Synlib: The Core of CDE Tests

Synlib is an application program interface for creating tests for graphical user interface applications. A collection of Synlib programs, each designed to verify a specific property of the target software, forms a test suite for the application. Synlib tests can be completely platform independent—an advantage for testing the Common Desktop Environment (CDE), which runs on the platforms of the four participating companies.

by Sankar L. Chakrabarti

Synlib is a C-language application program interface (API) designed to enable the user to create tests for graphical user interface (GUI) software. The user simply tells Synlib what to do and it will execute the user-specified tests on a variety of displays and in a variety of execution environments.

A complete description of the Synlib API is beyond the scope of this article and can be found in references 1 through 4. In this article we will only indicate how the ideas and capabilities supported by Synlib were applied to the development of tests for the Common Desktop Environment (CDE) described in ***Article 1***. An overview of CDE test technology including a description of the role played by Synlib can be found in ***Article 7***.

To test GUI software, we create programs using the Synlib API. These programs in turn manipulate and verify the appearance and behavior of the GUI application of interest (the target application). The recommended approach is to create many small Synlib programs to test the target software:

1. View the target GUI software as a collection of implemented properties.
2. Create a Synlib-based program to verify each property.
3. A collection of such Synlib programs forms a test suite for the GUI software.

With Synlib, the main task of testing GUI software reduces to creating the individual test programs. Considering each test program to be your agent, your task is to tell the agent what to do to verify a specific property of the program you wish to test. Assume that you want to test the following property of the front panel on the CDE display (Fig. 1): *On clicking the left mouse button on the style manager icon on the CDE front panel, the style manager application will be launched*. It is likely that you will tell your agent the following to verify if the property is valid for a given implementation of the front panel:

1. Make sure that the front panel window is displayed on the display.
2. Click the style manager icon on the front panel window. Alternatively, you might ask the agent to select the style manager icon on the front panel and leave it to the agent to decide how to select the icon. Synlib supports both alternatives.
3. Make sure that a new style manager window is now displayed. If the style manager window is mapped then the agent should report that the test passed. Otherwise, the agent should report that the test failed.

Fig. 1. *Front panel of the CDE desktop.*

These instructions are captured in the following Synlib program. All function calls with the prefix Syn are part of the Synlib API.

```
main(argc, argv)
int argc;
char **argv;
{
 Display *dpy;
 int windowCount;
 Window *windowList;
 Window shell_window;
 char *title;
 SynStatus result;

 dpy = SynOpenDisplay(NULL);
 result = SynNameWindowByTitle (dpy,
  ”MyFrontPanel”, ”One”, PARTIAL_MATCH,
  &windowCount, &windowList);
if (result == SYN_SUCCESS)
 {
  result = SynClickButton(dpy, ”Button1”,
   ”MyFrontPanel”, ”styleManager_icon”);
  /*
   * In the alternate implementation using focus     *  maps we could simply say:
   * result = SynSelectItem (dpy,
   *  ”MyFrontPanel”,
   *  ”StyleManager_Icon_FocusPath”);
   */
  result = SynWaitWindowMap (dpy,
    ”StyleManager”,TIME_OUT);
  if (result == SYN_SUCCESS)
   { result = SynGetShellAndTitle (dpy,
     ”Style Manager”, &shell_window, &title);
      if (strcmp (title, ”Style Manager”) ==           0)
       printf (”Test Passed: Style Manager
       window appeared. \n”);
    else
       printf (”Test Failed: Expected Style
        Manager window; found %s\n”,title);
   }
   else
    printf (”Test Failed: Expected Style Manager
      window would map but none did.\n”);
 }
 else
  printf (”Test Aborted: Front Panel window         could not be found. \n”);
}
```

SynOpenDisplay() connects to the display on which the target application is running or will run. SynNameWindowByTitle() determines if a window of the specified title (in this case One) is already displayed. If so, the user (i.e., the programmer) chooses to name it MyFrontPanel. Through SynClickButton(), the Synlib agent is instructed to click on the window MyFrontPanel at a location called styleManager_icon. The agent is being asked to expect and wait for a window to map on the display. The function SynWaitWindowMap accomplishes the wait and names the window StyleManager if and when a window maps on the display. If the mapped window has the expected title, StyleManager, then the agent is to conclude that the test succeeded, that is, the front panel window had the specified property (clicking on the style manager icon really launched the style manager application).

In practice, the tests would be more complicated than this example. Nonetheless, this simple program illustrates the basic paradigm for creating test agents. You need to name the GUI objects of interest and provide a mechanism for the agent to identify these objects during execution. The agent should be able to manipulate the named objects by delivering keystrokes or button inputs. Finally, the agent should be able to verify if specified things happened as a result of processing the delivered inputs. Synlib is designed to provide all of these capabilities. Table I is a summary of Synlib's capabilities.

Platform Independence

Synlib programs can be written so that they are completely platform independent. For example, in the program above there is nothing that is tied to specific features of a platform or a display on which the target GUI application may be executing. All platform dependent information can be (and is encouraged to be) abstracted away through the mechanism of *soft coding*. In the program above, the statement using the function SynClickButton is an example of soft coding. The last parameter in this statement, styleManager_icon, refers to a location in the front panel window. The exact definition of the location—the window's x,y location with respect to the FrontPanel window—is not present in the program but is declared in a file called the *object map*. At execution time the object map is made available to the Synlib agent through a command line option. The agent consults the object map to decode the exact location of the named object styleManager_ icon, then drives the mouse to the decoded location and presses the button. Because the location is soft coded, the program itself remains unchanged even if the front panel configuration changes or if the exact location of the named object is different on a different display. The named location, styleManager_icon, represents a semantic entity whose meaning is independent of the platform or the display or the revision of the target application. The semantics of the name is meaningful only in the test. In other words, the test remains portable. If changes in the platform, display, or application require that the exact location of the named object be changed, this is achieved either by editing the object map file or by supplying a different object map file specific for the platform. Synlib provides automated methods to edit or generate environment-specific object map files. The agent itself does not need any change.

Table 1
Synlib Capabilities

Functions to Name GUI Objects of Interest
SynNameWindow
SynNameWindowByTitle
SynNameLocation
SynNameRegion
.
.
.
Functions to Deliver Inputs to Named Objects
SynClickButton
SynClickKey
SynPressAndHoldButton
SynReleaseButton
SynMovePointer
SynPrintString
SynPressAndHoldKey
SynReleaseKey
SynSetFocus
SynSelectItem
Functions to Synchronize Application State with Test Agent
SynWaitWindowMap
SynWaitWindowUnmap
SynWaitWindowConfigure
SynWaitProperty
.
.
.
Functions to Verify the State of a GUI Application
SynGetShellAndTitle
SynStoreText
SynCompareWindowImage
.
.
.
Miscellaneous Functions to Process Needed Test Resources from the Environment
SynParseCommandOptions
SynParseObjectFile
SynBuildFocusMap
SynParseKeyMap

The format of a typical Synlib object map is:

```
! Object Map for the sample program
! Declares the locations named in the test
!   program
Location styleManager_icon 781 51

! Declares the full path of an item named in a   !   focus map
FocusPath StyleManager_Icon_FocusPath
  FrontPanel.ActionIcons.dtstyleIcon
```

Focus Maps

A far superior method of naming GUI objects of interest is to use Synlib's concepts of *focus maps* and *focus paths*. A focus map is a description of the logical organization of the input enabled objects in a widget-based application. An input enabled object is a region in a window that can accept keystrokes or button inputs from a user. Generally these objects are widgets or gadgets used in constructing the user interface.

The method of constructing a focus map is fully described in the Synlib User's Guide.[1] A more complete description of the concept of a focus map and its use in testing X windows applications has been published elsewhere.[2] A focus path is a string of dot-separated names declared in a focus map. For example, StyleManager_Icon_FocusPath is the name of the focus path FrontPanel.ActionIcons.dtstyleIcon, which is a string of dot-separated names declared in the focus map named FrontPanel. Focus maps are described in focus map files. Focus paths, on the other hand, are declared in object map files because, when associated with a window, a focus path identfies an actual object capable of accepting input.

In the example program above, the function SynSelectItem() represents an instruction to the agent to select the object named by the string StyleManager_Icon_FocusPath, which can be declared in the object map file as shown above.

The following is the focus map for the front panel window.

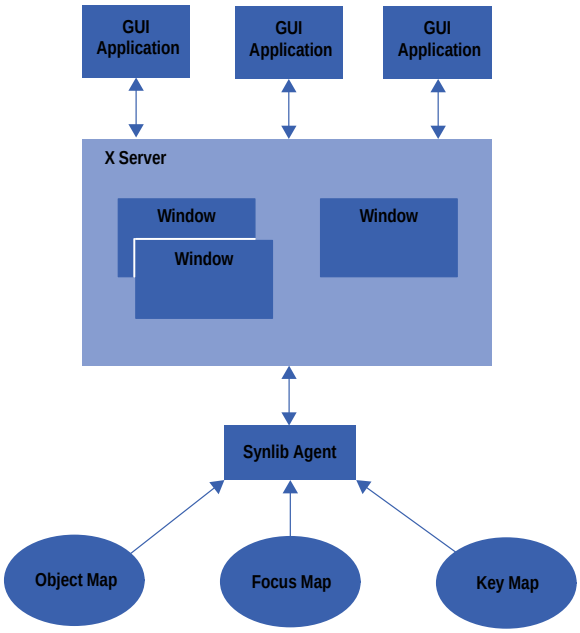
```
!
! Focus map for the default front panel window
!   of the CDE desktop.
!
(FocusMap FrontPanel
 (FocusGroup ActionIcons
    (App_Panel dtpadIcon dtmailIcon dtlockIcon
    dtbeepIcon workspace_One workspace_Three
    workspace_Two workspace_Four exitIcon
    printer_Panel printerIcon dtstyleIcon
    toolboxIcon Help_Panel helpIcon trashIcon
    dtcmIcon dtfileIcon)))
!
```

If the proper focus map and object maps are provided, the agent will apply Synlib embedded rules to decide how to set focus on the named item and then select or activate the item. During execution, Synlib first processes all supplied focus maps and creates an internal representation. Whenever the program refers to a focus path, Synlib decodes the identity of the desired object by analyzing the focus map in which the focus path occurs. Using the declarations in the focus map and applying OSF/Motif supported keyboard traversal specifications, Synlib generates a series of keystrokes to set the keyboard focus to the object indirectly named via the focus path. The rules for transforming the focus path name to the sequence of keystrokes are somewhat complex and have been fully described elsewhere.[2] These rules are embedded in Synlib and are completely transparent to the user.

This example shows the use of a focus map in naming icons in the front panel. Although the example here deals with a simple situation, the same principles and methods can with equal ease be used to name and access objects in deeply embedded structures like menus and submenus. In general, naming objects by means of a focus map is far superior to naming them by means of an object map. Because access to the objects of interest is via a dynamically generated sequence of keystrokes, the programs employing these methods are resistant to changes in window size, fonts, or actual object locations. This makes the tests completely portable across platforms, displays, and other environmental variabilities. Synlib programs using focus maps to name GUI objects need not be changed at all unless the specification of the target application changes.

Using a similar soft coding technique, Synlib makes it possible to create *locale neutral* tests, that is, tests that can verify the behavior of target applications executing in different language environments without undergoing any change themselves. Use of this technique has substantially reduced the cost of testing internationalized GUI applications. A complete description of the concept of locale neutral tests has been published.[4]

Fig. 2. *Synlib test execution architecture.*



Test Execution Architecture

Synlib provides concepts and tools that enable us to create "one test for one application." The tests, assisted by the required environment dependent resource files like object map, focus map, and key map files, can verify the behavior of target applications executing on different platforms, using different displays, and working in very different language environments.

Fig. 2 shows an execution architecture for Synlib tests. A key map file contains declarations to name keystrokes, button events, and sequences of keystrokes and button events. The key map file provides a way to virtualize and name all inputs to be used by a test program. This mechanism is very useful for creating tests for internationalized applications and is fully described in reference 4.

The cost of creating or modifying the environment resource files is minuscule compared to the cost of creating the tests themselves. Thus, the ability to create tests that are insensitive to differences in the execution environment of the target application has been a great productivity boost to our testing efforts.

A feature of Synlib test technology is that it does not require any change in the target application. It does not require that the application code be modified in any way. There is no need to incorporate any test hook in the application, nor is the application required to relink to any foreign test-specific library. Synlib supports a completely noninvasive testing framework. The test is directly executed on the application off the shelf. Synlib even makes it possible to write the tests before the application is ready for testing.[3]

The author originally designed Synlib to solve the problems of GUI testing facing our lab, mainly testing GUI applications that supported a variety of HP displays and operating systems. We designed Synlib to provide a technology that yields robust and platform-insensitive tests at a low cost. Synlib proved to be a marvelous fit for testing the CDE desktop since one of the main conditions was that the tests would have to verify applications running on the platforms of the four participating companies. Essentially it was a problem of creating platform-insensitive tests, a problem that we had already solved. The success of Synlib in this endeavour is shown by the large body of functioning test suites for the complex applications of the CDE desktop.

Acknowledgments

The development of Synlib has benefited from the comments, criticism, and support of many people. The author wishes to thank everyone who willingly came forward to help mature this technology. Harry Phinney, Fred Taft, and Bill Yoder were the first to create test suites for their products using Synlib. Their work proved the value of Synlib to the rest of the laboratory. Subsequently, Bob Miller allowed his group to experiment with Synlib, which led to its adoption as the testing tool for CDE. Thanks Bob, Harry, Fred, and Bill. Julie Skeen and Art Barstow volunteered their time to review the initial design of Synlib. A very special thanks is due Julie. In many ways the pleasant and intuitive user interface of Synlib can be traced to her suggestions. Thanks are also due the engineers at the multimedia lab who proved the effectiveness of Synlib in testing multimedia applications. Ione Crandell empowered this effort. Kristann Orton wrote many of the man pages. Dennis Harms and Paul Ritter effectively supported Synlib in many stormy CDE

sessions. Thanks Dennis, Kritann, and Paul. Michael Wilson taught me how Synlib could solve the knotty problem of testing hyperlinked systems. Thanks, Mike. Claudia DeBlau and Kimberly Baker, both of Sun Microsystems, helped in developing Synlib's interface to the Test Environment Toolkit (TET). Finally the author thanks Ken Bronstein. Ken appreciated the value of this unofficial job from the very beginning. Ken's unwavering support has been crucial to the continued development of this technology.

References

1. S.L. Chakrabarti, *Synlib User's Guide—An Experiment in Creating GUI Test Programs*, Hewlett-Packard Company.
2. S.L. Chakrabarti,"Testing X Clients Using Synlib and Focus Maps," *The X Resource*, Issue 13, Winter 1995.
3. S.L. Chakrabarti, R. Pandey, and S. Mohammed,"Writing GUI Specifications in C," *Proceedings of the International Software Quality Conference*, 1995.
4. S.L. Chakrabarti and S. Girdhar,"Testing 'Internationalized' GUI Applications," accepted for publication, *The X Resource*, Winter 1996.

▶ Go to Article 9
▶ Go to Table of Contents
▶ Go to HP Journal Home Page