Automatically Testing "Nightly/daily Builds" of GUI Applications

Atif Memon*, Ishan Banerjee, Adithya Nagarajan
Department of Computer Science, University of Maryland College Park, Maryland, USA
(* also with Fraunhofer Center for Experimental Software Engineering, Maryland)
{atif, ishan, sadithya}@cs.umd.edu

Abstract

We describe a technique that addresses the needs of "Nightly/daily building and smoke testing" of software that has a graphical user interface (GUI). The key to our success is automation of structural GUI analysis, test case generation, test oracle creation, code instrumentation, test execution, coverage evaluation, regeneration of test cases, and their re-execution. We empirically evaluate the time taken and memory required for GUI analysis, test case and test oracle generation, and test re-execution.

Introduction: "Nightly/Daily building and smoke testing" have become popular as many software developers find them useful. During nightly builds, software is compiled, linked and smoke tested. Smoke tests exercise the entire system. They need not be an exhaustive test suite; the goal is to detect major problems.

A limitation of current smoke tests is inadequate testing of software that has a *graphical user interface* (GUI). Although there has been considerable success in developing techniques for efficient re-testing of conventional software, the area of frequent GUI re-testing has been neglected.

Not being able to re-test a GUI efficiently has negative impact on overall software quality because GUIs have become ubiquitous as a means of interacting with software systems. Currently to perform re-testing, a developer either uses test harnesses that call methods of the underlying logic as if initiated by a GUI, which requires major changes to the software architecture and does not test the "end user" software, or testers perform very limited smoke testing of the GUI using manual tools.

In this paper, we present a new technique for automated smoke testing of GUI software with minimal tester interaction. Our technique automates GUI structure analysis, test case generation, oracle creation, code instrumentation and execution, coverage evaluation, regeneration and re-execution of test cases. Our technique also automatically identifies smoke tests for the GUI. We present preliminary results of experiments demonstrating its feasibility for daily re-testing of GUIs.

High-level Design: Figure 1 shows the primary tools employed by our technique. All tools interact with each other via a common GUI representation. We first describe the representation and then briefly describe each tool.

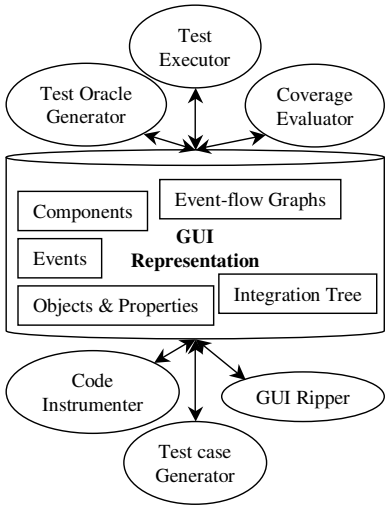

Figure 1. Tools Used for Re-testing.

The *GUI representation* is a formal model of the software's GUI. Note that the entire representation is extracted automatically from the implemented GUI using a *GUI Ripper*. The GUI is modeled as a set of *objects* $O = \{o_1, o_2, \ldots, o_m\}$ (e.g., `label`, `form`, `button`, `text`) and a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (e.g., `background-color`, `font`, `caption`). Each GUI will use certain types of objects with associated properties; at any specific point in time, the GUI can be described in terms of all the objects that it contains, and the values of all their properties.

A *Test Case Generator* uses the representation to automatically generate smoke tests. The smoke tests are determined by a *Coverage Evaluator* that specifies smoke testing requirements using several coverage criteria: (1) *Event Cov-*

erage, which requires that individual events in the GUI be exercised, (2) *Event-Interaction Coverage*, which requires all events in a given path be exercised, and (3) *Length-n Event-sequence Coverage*, which requires event sequences of a given length be exercised.

A *Test Oracle Generator* collects/extracts complete (or partial) state information of the GUI. This information is used for verfying the correctness of the GUI during re-testing. We can generate oracle information at different levels of decreasing cost and accuracy: (1) *Complete* in which all windows, invisible, visible, and their properties are collected, (2) *Complete Visible* in which all visible windows and their properties are collected, (3) *Active Window* in which the active window and its properties are collected, and (4) *Widget* in which only information about the widget in question is collected.

A *Test Executor* automatically replays the entire test suite. It takes as input the test suite to replay and the level of oracle to incorporate as a test verifier. The test designer has the ability to select the level of oracle to use and from that, the test executor executes the tests and outputs from the tests are automatically verified. The level of oracle information to incorporate has overheads as discussed in the experiment section. The test executor further categorizes the outputs as successful or unsuccessful and these results are then processed by the testers.

Experiments: We now present results of experiments to show that our re-testing technique is practical. We selected a set of programs developed in-house as our subject applications. Table 1 summarizes the characteristics of these programs.

In the experiment, we manipulated three independent variables: (1) **P**, i.e., the six subject programs, (2) **LOI**, the four levels of oracle information detail, i.e., complete, complete visible, active window, and widget, and (3) **LOT**, the four levels of testing. Note that for a given test run, LOI $\geqslant$ LOT, i.e., the information must be generated before it can be used.

On each run, with program P, levels LOI, levels LOT, we generated test information for 1000 test cases and measured the total generation time and memory required. We then executed all these test cases for each of the 10 possible LOI and LOT combinations. In all, for our complete experiment, we generated and executed $6 * 10 * 1000 = 60{,}000$ test cases.

Results: The levels of testing (LOT1–LOT4) correspond directly to the oracle information that was collected, i.e., complete, complete-visible, active window, and widget. Figure 3 shows the total execution times for TerpSpreadSheet for all possible combinations of LOI and LOT. We see that using all-windows, is, in general expensive compared to widgets. Figure 2 compares the memory requirement for all of our subject programs for all levels of oracle. LOI0 represents test cases with no oracle information. The memory requirements grow very rapidly when using detailed level of test oracle.

Subjects	Windows	LOC	Classes	Components
TerpPaint	8	9287	42	7
TerpSpreadsheet	6	9964	25	5
TerpPad	8	1747	9	5
TerpCalc	3	4356	9	3
TerpDraw	5	4769	4	3
TerpManager	1	1452	3	1
TOTAL	***31***	***31575***	***92***	***24***

Table 1. Subject Applications

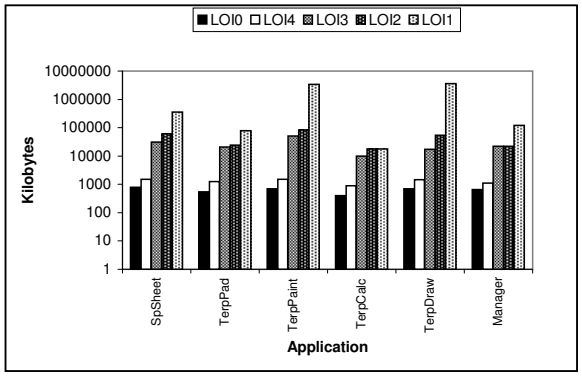

Figure 2. Storage Requirements. Y-axis is logarithmic

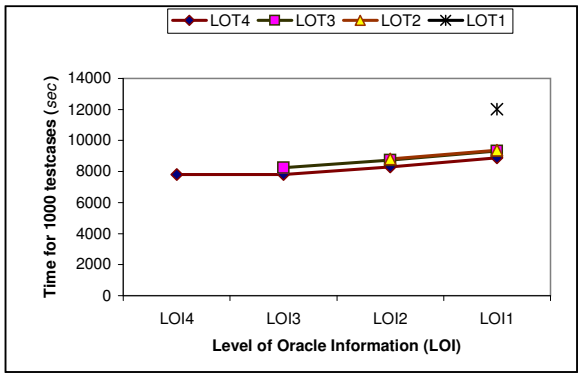

Figure 3. Time for TerpSpreadSheet

Conclusions and Future Work: We presented a technique for re-testing software that has a GUI. We empirically demonstrated that the technique is practical and may be used for smoke testing nightly/daily builds of GUI software. Our technique is not restricted to smoke testing of nightly builds only. In the future, we will extend our technique to generate and execute tests other than smoke tests.