script to set up the scenario.

3 Run the scenario.

When you click on Run from the controller, LoadRunner executes the script and records scenario performance data. You use this information to generate analysis of server performance. The scenario will end when all the Vusers complete executing their designated Vuser scripts.

4 Analyze server performance.

After the scenario run, you can use LoadRunner's graphs and reports to analyze server performance.

Creating Scripts for XRunner GUI Vusers

To create a script for an XRunner GUI Vuser, you use XRunner as you normally would through recording and programming. In addition, you add several LoadRunner functions that allow you to measure system performance and synchronize multiple Vusers.

To create a script for a VXRunner GUI Vuser, you must invoke VXRunner through LoadRunner. For a detailed explanation of VXRunner GUI Vusers, refer to the *LoadRunner GUI Vuser User's Guide*.

To create a script for an XRunner GUI Vuser:

1. Open XRunner.
2. Invoke the client application.

   A client application is any application used to access and interact with the server, such as a database client.
3. Record operations on the client application.

   Use XRunner to record the keyboard and mouse strokes you want the Vuser to perform on the client application. The actions are automatically recorded and transcribed into a Vuser script.
4. Edit your Vuser script with XRunner and program additional TSL statements. Enhance the test script using loops and other control-flow structures.
5. Define actions within the Vuser script as transactions to measure server performance.

   A Transaction measures the server's response time to requests submitted by Vusers. For instance, you can define a transaction that measures how long it takes the server to respond to an SQL query sent by a Vuser. After you define the transaction, you use it to measure server performance under different loads. For example, you can measure how the server performs when one user or one hundred users perform the transaction.
6. Add synchronization points to the Vuser script. These tell the Vuser to wait for a specific event to occur, and then resume script execution.

   Synchronization points ensure that transactions accurately measure the time it takes the server to respond to requests sent by users. For example, you could insert a synchronization point that waits for confirmation that a request has been processed. The user knows that the server has processed the request when the word "Done" appears in the client application.
7. Add *rendezvous* points to the Vuser script to coordinate the actions of multiple Vusers.

   During a scenario run, each time a Vuser encounters a rendezvous point, it pauses and waits for other Vusers to arrive at a designated meeting place (the rendezvous). LoadRunner ensures the Vusers cannot leave the rendezvous until the last Vuser arrives, or it receives a command to release them. When

all the Vusers have arrived at the rendezvous, they are released together and perform the next task in their Vuser scripts.

8 Save your script and close XRunner.

Measuring Server Performance

Transactions measure how your server performs under the load of many users. A transaction may be a simple task, such as entering text into a text field, or it may be an entire test which includes multiple tasks. LoadRunner measures the performance of a transaction under different loads. You can measure the time it takes to perform the same transaction by a single user or by a hundred users.

The first stage of creating a transaction is to declare its name at the start of the Vuser script. When you assign the Vuser script to a Vuser, in the LoadRunner controller, LoadRunner scans the Vuser test for transaction declaration statements. If the script contains a rendezvous declaration, LoadRunner reads the name of the transaction and displays it in the Transactions window.

To declare a transaction, you use the **declare_transaction** function. The syntax of this function is:

declare_transaction ("*transaction_name*" **);**

The *transaction_name* can be any string expression that names the transaction. The string can contain letters, numbers, and underscore characters (_). Note that the first character of the name cannot be a number.

Next, you mark the point where LoadRunner will start to measure the transaction. To mark the start of a transaction you insert a **start_transaction** statement into your Vuser script. This must be placed immediately before the action you want to measure. The syntax of this function is:

start_transaction (*transaction_name* **);**

The *transaction_name* is the name you assigned in the **declare_transaction** statement.

To indicate the end of the transaction, you insert an **end_transaction** statement into your Vuser script. If the transaction to be analyzed is the entire test, then the **end_transaction** statement will be the last line in the Vuser test script. The syntax of this function is:

end_transaction (*transaction_name* , **[** *status* **])**;

The *transaction_name* parameter is the name you assigned in the **declare_transaction** statement. The *status* parameter can be set to PASS or FAIL. This tells LoadRunner if the transaction passed or failed. The default is PASS.

Synchronizing Virtual User Transactions

For transactions to accurately measure server performance, they must reflect the time the server takes to respond to user requests. A human user knows that the server has completed processing a task when a visual cue, such as a message, appears. For instance, suppose you want to measure the time it takes for a database server to respond to user queries. You know that the server completed processing a database query when the answer to the query is displayed on the screen. In Vuser tests, you instruct the Vusers to wait for a cue by inserting synchronization points.

Synchronization points tell the Vuser to wait for a specific event to occur, such as the appearance of a message in an object, and then resumes script execution. If the object does not appear, the Vuser continues to wait until the object appears or the timeout expires. You can synchronize transactions by using any of XRunner's synchronization or object functions. For more information about synchronizing GUI Vuser tests, refer to the *LoadRunner GUI Vuser User's Guide*. For more information about XRunner's synchronization functions, see Chapter 13, "Synchronizing Test Execution: Context Sensitive Testing."

Creating a Rendezvous

When designing a scenario, you need to coordinate the test runs of two or more Vusers. You coordinate multiple Vusers by creating a *rendezvous*.

A rendezvous is a meeting place for Vusers. To designate the meeting place, you insert rendezvous statements into your Vuser scripts. When the rendezvous statement is interpreted, the Vuser is held by the controller until all the members of the rendezvous arrive. When all the Vusers have arrived, they are released together and perform the next task in their Vuser scripts.

The first stage of creating a rendezvous is to declare the name of the rendezvous at the start of the Vuser script. When you assign the Vuser script to a Vuser, LoadRunner scans the script for rendezvous declaration statements. If the script contains a rendezvous declaration, LoadRunner reads the name of the rendezvous and creates a rendezvous. If you create another Vuser that runs the same script, the controller will add the Vuser to the rendezvous.

To declare a rendezvous, you use the **declare_rendezvous** function. The syntax of this functions is:

```
declare_rendezvous ( "rendezvous_name" );
```

where *rendezvous_name* is the name of the name of the rendezvous. The *rendezvous_name* must be a string constant and not a variable or an expression.

Next, you indicate the point in the Vuser test where the rendezvous will occur by inserting a **rendezvous** statement. The rendezvous statement tells LoadRunner to hold the Vuser at the rendezvous until all the other Vusers arrive. The function has the following syntax:

```
rendezvous ( rendezvous_name );
```

where *rendezvous_name* is the name of the rendezvous.

A Sample Vuser Script

In the following sample Vuser test, the "Ready" transaction measures how long it takes for the server to respond to a login request from a user. The user clicks the Login button. The user knows that the request has been processed when the word "Done" appears in the client application's Status field.

```
# Declare the transaction and rendezvous names
declare_transaction ("Ready");
declare_rendezvous ("wait");

# Mouse pointer moved to deposit button
move_locator_abs (10, 10, 0);

# Start measuring deposit operation
start_transaction ("Ready");

# Set the rendezvous point
rendezvous ("wait");

# Click left mouse button on Login button
click ("Left");

# Wait for "Done" to appear in the window.
rc = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);

# End Ready transaction
if (rc == 0)
    end_transaction ("Ready", PASS);
else
    end_transaction ("Ready", FAIL);
```

In the first part of the Vuser script, the **declare_transaction** and **declare_rendezvous** functions declare the names of the transaction and rendezvous points in the Vuser script. In this script, the transaction "Ready" and the rendezvous "wait" are declared. The declaration statements enable the LoadRunner controller to display transaction and rendezvous information.

```
# Declare the transaction and rendezvous names
declare_transaction ("Ready");
declare_rendezvous ("wait");
```

Next, the mouse is moved to the location of the Login button on the screen.

```
# Mouse pointer moved to deposit button
move_locator_abs (10, 10, 0);
```

A **start_transaction** statement is inserted just before the Vuser presses the Login button. This instructs LoadRunner to start recording the "Ready"

transaction. The "Ready" transaction measures the time it takes for the server to process the request sent by the Vuser.

```
# Start measuring deposit operation
start_transaction ("Ready");
```

A **rendezvous** statement ensures that all Vusers will press the Login button at the same time, thus creating heavy load on the server.

```
# Set the rendezvous point
rendezvous ("wait");

# Click left mouse button on Login button
click ("Left");
```

Before LoadRunner can measure the time taken to perform the transaction, it must wait for a queue to say that the server has finished processing the request. LoadRunner uses the **wait_text** function to check for a specific text. The user knows that the request has been processed when the "Done" message appears within a certain location on the screen. The *timeout* is set to five seconds.

```
# Wait for "Done" to appear in the window.
rc = wait_text ("Done", 5, ret_text, ret_index, 0, 0, 500, 500, ret_bbox);
```

The final section of the test measures the time it took to perform the transaction. An **If** statement is defined to process the results of the **wait_text** function. If the message appeared inside the field, within the timeout, the first **end_transaction** statement records the time it took to perform the transaction. It also notes that the transaction passed. If the timeout expired, and the message did not appear, the transaction fails.

```
# End transaction.
if (rc == 0)
    end_transaction ( "OK", PASS );
else
    end_transaction ( "OK" , FAIL );
```

Part IX

Appendixes

A

Troubleshooting

The following paragraphs present solutions to some problems you may encounter when working with XRunner.

Starting XRunner

The XRunner window does not open and the message "system error" is displayed.
XRunner could not connect to its License Manager. The error message is misleading. Check that the MERCURY_ELMHOST environment variable points to the name of your License Manager machine and that the License Manager is running.

XRunner does not start on the standard MIT X11R4 server, or starts on the standard MIT X11R5, but no softkeys are available.
In order to run, XRunner needs special capture and execution "hooks" installed in the X server. This release of XRunner includes:

- An enhanced version of the MIT X11R4 server which provides the required hooks with the Input Synthesis Extension
- An enhanced version of the MIT X11R5 server which provides the required hooks with the DEC-XTRAP extension.

XRunner refuses to open and the system message "Unable to connect - XXX Program not registered" appears.
This problem can occur if XRunner is terminated in an incorrect manner. Check that another *xrun* process is not running. If you find such a process, kill it by entering the command:

kill -9 XRunner_process_id

The XRunner window comes up, but nothing is drawn in the window and XRunner hangs.
You must kill the xrun and crvx or crux processes in order to recover. Make sure that you do not have a file locking problem: Note any messages from the lock manager daemon when you kill XRunner. Try to set the configuration parameter XR_FILE_LOCKING in your *XRunner.cfg* file to FALSE. If XRunner starts correctly this time, set this parameter to FALSE for all users using XRunner.

Login

You login after a system crash and receive the message, "number of licenses has reached the limit."
This happens because it takes the license manager three minutes to update. Wait three minutes and then try to login again.

Record

An inappropriate type command is written to the test script when the record softkey is pressed.
This happens when the RECORD softkey is pressed shortly after the keyboard is activated. Keyboard events are cached in the X server, and are erroneously identified as part of user input in the current recording session. Wait about two seconds before pressing the Record button.

The display of a captured window is faulty.
XRunner verifies only windows which are at the front of the window stack. Always bring a window to the front before capturing it.

Your machine beeps several times when a window is captured.
This is normal and does not affect operations.

When learning the GUI (using recording or any other method), the physical description of objects learned is wrong (i.e., does not reflect the default recording of the learned object).
Run the command **csmode_init( );** This will reset all default attributes for all classes.

When a two-button mouse is used, pressing the right button is recorded as <kMiddle>. Pressing both buttons simultaneously is recorded as <kRight>.
Change the value of the XR_INP_MKEYS system parameter in the *xrunner.cfg* file, so that the value <kRight> is assigned to the right mouse button.

Running Tests

Execution hangs when playing sequences which cause long X server grabs.
You can recover by manually handling the grab situation. For example, if a pop-up window has grabbed the server and XRunner has failed to close it, you should try to close it manually. Note that using a Quick Run command instead of Run may prevent this problem. TSL grab and **ungrab** functions provide a reliable solution to the grabbing problem.

A programmed sequence which simulates a mouse drag does not work correctly.
Some applications do not accept such programmed actions as being equivalent to a mouse drag. Insert a move_locator_rel function call with a low *x, y* value (for example, 1,1) after the first mtype statement.

Using XRunner softkeys impairs xterm operation.
Try reassigning frequently used softkeys (for example, ABORT and CHECK WINDOW).

File Locking

When you try to run a test, XRunner displays the message: "Test <test_name> is locked by <user_name> with process ID XXXX ".
XRunner has a test locking system that prevents one user from modifying a test while another user is editing the same test. XRunner displays the message in one of two situations:

- when the test is simultaneously being edited by another user.
- when XRunner is abnormally terminated during test creation and the lock file is not removed.

First, if the file was locked by a user other than yourself, ensure that user is not currently editing the test.

If not, you can delete the lock file from the UNIX prompt, by issuing the following command:

rm <*test_name*>/lock

For example, if you attempted to open the test /home/xrunner/test/sample1 from XRunner and received an error message, enter

rm /home/xrunner/test/sample1/lock

The test will be unlocked. Try to open it from within XRunner.

Context Sensitive

You are unable to record or run tests, and the message "The XRunner version and the Context Sensitive Library version are not the same" appears in stderr.
You are trying to record or run an application that is linked with a Context Sensitive Library version which is not the same as the current XRunner version. Check both version numbers and make sure that they are identical.

Operations on pop-up menus do not execute properly (Motif only).
Sometimes, when you execute a test, the part of the script which should open a pop-up menu and select an item does not succeed. XRunner executes on standard Motif pop-up menus only.

Check that the mouse clicks that operate the pop-up menus conform to the standard Motif conventions. If not, you can often configure mouse buttons in order to solve this problem. For more information, contact Mercury Support.

Reading Text

XRunner has difficulty recognizing very small or italic fonts.
XRunner provides a test which automatically checks whether a font is

supported. This test is located under the pathname *$(M_ROOT)/lib/font_test*. This test must be called by an XRunner test which contains the line:

```
t= call font_test(font_name);
```

XRunner confuses characters which use the same bitmap.
This confusion may also arise when the active font group has several fonts with very similar characters. Change the XR_TEXT_CHECK_DIST configuration parameter to TRUE. This causes XRunner to check the distance of characters from the base line.

Online Help

The "Using Help" option in Hyperhelp does not work.
Before invoking XRunner, type at the prompt:

```
setenv HOHPATH $M_ROOT/dat
```

TSL

The Output redirection operators ">" and ">>" do not work correctly.
In some cases, the output files are not flushed; in all cases, files are created in the XRunner startup directory and not in the current directory. Make sure that you close files so that XRunner flushes file buffers.

An error message "Syntax error" occurs with a legal type statement.
You may have created a **type** statement with a variable which begins with one of the special type string character values (for example, - or +). Insert a "\", for example, `type("\" x)` to prevent the special interpretation of the first special character.

XRunner, and in some cases the X server, hang when a system command is executed.
You probably invoked XRunner in the background and used a system function to start a command which waits for *tty* input. This causes XRunner to block waiting for input. In some cases you get the message: "xrun (stopped on *tty* input)." Make sure that XRunner is not started as a background process.

Background Operation

When working with XR in background mode, you try entering input and receive the message "Suspended (tty input)."
At the command line, type fg (moves XRunner to foreground).

Bitmaps

XRunner does not display the selected bitmap.
Make sure that the standard X window system program *xwud* is in your path. XRunner bitmaps are in standard *xwd* format and XRunner uses the *xwud* program for bitmap displays. Also check the shell window from which you invoked XRunner for other error messages. A common message is "too many processes" when displaying several bitmaps. If this happens, close a few bitmaps and try displaying the bitmap you need again.

Command Line Interface

Parameters cannot be passed to a test called from the command line.
Use environment variables to store parameter values, then use the TSL **getenv** function to retrieve the desired values.

Network

XRunner will not connect to a remote machine.
Make sure that:

- you are trying to connect to a supported server.
- the DISPLAY environment variable is defined correctly and you have provided the correct values for the **-server** and/or the **-display** options.
- you have permission to connect to the remote machine. Use the *xhost* utility to set remote access permissions.
- if the remote machine runs the standard OpenWindows (X11/NeWS) server, verify that it was started with the parameters:

-noauth and -defeateventsecurity.

➤ the remote machine running OpenWindows (X11/NeWS) can be recorded only by another Sun machine.

User Interface

The XRunner window appears to come up without a cursor.
The cursor color and background color are the same. Open and close any XRunner form (such as the Test Header), and the cursor appears in a visible color.

HP Platforms

When using the HP X Terminal version B.04.01, the server hangs when XRunner is started.
This is caused by a bug in the HP server. Use an earlier server version (B.03.01).

UnixWare Platforms

Pressing F1 (UnixWare Help key) causes XRunner to hang.
Set the Help key to a different key (such as Alt F1) in the keyboard folder.

A filter jumps to a different on-screen location, then returns to its correct location.
Set the environment variable XRUN_FL_DELAY to an interval longer than the default, 0.65 (seconds). This variable sets the delay time (in seconds) for placing the filter in the location where it is defined.

B

Configuring Your Keyboard

This chapter describes:

- Defining Global Key Aliases
- Defining Platform-Specific Key Aliases

About Configuring Your Keyboard

In XRunner 4.0 and higher, all keyboard data is acquired from the Xserver and not (as in previous releases) from a specific keyboard file.

The change in keyboard mechanism may create conflict, for instance, when you run tests created with XRunner 3.0 on XRunner 4.0, or when transporting test scripts across platforms which use different keyboards.

By modifying the key definitions contained in alias files, you allow XRunner to identify a key by several names. For example, you can define ESC and Esc as aliases for the Escape key, so that when programming, you can use the abbreviations without causing errors.

The following example illustrates how key aliases may be integrated in your tests:

Suppose that while recording a test on a Sun machine, you repeatedly hit the Line Feed key to advance to the next line. In the script sequence generated, the type statement includes the name LineFeed.

At a later date, you decide to play back this test on an HP machine. However, the HP does not have a LineFeed key. When the type("<kLineFeed>") statement is reached, execution stops, and an error message is generated indicating "No such key."

You can prevent this situation by assigning the alias LineFeed to the Return key in the HP platform-specific keyboard configuration file. During test execution on the HP machine, XRunner will press Return rather than LineFeed to advance to the next line.

The files containing key aliases are defined by two configuration parameters:

- XR_KBD_ALIAS_FILE for defining global (cross-platform) aliases
- XR_MACHINE_DB_NAME for defining platform-specific aliases

Defining Global Key Aliases

You use the XR_KBD_ALIAS_FILE configuration parameter to define global key aliases. The default file is located in $M_ROOT/dat.

If your test requires a specific key which is not defined in the alias file, add a new alias to the default file or create your own alias file. Ensure the XR_KBD_ALIAS_FILE configuration parameter is defined for the correct file. Then restart XRunner.

The data in alias files is organized according to the following format:

<KeySymName/Key Code> <key_name> <shift_name> <aliases>

for example:

# keysym-name/ key-code	key-name	shift-name	aliases
XK_F29	F29	S_F29	PgUp
XK_KP_7	KP_7	S_KP_7	Home
XK_Insert	Insert	S_Insert	Ins
XK_KP_0	KP_0	S_KP_0	KP_Insert
XK_KP_1	KP_1	S_KP_1	End
XK_KP_9	KP_9	S_KP_9	Prior
XK_Multi_key	Multi_key	S_Multi_key	Compose
XK_Num_Lock	Num_Lock	S_Num_Lock	NumLock
XK_Alt_Graph	Alt_Graph	S_Alt_Graph	script_switch

The following paragraphs describe the data in each column.

KeySyms/Key Code

The first column of the file specifies the key code. This can be one of two codes: a KeySym, or a key code.

In the alias file supplied with XRunner, each key is identified by a KeySym. A KeySym is a name that the X Server assigns to each key. Through KeySyms, XRunner identifies most keys on most platforms. The format of a KeySym is a string starting with the characters XK_.

Note that the KeySym string used to identify a key does not necessarily resemble, the name actually printed on the key.

Alternatively, the Code column can specify the absolute, numeric code associated with any given key. This code may be expressed as a decimal, octal, or hexadecimal value. For example, the RETURN key on a keyboard may be represented by one of the following scan codes:

decimal	25
octal	O33
hexadecimal	0x5A

Note that scan codes represented in octal notation are always preceded by the prefix "O" (uppercase o). Scan codes represented in hexadecimal notation are preceded by the prefix "0x" (zero-lowercase x).

The keyboard definition file must not assign the same code (KeySym or Scan Code) to more than one physical key.

Formal Name

The second column specifies the formal name. The formal name of a key is the character string that will be generated in the TSL script when this key is activated during a record session. Conversely, when this code is encountered by the XRunner interpreter during test execution, the appropriate input will be sent to the AUT.

The formal name may consist of one to twenty characters: letters (upper- and lowercase), digits, and underscores. The first character must be a letter or underscore.

By editing the keyboard definition file you can configure the TSL code generated when specific keystrokes are entered. However, if you intend to edit the key name, you should do so before generating tests. In this way the code appearing in your test scripts will always be played back correctly during test execution.

The keyboard definition file must not assign the same name to more than one physical key.

Shift

The third column specifies the shift attribute of the key. This shift attribute is the value associated with the key when it is pressed in the Shift mode (the SHIFT key is held down). As with the key name, the indicated value may consist of one to twenty characters: letters (upper- and lowercase), digits, and underscores. The first character must be a letter or underscore.

The keyboard definition file must not assign the same shift to more than one physical key.

Alias

The fourth column specifies the alias(es) defined for each key. The keyboard configuration files supplied with XRunner already include aliases required to make tests compatible across supported machines.

Note that you cannot assign an alias which is already used as the formal name or as the alias of another key. To assign several aliases to the same key, separate the aliases with commas.

Defining Platform-Specific Key Aliases

XR_MACHINE_DB_NAME points XRunner to the appropriate keyboard and configuration files for the specific platform or server. You modify aliases contained in this file, when you want the alias to apply only to a specific platform. By default, the alias file defined for this parameter resides in $M_ROOT/dat.

The data in a platform-specific alias files is organized according to the following format:

PLATFORM/SERVER <platform_name> <kbd_file> <cfg_file> <alias_file>

for example:

PLATFORM/SERVER	Platform Name	Kbd File	Cfg File	Alias File
PLATFORM	sun	-	generic.cfg	sun_alias
PLATFORM	hp	-	generic.cfg	hp_alias
SERVER	gold:0	-	generic.cfg	sun_type_alias

The following paragraphs describe the data in each column.

PLATFORM/SERVER

The first column can specify either PLATFORM or SERVER. For instance, if you work on Sun and on some X servers, use a type 4 keyboard, and on others a type 5 keyboard, then you can specify a different keyboard configuration file for each one.

The Platform Name

The second column can specify: sun, ibm, hp, dec, solaris, visual, sgi, dg, ncr, sco, or any server name.

Kbd File

The third column indicates the corresponding keyboard file per platform. If you are not planning to work in this way, you can replace the Kbd file with a hyphen.

Cfg File

The next column specifies a configuration file for each platform. This file is important if you are planning to run XRunner on one platform, and check an application running on another platform. When you invoke XRunner with the **-server** option, XRunner reads the necessary keyboard file and configuration file for that platform, thus ensuring correct test execution.

Alias File

The fifth column indicates the corresponding alias file per platform.

Keyboard Error Checking

When XRunner comes up, it reviews the alias file and the keyboard, and reports problems it identified to the error message file. This message file is called *xr_kbd.err* and is created in the current directory.

You can use the information provided in the error file to update the alias configuration file before the next time you work with XRunner.

C

External Utilities for Bitmap Capture/Check/Display

Some platforms and applications work with bitmaps in non-standard formats. XRunner allows you to use your own utilities to capture, check, and display bitmaps.

This chapter describes the following:

- Capture Utility
- Compare Utility
- Display Utility

About External Bitmap Utilities

XRunner allows you to use your own external utilities for capturing, checking and displaying bitmaps. The mode that XRunner uses to handle bitmaps is defined by the XR_IMAGE_MODE configuration parameter. By default, the image mode is set to *xwd*. The image mode can also be set to *gl* and *external*.

If XR_IMAGE_MODE is set to *xwd* or *gl*, XRunner uses its own built-in code to capture, check and display bitmaps. For more information, see Chapter 33, "Changing System Defaults."

To use external utilities instead of XRunner's built-in utilities, you must first set the XR_IMAGE_MODE configuration parameter to *external*. Then, specify the path of each of the three utilities you want to use, using the appropriate configuration parameters. These are the XR_CAPTURE_UTIL, XR_COMPARE_UTIL and XR_DISPLAY_UTIL configuration parameters.

Capture Utility

This utility captures a bitmap and saves it to a file.

status = my_capture **-id** <*window id*> **-rect** <*x y w h*> **-display** <*display name*>
[**-compress**] **-out** <*dump file name*>

-id specifies the X Window ID of the window to be captured.

-rect specifies the coordinates that define a bitmap area rectangle relative to the window origin. The *x* and *y* parameters are the pixel offsets from the origin of the window; w and h are the width and height of the rectangle. The special values -1,-1,-1,-1 indicate the full window.

-display specifies the name of the display used for the window. If the display parameter is not provided, the current display is used.

-compress specifies that the utility will compress the bitmap file. The compression method can be the standard UNIX compression format or any proprietary format. The only limitation is that the display and compare utilities must be able to identify the bitmap as compressed.

-out specifies the full pathname of the bitmap file to be written. The only modification the utility may make to this name is the addition of a known extension identifying it as a compressed file.

The utility's return status is 0 for success and -1 for failure. In case of failure, error messages are printed to *stderr*.

Compare Utility

This utility compares a window on screen with a bitmap file on disk.

status = *my_compare* **-id** <*window id*> **-rect** <*x y w h*> **-display** <*display_name*>
[**-compress**] **-in** <*bitmap file*> [**-act** <*actual file name*>] [**-diff** <*diff file name*>]
-min_diff <n> [**-filter** <*filter file*>]

-id specifies the X Window ID of the window to be compared.

-rect specifies the coordinates that define a bitmap area rectangle relative to the origin of the window. The *x* and *y* parameters are the pixel offsets from the origin of the window; w and h are the width and height of the rectangle. The special values -1,-1,-1,-1 indicate the full window.

-display specifies the name of the display used for the window. If the display parameter is not provided, the current display is used.

-compress specifies that the utility will compress the bitmap file. The compression method can be the common UNIX compression format or any proprietary format. The only limitation is that the display and compare utilities must be able to identify the bitmap as compressed.

-in specifies the full pathname of the expected bitmap file. This is the file captured previously by the capture utility. If this file is in compressed format, then it must be uncompressed prior to comparison, and then compressed again.

-act specifies the full pathname of the actual bitmap file in case of a mismatch.

-diff specifies the full pathname of the difference bitmap file in case of a mismatch. Each pixel in the bitmap which is not a difference pixel must have a special value identifying it as such.

-min_diff specifies the number of pixels that can be different without constituting a mismatch.

-filter is a name of a file containing a representation of the Area of Interest. Each character in this file represents a pixel in the bitmap. The bitmap is laid out row by row, starting with the top left pixel. Each pixel filtered out (i.e., not compared) has a corresponding character with the value 1. Each pixel compared has a corresponding character with the value 0.

The utility's return status is 0 for success and 1 for a mismatch. In case of failure, error messages are printed to stderr, and the return value is -1.

Display Utility

This utility displays a window on screen.

my_display [**-bw**] **-display** <*display*> **-title** <*window title*> <*file_name*>

-bw indicates that a difference bitmap is displayed in black and white. Each pixel in the bitmap which is a difference pixel is displayed as black (or the darkest color available). Each pixel in the bitmap which is a non-difference pixel is displayed as white (or the lightest color available).

-display specifies name of the display used to display the bitmap. If the display name is not provided, the current display is used.

-title is the title to be displayed in the window banner. For <*filename*>, specify the full pathname of the bitmap file to be displayed. This is the file captured previously by the capture utility, in a known format. If the file is in compressed format and must be uncompressed for display, then the file must be compressed again following display.

The utility's return status is 0 for success. In case of failure, error messages are printed to *stderr*, and the return value is -1.

D

Support for Servers With No Record/Replay Extension

This chapter describes the *xextend* utility.

Ordinarily, XRunner provides record and replay capabilities using either the XTestExtension1, DEC-XTRAP or Record (R6) standard extensions. (Users working with Sun NeWS servers enjoy the same support without any of these two extensions being explicitly present.)

A further alternative is the xgate server. You can find xgate in the /bin subdirectory of your installation directory.

If none of these is suitable, you can use the *xextend* utility.

The xextend Utility

Xextend is based on the public domain utility *xmond*. *Xextend* sits transparently between the X clients and an X server. To the clients it behaves exactly like an X server; to the server, it behaves exactly like a number of X clients.

When you connect your AUT and XRunner to *xextend*, the application sees *xextend* as a real server to which it is connected. This server appears to have an additional extension called XREC. The XREC extension (emulated by *xextend*) allows XRunner to record the client application connected to *xextend*. The operation of the application connected to *xextend* is not affected in any way while XRunner connected to *xextend* detects the additional XREC extension.

No problem results if one of the standard extensions is already present when you connect your application to *xextend*. The standard extension is used,

and XREC is simply ignored. If the XREC extension introduced by *xextend* is the only available recording extension, XRunner uses it for its operation.

Using xextend

To connect an application to *xextend*, enter the following command at your system prompt:

xextend [*options*]

The xextend options are as follows:

-server *display_name*

This option sets the X display to which *xextend* connects for record purposes.

-port *display_number*

This option sets the port on which *xextend* listens for client communications. This port is always on the host on which *xextend* is running. The default is 1. The *display_number* entered in the command line used to invoke the application must be the same as the display number specified by the **-port** option to *xextend*. If no other value is indicated, you must enter the default 1 as the *display_number*.

Then invoke the application to be connected as follows:

application_name **-***display* <*display_name*> **:***display_number*

Note that in addition to the AUT, XRunner must also be individually invoked and connected to *xextend*. While not mandatory, it is recommended to connect all applications (including the window manager) to *xextend*.

E

Using ToolTalk with XRunner

XRunner (on SunOS/Solaris platforms) supports the ToolTalk service, which allows applications to communicate with each other. For detailed information about how to install and use ToolTalk, refer to *The ToolTalk Service: An Inter-Operability Solution*.

XRunner supports three forms of ToolTalk communication:

- Requests from external applications to perform XRunner file operations and run commands. For example, a configuration control application can retrieve the latest version of a particular XRunner test, and from within the application run the specified test on XRunner.
- XRunner notices regarding file operations and run commands, which are automatically sent to external applications. For instance, a test management application can monitor XRunner's run commands and automatically keep track of which tests were run.
- User-defined ToolTalk messages which are sent from within a TSL test script. For example, an XRunner user can send a message to a configuration-control application and request that it retrieve a certain test.

Note: A demo application that communicates with XRunner can be found under $M_ROOT/samples/tooltalk. The README file under this directory describes the demo application.

Invoking XRunner in ToolTalk Mode

To allow ToolTalk communication with XRunner, you must invoke XRunner in ToolTalk mode. In your *.xrunner* configuration file, define the XR_TOOLTALK system parameter as follows:

XR_TOOLTALK = TRUE

To invoke XRunner in ToolTalk mode through the command line interface, use the **-tooltalk** option.

Requests From External Applications

XRunner handles ToolTalk requests from external applications to perform file operations and run commands. The table below lists the operations that XRunner supports. At the end of this section you can find examples for each of the available requests.

In terms of the scope of requests, XRunner supports the TT_SESSION (the default) and TT_FILE_IN_SESSION options. If you use TT_FILE_IN_SESSION (so that only an XRunner with a specific test receives the request), you should define the test name using the tt_message_file_set function.

The message arguments that appear in the table are:

Operation: The operation to be performed by XRunner. (Inserted in the tt_prequest_create and tt_notice_create ToolTalk functions).

Arg mode: One of two argument modes: TT_IN or TT_OUT. (Inserted in the tt_message_arg_add ToolTalk function.)

Arg type: Integer or string. (Inserted in the tt_message_arg_add ToolTalk function.)

Arg content: The content of the argument. (Inserted in the tt_message_arg_add ToolTalk function.) The "Force" option indicates that

the operation should be performed while ignoring any currently unsaved changes.

Operation	Description	Arg Mode	Arg Type	Arg Content
XR_new_test	Create a new test	TT_IN TT_OUT	integer string	0 = do not force 1 = force Testname (temporary test name)
XR_open_test	Open an existing test	TT_IN TT_IN	string integer	Expected results directory 0 = do not force 1 = force
XR_save_test	Save current test	–	–	–
XR_save_as_test	Save as the current test	TT_IN TT_IN	string integer	New test name 0 = do not force 1 = force
XR_close_test	Close current test	TT_IN	integer	0 = do not force 1 = force
XR_get_test_ name	Get the current test name	TT_OUT	string	Current test name
XR_get_mode	Return XRunner's current mode	TT_OUT	integer	Current mode: 0 = idle 1 = Analog record 2 = Context Sensitive record 3 = run 4 = verify
XR_get_sysvar	Get a system variable value	TT_IN TT_OUT	string string	Configuration item Configuration item value

Operation	Description	Arg Mode	Arg Type	Arg Content
XR_record	Start recording	TT_IN	integer	Recording level: 0 = Analog 1 = Context Sensitive
XR_record_stop	Stop recording			
XR_replay	Start execution	TT_IN TT_IN	integer integer	Run mode: 0 = quick run 1 = run 2 = step 3 = step into Line number (0 = execute from current line position.)
XR_abort	Abort test execution			
XR_pause	Pause test execution			
XR_set_verify	Set verify mode on/off	TT_IN	string	Verification results directory (full path name)

The following examples illustrate the usage of each of the available requests.

XR_new_test

```
msg = tt_prequest_create(TT_SESSION, "XR_new_test");
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 0, force);
tt_message_arg_add(msg, TT_OUT, "string", NULL);
```

XR_open_test

```
msg = tt_prequest_create(TT_SESSION, "XR_open_test");
tt_message_arg_add(msg, TT_IN, "string", testname);
tt_message_arg_add(msg, TT_IN, "string", exp_name);
```

```
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 2, force);
tt_message_send(msg_out);
```

XR_save_test

```
msg = tt_prequest_create(TT_SESSION, "XR_save_test");
tt_message_send(msg_out);
```

XR_save_as_test

```
msg = tt_prequest_create(TT_SESSION, "XR_save_as_test");
tt_message_arg_add(msg, TT_IN, "string", testname);
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 1, force);
tt_message_send(msg_out);
```

XR_close_test

```
msg = tt_prequest_create(TT_SESSION, "XR_close_test");
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 0, force);
tt_message_send(msg_out);
```

XR_get_test_name

```
msg = tt_prequest_create(TT_SESSION, "XR_get_test_name");
stat = tt_message_arg_add(msg, TT_OUT, "string", NULL);
tt_message_send(msg_out);
```

XR_get_mode

```
msg = tt_prequest_create(TT_SESSION, "XR_get_mode");
tt_message_arg_add(msg, TT_OUT, "integer", NULL);
tt_message_send(msg_out);
```

XR_get_sysvar

```
msg = tt_prequest_create(TT_SESSION, "XR_get_sysvar");
tt_message_arg_add(msg, TT_IN, "string", sysval);
tt_message_arg_add(msg, TT_OUT, "string", NULL);
tt_message_send(msg_out);
```

XR_record

```
msg = tt_prequest_create(TT_SESSION, "XR_record");
tt_message_arg_add(msg, TT_IN, "integer", NULL);
```

```
tt_message_arg_ival_set(msg, 0, recording_level);
tt_message_send(msg_out);
XR_record_stop
msg = tt_prequest_create(TT_SESSION, "XR_record_stop");
tt_message_send(msg_out);
```

XR_replay

```
msg = tt_prequest_create(TT_SESSION, "XR_replay");
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 0, run_mode);
stat = tt_message_arg_add(msg, TT_IN, "integer", NULL);
stat = tt_message_arg_ival_set(msg, 1, line_no);
tt_message_send(msg_out);
```

XR_abort

```
msg = tt_prequest_create(TT_SESSION, "XR_abort");
tt_message_send(msg_out);
```

XR_pause

```
msg = tt_prequest_create(TT_SESSION, "XR_pause");
tt_message_send(msg_out);
```

XR_set_verify

```
msg = tt_prequest_create(TT_SESSION, "XR_set_verify");
tt_message_arg_add(msg, TT_IN, "string",  result_directory);
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 1, on_off);
tt_message_send(msg_out);
```

The following example illustrates how a configuration control application uses ToolTalk to request XRunner to open the latest version of a specified test.

```
/* get the test name*/
cc_get_test_name(test);
/* create a message for XRunner to open test*/
msg = tt_prequest_create(TT_FILE_IN_SESSION, "XR_open_test");
/* add expected results directory argument*/
tt_message_arg_add(msg, TT_IN, "string", "exp3.2");
```

```
/* Open test even if an unsaved test is currently open*/
tt_message_arg_add(msg, TT_IN, "integer", NULL);
tt_message_arg_ival_set(msg, 1, 1);
/* set the test name*/
tt_message_file_set(msg, test);
/* add callback to handle return codes*/
tt_message_callback_add(msg, cc_msg_callback);
/* send the message*/
tt_message_send(msg);
```

This can be followed by a request to run the specified test:

```
...
msg = tt_prequest_create (TT_FILE_IN_SESSION, "XR_replay");
tt_message_file_set(msg, test);
tt_message_arg_add(msg, TT_IN, "integer", 0);
/* select run mode*/
tt_message_arg_add(msg, TT_IN, "integer", 1);
/* begin at the first line*/
tt_message_callback_add(msg, cc_msg_callback);
tt_message_send(msg);
...
```

XRunner Notices

When invoked in ToolTalk mode, XRunner automatically sends the following notices regarding file and run operations:

Notice	Description	Files
XR_created_new_test	A new test was created.	
XR_opened_test	A test was opened.	Test name
XR_saved_test	A test was saved.	Test name
XR_saved_as_test	A test was saved as.	New test name
XR_closed_test	A test was closed.	Test name

Notice	Description	Files
XR_started_recording	XRunner started recording.	Test name; "noname" for new test
XR_stopped_recording	XRunner stopped executing.	Test name
XR_started_replay	XRunner started executing.	Test name
XR_stopped_replay	XRunner stopped replaying due to end of test, step or breakpoint.	Test name
XR_paused	Test execution paused.	Test name
XR_aborted	Test execution aborted.	Test name
XR_changed_verify_mode	Verify mode was set to on/off	

The following example illustrates how a test management application can wait for XRunner to complete test execution:

```
/* create observation pattern, so that ToolTalk knows we */
/* are interested in XR_stopped_replay messages within */
/* the session*/
pat = tt_pattern_create();
tt_pattern_category_set (pat, TT_OBSERVE);
tt_pattern_scope_add (pat, TT_SESSION);
tt_pattern_op_add (pat, "XR_stopped_replay");
tt_pattern_register (pat);
/* receive message*/
msg_in = tt_message_receive ();
mark = tt_mark ();

/* handle end of replay message*/
if (!strcmp (tt_message_op (msg_in), "XR_stopped_replay"))

/* test management function to handle end of execution*/
tm_handle_end_of_replay ();
```

ToolTalk Messages from within TSL

You can send two types of ToolTalk messages from within an XRunner test script: notices and requests. Using TSL built-in functions, you first create a notice or a request, add attributes and arguments, and then send it.

When you send a request, XRunner waits the specified number of seconds for the response to the request. You can then examine the value of any of the message's attributes or arguments. Note that it is your responsibility to destroy the message when it is no longer needed. (The maximum number of messages you can create is 20.)

Below are the descriptions of notices and requests that XRunner supports. At the end of this section you can find two examples that illustrate the use of notices and requests.

notice **= tooltalk_create_notice (**"*operation*"**);**
Creates a tooltalk notice, with the following default attributes:

class:	TT_NOTICE
address:	TT_PROCEDURE
scope:	TT_SESSION
operation:	*operation*

Returns a number between 1 and 20, or 0 for an error.

request **= tooltalk_create_request (**"*operation*"**);**

Creates a tooltalk request with the following default attributes:

class:	TT_MESSAGE
address:	TT_PROCEDURE
scope:	TT_SESSION
operation:	*operation*

Returns a number between 1 and 20, or 0 for an error.

tooltalk_set_attr (*msg*, $attr_1$, val_1, $attr_2$, val_2, .., $attr_n$, val_n**);**

Sets the attributes of a message.

msg The notice or request for which attributes are set.

Attributes are set using pairs of attribute-value arguments. Attributes can be one of the following: "address", "scope", "operation", "file", "object", "otype," "handler", "handler_ptype", "disposition", "session".

Note that attribute values should be set as strings. For example:

```
tooltalk_set_attr (msg, "scope", "TT_SESSION");
```

tooltalk_set_arg (*msg*, *arg_number*$_1$, *mode*$_1$, *type*$_1$, *value*$_1$, ... **);**

Sets the arguments of a message.

msg	The notice or request for which arguments are set.
arg_number	The argument number. The first argument should be 0 and additional arguments must have consecutive numbers. If the same arg_number appears twice in a the function, the last appearance is the one used.
mode	One of the following integers: 1 for TT_IN, 2 for TT_OUT, 3 for TT_INOUT. You may also predefine constants in your startup test.
type	The type of argument: either "integer" or "string."
value	The argument value. Use 0 or "" for arguments of the type TT_OUT.

Note that currently you can set a maximum of 10 arguments per message.

tooltalk_send_notice (*notice***);**

Sends a notice.

notice The notice to be sent.

state = **tooltalk_send_request (***request*, *timeout***);**

Sends a request and waits <*timeout*> seconds for a response.

state	Indicates the state of the request: (-1) if timeout was reached before a reply was received, (0) for TT_HANDLED, (1) for TT_FAILED.

request	The request to be sent (integer).
timeout	The number of seconds XRunner should wait for the request to return.

status **= tooltalk_get_attr (***message*, *attribute*, *value***);**

Returns the value of an attribute of a message.

status	The variable that stores the return value of the function (0 if succeeds;1 if fails).
message	The name of the message (integer).
attribute	The attribute to be returned (string). Valid attributes are: "address", "scope", "operation", "file", "object", "otype", "handler", "handler_ptype", "disposition", "session", "status", "status_string".
value	The variable that will store the value of the attribute (string).

status **= tooltalk_get_arg (***message*, *arg_number*, *value***);**

Returns the value of an argument of a message.

status	The variable that stores the return value of the function (0 if succeeds;1 if fails).
message	The name of the message (integer).
arg_number	The number of the argument.
value	The variable that will store the value of the argument (string).

tooltalk_destroy_msg (*message***);**
Destroys a message.

message	The name of the message to be destroyed (integer).

session = **tooltalk_get_session** ();
Returns the default ToolTalk session.

In the following example, a notice (check_notice) is sent from within a test script to a test management application. The notice is sent following a **check_window** function, and notifies the test management application about the results of the function.

```
# Perform check_window*/
return_status = check_window (1,"Win_1","mainwin",
                              150,150,74,74);

# Create notice*/
check_notice = tooltalk_create_notice ("TM_check");
if (check_notice) {

    # Set notice arguments to includes the result of
    # the check window function*/
    tooltalk_set_argument(check_notice,
    0, TT_IN, "integer", return_status,
    1, TT_IN, "string", "check_window");
    tooltalk_send_notice(check_notice);
    tooltalk_destroy_msg(check_notice);
}
```

In the next example, a request is sent to a configuration control application to get the latest version of the test, before the test is called.

```
function get_test (in testname)
{
    auto ret_val = 1;
    auto status = 1;
    auto test_req = 0;
    test_req = tooltalk_create_request ("CC_get_test");
    if (test_req) {
        tooltalk_set_arg (test_req,0,TT_IN,"string",testname);
        ret_val = tooltalk_send_request (test_req, 30);
        if (!ret_val) {
            ret_val = tooltalk_get_attr (test_req,"status",
                                status);
        }
        tooltalk_destroy_msg (tst_req);
    }
    if (!ret_val &&! status)
```

```
        return (0);
    if (ret_val)
        return (SEND_FAILED);
    if (status)
        return (GET_TEST_FAILED);
}
```

Error Messages

Following are the error messages for ToolTalk notices and requests sent from within a test script. Error messages can be received as TSL errors, or as the *status* attribute of a message.

TSL Error	Status Attribute	Error
-10201	1550	Illegal parameter
-10202	1551	File parameter is incorrect
-10203	1552	XRunner's mode is incorrect for this operation
-10204	1553	Illegal command for this test
-10205	1554	Test has unsaved changes
-10206	1555	Argument's value is out of bounds
-10207	1556	Argument's value is incorrect for this function
-10208	1557	Supported number of messages exceeded
-10209	1558	Message does not exist
-10210	1559	Tooltalk Error
-10211	1560	An XRunner form is up, preventing operation

F

Using XRunner with SoftBench

XRunner is one of the integrated HP SoftBench tools you can use for developing and testing software.

This chapter describes:

- The SoftBench-XRunner User Interface
- Communicating With Other SoftBench Tools

About Using XRunner with HP SoftBench

XRunner is fully integrated with the SoftBench set of development tools. This allows you the convenience, for instance, of:

- invoking XRunner from the SoftBench ToolBar
- running an XRunner test from the SoftBench Development Manager, Debugger, or Editor
- building an XRunner executable in the SoftBench Program Builder

Interaction between XRunner and other SoftBench tools is enabled at installation. To assist you, the XRunner icon appears in the SoftBench ToolBar and an XRunner menu item is included in the main window of several SoftBench tools.

The communication between XRunner and other SoftBench tools takes place via BMS (Broadcasting Message Server) messages. A full list of messages supported by XRunner appears in this chapter. You use these messages the same way you use standard BMS messages.

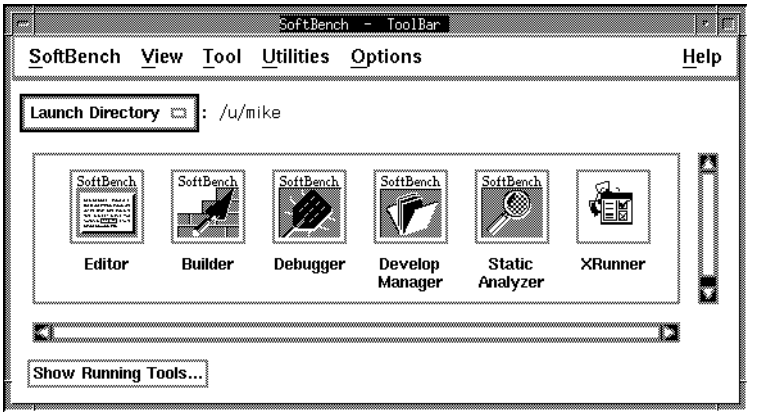
In addition, the **send_message** TSL function allows you to send BMS messages from an XRunner test script.

The SoftBench-XRunner User Interface

When you install the integrated SoftBench-XRunner package, XRunner appears in the SoftBench user interface.

SoftBench ToolBar

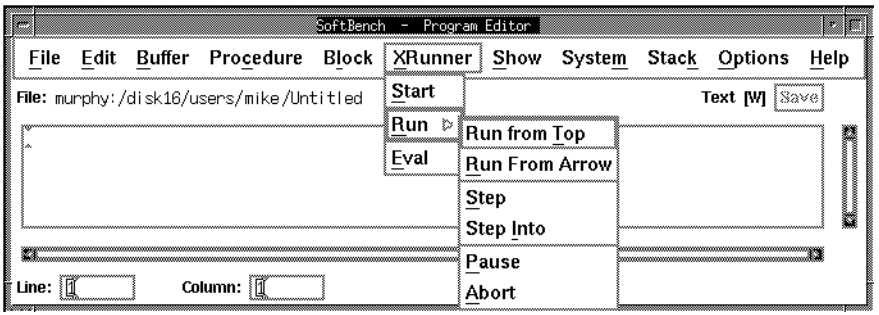
XRunner is one the tools you can invoke from the SoftBench ToolBar:



Select ToolBar Setup from the Options menu. Select XRunner from the Available Tools list, then click Add to ToolBar. An XRunner icon appears in the ToolBar window.

SoftBench Program Editor

XRunner appears, with slight variations, in the menu-bar of many SoftBench tools. For example, this is how the XRunner menu appears in the SoftBench Program Editor:



➤ Click Start to invoke XRunner.

➤ Click Run to display a menu of XRunner Run commands. For a full description of these commands, see Chapter 24, “Running Tests.”

➤ Click Eval to instruct the TSL interpreter to evaluate a selected line in the editor.

SoftBench Development Manager

In the SoftBench Development Manager window, the XRunner menu contains the Open command for opening tests and the Run commands for running an open test.

XRunner tests are named with a .tst extension. Double-clicking on a test starts XRunner if it is not up and loads the test. When you click an XRunner test, the Actions menu displays two options: Browse and Open Test.

XRunner executables are named with an .xrun extension. An XRunner executable is an application which has been linked with Mercury Interactive’s Context Sensitive libraries and is ready for testing with XRunner. Double-clicking on a .xrun file invokes the executable.

SoftBench Program Builder

You can use the Program Builder tool to generate XRunner executables. Simply add a .xrun extension to the target name and press Build. During the build, SoftBench automatically links the target with Mercury Interactive's Context Sensitive libraries. The executable is ready for testing with XRunner.

SoftBench Debugger

The XRunner menu in this window contains

- a Start command for invoking XRunner
- a Run menu containing XRunner Run commands. For a full description of these commands, see Chapter 24, "Running Tests."

Communicating With Other SoftBench Tools

The communication between XRunner and other SoftBench tools takes place via BMS (Broadcasting Message Server) messages. Communication can take place on three levels:

- Requests from SoftBench tools to perform XRunner file operations and run control commands. For example, a configuration management application can retrieve the latest version of a particular XRunner test, and from within the application, run the specified test in XRunner.
- XRunner notices regarding file operations and run control commands that are automatically sent to SoftBench tools. For instance, a test management application can monitor XRunner's run commands and automatically keep track of which tests were run.
- User-defined BMS messages that are sent from within a TSL test script. For example, an XRunner user can send a message to a configuration management application and request that it retrieve a certain test.

SoftBench Request Messages

XRunner handles BMS requests from other SoftBench tools to perform file operations and run commands.

These messages are displayed in the SoftBench Message Monitor. The table below lists the operations that XRunner supports. For more information, see the *HP SoftBench Programmer's Reference*.

The message arguments are:

- **Action:** The operation to be performed by XRunner.
- **Data:** The information needed to perform a requested action. The data can be used as the DATA instance in the **send_message** TSL function (see above section).

Data type can be an integer or string, depending on the Data argument.

Content describes the options available. The "force" option indicates that the action should be performed while ignoring any currently unsaved changes.

Action	Description	Data	Data type	Data content
R NORMALIZE	Normalize XRunner window	-	-	-
R ICONIFY	Iconify XRunner window	-	-	-
R START	Start XRunner	-	-	-
R STOP	Quit XRunner	-	-	-
R XR_new_test	Create a new test	force (optional)	int	0 = do not force (default) 1 = force

Action	Description	Data	Data type	Data content
R XR_open_test	Open an existing test	testname exp force (optional)	string string int	Full path of test Expected results directory 0 = do not force (default) 1 = force
R XR_save_test	Save current test	-	–	–
R XR_save_as_test	Save as the current test	testname force (optional)	string int	Full path of new test 0 = do not force (default) 1 = force
R XR_close_test	Close current test	force (optional)	int	0 = do not force (default) 1 = force
R XR_record	Start recording	mode	int	Recording level: 0 = Analog 1 = Context Sensitive
R XR_record_stop	Stop recording	-	-	-
R XR_run	Run test	mode line	int int	Run mode: 0 = Run From Top 1 = Run From Line 2 = Step 3 = Step Into Line number (0 = run from current line position-default)

Action	Description	Data	Data type	Data content
R XR_open_run	Combine XR_run and XR_open_test	testname	int	Full path of test. For this option, you must also supply the expected results directory
		mode	int	Run mode: 0 = Run From Top 1 = Run From Line 2 = Step 3 = Step Into
		line	int	Line number (0 = run from current line position-default)
R XR_set_verify	Set verify mode on or off	on	int	0 = off (default) 1 = on
		force (optional)	int	0 = do not force (default) 1 = force (for this option, you must specify a results directory)
		dirname	string	Verification Results directory (full path name)
R XR_abort	Abort test execution	-	-	-
R XR_pause	Pause test execution	-	-	-

Action	Description	Data	Data type	Data content
R XR_eval	Evaluate a statement in the XRunner interpreter	statement	string	
R XR_lower	Lower the XRunner window	-	-	-
R XR_raise	Raise the XRunner window	-	-	-
R XR_move_ execution_marker	Move the execution marker	line	int	Line number (0 = move to first line-default)
R XR_move_ cursor	Move the cursor	line	int	Line number (0 = move to first line-default)

XRunner Notification Messages

The following notices are sent by XRunner during a file operation or run command:

Notices	Description	Files
R XR_connected	Communication established.	-
R XR_ready	XRunner is ready to receive requests.	-
R XR_opened_test	A test was opened.	Test name
R XR_saved_test	A test was saved.	Test name
R XR_saved_as_test	A test was saved as.	New test name
R FILE_MODIFIED (SoftBench notice)	Message sent in addition to XR_saved_test or XR_saved_as_test	-
R XR_closed_test	A test was closed.	Test name
R XR_started_recording	XRunner started recording.	Test name; “noname” for new test
R XR_stopped_recording	XRunner stopped recording.	Test name
R XR_started_run	XRunner started test execution.	Test name
R XR_stopped_run	XRunner stopped test execution due to end of test, step or breakpoint.	Test name
R XR_paused	Test execution paused.	Test name
R XR_aborted	Test execution aborted.	Test name
R XR_changed_verify_mode	Verify mode was set to on/off.	Test name
R XR_evaled	Evaluation request processed	-
R XR_exited	XRunner exited.	-

Sending BMS Requests and Notifications From a Test Script

You can use the **send_message** TSL function in an XRunner test script to send BMS messages to other SoftBench tools.

The **send_message** function takes a single parameter demarcated by double quotation marks. It can receive two sets of arguments, depending on whether the message you are sending is a notification or a request.

Notification Messages

The syntax is as follows:

send_message ("*N action [operand] [status] ['data'] ['replydata']*" **);**

N	Indicates a notification message.
action	A user-defined notification action.
operand	(Optional) You may pass NULL as an argument. For XRunner's notifications, the value is DISPLAY.
status	(Optional) Either PASS or FAIL. The default is PASS.
data	(Optional) A set of data enclosed in single quotation marks. See the SoftBench documentation for a full list of possible BMS notification messages. XRunner notification messages are described in this chapter.
replydata	(Optional) A set of data enclosed in single quotation marks. Typically, an error message is passed if STATUS is FAIL.

For example, suppose you wish to enhance XRunner's notifications by sending a BMS message informing other SoftBench tools that a **check_window** command failed.

In your test script, you might program the following lines:

```
if (check_window(1, "Win1", "XBur", -1, -1, -1, -1) == 0) {
    send_message("N XR_check host:0 FAIL '/tmp/test7 1' 'check Failed.'");
}
```

Here, the OPERAND is set to equal DISPLAY and DATA is the testname and the line number of the **check_window** statement.

For more information on the parameters used in the **send_message** function, please refer to the *HP SoftBench Programmer's Reference*.

Request Messages

The syntax is as follows:

send_message (" *R toolclass action [operand] [data]* " **);**

R	Indicates a request message.
toolclass:	The SoftBench tool to which the request belongs, for example, EDIT.
action	The action requested.
operand	(Optional) The object of the action, usually a filename. You may pass NULL as an argument.
data	(Optional) A string. It specifies additional data needed to perform the ACTION. This chapter contains a description of data instances for possible ACTIONS.

For example, at the end of a test run, you may wish to view your AUT's logfile in your preferred SoftBench editor. DATA is not used.

The command you program in a TSL script might be:

```
send_message ("R EDIT WINDOW /tmp/aut.log");
```

Here, DATA has not been specified.

For more information on the parameters used in the **send_message** function, please refer to the *HP SoftBench Programmer's Reference*.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Mercury Interactive Corporation
1325 Borregas Avenue
Sunnyvale, CA 94089 USA

Main Telephone: (408) 822-5200
Sales & Information: (800) TEST-911
Customer Support: (408) 822-5400
Fax: (408) 822-5300

Home Page: www.merc-int.com
Customer Support: web.merc-int.com