Empirical Evaluation of the Fault-detection Effectiveness of Smoke Regression Test Cases for GUI-based Software

Atif M. Memon
Department of Computer Science
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland, USA
atif@cs.umd.edu

Qing Xie
Department of Computer Science
University of Maryland
College Park, Maryland, USA
qing@cs.umd.edu

Abstract

Daily builds and smoke regression tests have become popular quality assurance mechanisms to detect defects early during software development and maintenance. In previous work, we addressed a major weakness of current smoke regression testing techniques, i.e., their lack of ability to automatically (re)test graphical user interface (GUI) event interactions – we presented a GUI smoke regression testing process called Daily Automated Regression Tester (DART). We have deployed DART and have found several interesting characteristics of GUI smoke tests that we empirically demonstrate in this paper. We also combine smoke tests with different types of test oracles and present guidelines for practitioners to help them generate and execute the most effective combinations of test-case length and test oracle complexity. Our experimental subjects consist of four GUI-based applications. We generate 5000-8000 smoke tests (enough to be run in one night) for each application. Our results show that (1) short GUI smoke tests with certain test oracles are effective at detecting a large number of faults, (2) there are classes of faults that our smoke test cannot detect, (3) short smoke tests execute a large percentage of code, and (4) the entire smoke testing process is feasible to do in terms of execution time and storage space.

1 Introduction

Most of today's software applications are developed and maintained by multiple programmers, often geographically distributed, who work on parts of the overall application code. While leading to improved code churn rates, this practice also leads to problems, often affecting programmer productivity. For example, developers may not realize that they have inadvertently broken parts of the code. To help sustain programmer productivity, quality assurance mechanisms are integrated into the development and maintenance cycle. One such mechanism requires executing smoke regression tests together with nightly/daily building of the software. In recently reported work [24], we addressed an important weakness of current smoke testing techniques, i.e., their inability to automatically and efficiently smoke test the graphical user interface (GUI) front-end part of the software – we defined GUI smoke tests and described a process called *Daily Automated Regression Tester* (DART) that re-tests frequent builds of GUI software.

We have implemented and successfully deployed DART. The key to the success of DART is that developers can work on the code during the day; DART automatically launches the *application under test* (AUT) at night, builds it and runs GUI smoke test cases (sequences of GUI events). Coverage and bug reports are e-mailed to developers, who can quickly fix the bugs. DART automates everything required for GUI smoke testing including structural GUI analysis (which we refer to as *GUI ripping* [25]), test case generation [29, 27], test oracle creation [28], code instrumentation, test execution, coverage evaluation [30], regeneration of test cases, and their re-execution. Together with the operating system's task scheduler (e.g., MS Windows task scheduler, Unix cron job), DART can execute frequently with little input from the developer/tester to smoke test the GUI software.

Since the deployment of DART, we have identified important characteristics of GUI smoke tests that we describe and experimentally demonstrate in this paper. We also describe two important, previously ignored, aspects of automated GUI smoke testing, i.e., test oracles and code coverage. During the testing process, as test cases are executed on the AUT, *test oracles* are used to determine whether the AUT executed as expected [11]. In previous work, we developed several types of automated GUI test oracles [26, 28]. Our work showed that the overall effectiveness of

the testing process depends not only on the number and type of test cases used, but also on the type of test oracle used. We now develop 5 different oracles of varying complexity for GUI smoke test cases. Finally, we present guidelines for practitioners who perform smoke testing of GUI-based software to help them generate and execute the most effective combination of test-case length and test oracle complexity.

Our experimental subjects consist of four GUI-based applications. We create 200 versions of each application by artificially seeding them with 200 faults. We generate and execute 5000-8000 smoke test cases (enough to be run in one night) using the structure of each application. In all, we report results of more than $5000 \times 200 \times 4 = 4,000,000$ test runs. Our results show that (1) short GUI smoke tests with certain test oracles are effective at detecting a large number of faults, (2) there are classes of faults that our smoke test cannot detect, (3) short smoke tests execute a large percentage of code, and (4) the entire smoke testing process is feasible to do in terms of execution time and storage space.

The specific contributions of this work include:

- A first empirical study evaluating the strengths and weaknesses of GUI smoke tests.
- Five new test oracles for GUI smoke tests, their relative strengths and costs.
- Relationship between GUI smoke tests and coverage of the underlying code.
- Classification of faults that can and cannot be detected by GUI smoke tests.

In the next section, we present background and related work. In Section 3, we give a brief overview of the DART process, and in Section 4 an overview of all its parts relevant to the experiments described in Section 5. Finally, we conclude in Section 6 with a discussion of future work.

2 Background & Related Work

Nightly/daily builds and smoke tests [18, 21, 32] have become widespread [34, 15]. Software that use daily/nightly builds and smoke tests include *WINE* [9], *Mozilla* [6], *AceDB* [2], and *openwebmail* [7]. During nightly builds, a development version of the software is checked out from the source code repository tree, compiled, linked and "smoke tested" ("smoke tests" are also called "sniff tests" or "build verification suites" [19]). Typically *unit tests* [34] and sometimes *acceptance tests* [12] are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system [19]. Smoke tests exercise the entire system; they don't have to be an exhaustive test suite but they should be capable of detecting major problems. A build that passes the smoke test is considered to be "a good build". Bugs are reported, usually in the form of e-mails to the developers [34], who can quickly resolve the bugs. Frequent building and retesting is also gaining popularity because new software development processes (XP [14, 35]) advocate a tight development/testing cycle [33]. A number of tools support daily builds; some of the popular tools include *CruiseControl* [1], *IncrediBuild* [3], *Daily Build* [8], and *Visual Build* [5].

Four different approaches are used to handle GUI software when performing smoke testing. First, and most popular, is to perform no GUI smoke testing at all [19], which either leads to compromised software quality or expensive GUI testing later. Second is to use test harnesses that "bypass" the GUI and invoke methods of the underlying business logic as if initiated by a GUI. This approach not only requires major changes to the software architecture (e.g., keep the GUI software "light" and code all "important" decisions in the business logic [20]), it also does not perform testing of the end-user software. Third is to use existing tools to do limited GUI testing [13, 36]. Examples of some tools used for GUI testing include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module*[1] and capture/replay tools [17] such as WinRunner[2] that provide very little automation [23], especially for *creating* smoke tests. Developers/testers who employ these tools typically come up with a small number of smoke tests [21]. Finally, the most comprehensive and complete solution is provided by our own work [24] that addresses the needs of smoke testing of software applications that have a GUI. We give an overview of our approach next.

3 Overview of DART

As discussed in Section 1, the main design goal of DART is to automate GUI smoke testing. We have developed the DART process that realizes this automation. In this section we present the steps of the DART process. The goal is to provide the reader with a high-level picture of the operation of DART. These steps are also summarized in Table 1. Note that the names of the modules of DART are highlighted in bold-face and described later in Section 4.

1. The developer (or test designer) identifies the AUT.
2. DART analyzes the AUT's GUI structure (using the **GUI ripper**) by automatically traversing all the windows of the GUI and identifying all the GUI objects (widgets) and their properties. It then computes the total number of possible smoke test cases (event sequences).
3. The developer chooses a subset of these test cases by specifying the number of test cases of each length (i.e., the number of events in the sequence) to generate. We advocate running at least all test cases of length 1 and 2 for smoke testing. In our experiments (Section 5),

[1] http://junit.org/news/extension/gui/index.htm
[2] http://mercuryinteractive.com

Phase	Step	Developer/tester	DART
Identification	1	Identify AUT	
Analysis	2		Analyze AUT's GUI
Test Generation	3	Choose test cases	
	4		Generate test cases
	5		Generate expected output
Modification	6	Modify AUT	
Regression Testing	7		Instrument code
	8		Execute test cases and compare with expected output
	9		Generate execution report
	10		Generate coverage report
	11		E-mail reports
Analysis and Regeneration	12	Examine reports and fix bugs	
	13	Specify additional test cases	
	14		Generate additional test cases
	15		Generate additional expected output

Table 1. The DART Process

we generate 5000-8000 test cases since they can easily be executed in one night.

4. DART uses an automated **test case generator** to generate the smoke test cases.
5. A **test oracle generator** is used to automatically create an expected output for the next version of the AUT. The *smoke test suite* for subsequent versions is now ready.
6. The development team modifies the AUT.
7. The operating system's task scheduler launches DART, which in turn launches the AUT. DART automatically instruments the AUT's source code using a **code instrumenter** (e.g., Instr [4]).
8. Test cases are executed (using a **test case executor**) on the AUT automatically and the output is compared to the stored expected output (from Step 5).
9. An execution report is generated in which the executed test cases are classified as *successful* or *unsuccessful*.
10. The coverage and bug reports are generated.
11. These results are e-mailed to the developers.
12. The next morning, the developers examine the reports and fix the bugs. They also examine the unsuccessful test cases. Note that a test case may be unsuccessful because (1) it crashed the software, (2) the expected output did not match the actual output; if the expected output is found to be incorrect, then the test oracle generator is used to automatically update the expected output for the modified AUT, or (3) an event in the test case had been modified (e.g., deleted) preventing the test case from executing. These test cases can no longer be run on the GUI and are deleted.
13. Using the coverage reports, the developers identify new areas in the GUI that should be tested. They specify additional test cases to generate.
14. The new test cases, and
15. oracle information are generated.

    Steps 7 through 15 are repeated throughout the development cycle of the AUT.

The developer may also include code-based smoke test cases in the above cycle to improve overall test effectiveness.

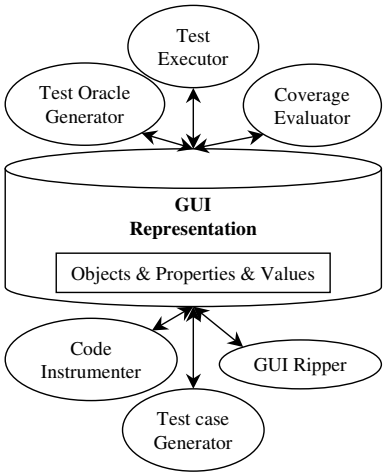


Figure 1. Modules of DART

4 Modules of DART

We now briefly describe each module mentioned in the previous section. Note that due to lack of space, we provide, for each module, only the details needed to understand the experiments and interpret the results. Additional details and algorithms are available in [24, 22, 28, 30, 29].

Representation: Figure 1 shows the primary modules of DART and their interaction. All modules interact with each other via the GUI representation. We represent a GUI as a set of *objects* $O = \{o_1, o_2, \ldots, o_m\}$ (e.g., widgets, windows, frames) and a set of *properties* $P = \{p_1, p_2, \ldots, p_l\}$ of those objects (e.g., `background-color`, `font`, `caption`). Each GUI uses certain types of objects with associated properties; at any specific point in time, we represent the state of the GUI in terms of all the objects that it contains, and the values of all their properties.

GUI ripper: "GUI Ripping" is a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets

(GUI objects), properties, and values. The extracted information is then verified by the test designer.

Test Case Generator: Users interact with the GUI by performing *events* $\{e_1, e_2, \ldots, e_n\}$ on some widgets, such as clicking on a button, opening a menu, and dragging a file. During GUI testing, test cases, consisting of sequences of events are executed on the GUI.

The output of the GUI ripper is used to automatically generate *event-flow graphs* (EFG), which are used as the basis for generating test cases. Intuitively, an EFG contains nodes that represent events and edges. An edge from node n_1 to n_2 means that the event represented by n_2 can be performed immediately after the event represented by node n_1. To generate test cases, we start from a known initial state of the GUI and use a graph traversal algorithm, enumerating the nodes during the traversal, on the EFGs. If the event requires text input, e.g., for a text-box, then its value is read from a database, initialized by the software developer. Sequences of events $e_1; e_2; \ldots; e_n$ are generated as output that serve as a GUI test case.

Definition: A *test case* for a GUI is $e_1; e_2; e_3; ...; e_n$ where e_{i+1} can be performed *immediately* after e_i. □

We then classify event sequences by length. Length 1 event sequences consist of all events in the software's EFGs. Length 2 event sequences are all possible subsequences of the form $< e_i, e_j >$, where there is an edge from e_i to e_j in an EFG. For smoke testing, we require that all length 1 and 2 event sequences be executed although the final choice of smoke tests lies with the developer. Note that all test cases of length 1 and 2 execute *all* GUI events and all pairs of events. Some events in the GUI may not be executable using length 1 and 2 event sequences because of the structure of the GUI, e.g., they may be in a window that needs several events to be executed before it can be opened. In such cases, we generate a *prefix* sequence of events to launch the window. The final test case is obtained by concatenating the prefix with the event sequence of interest. We will ignore the length of the prefix in the rest of the paper. In our experiments, we will also ignore the code coverage and faults detected, if any, by the events in the prefix.

Test Executor: The test executor is capable of executing an entire test suite automatically on the AUT. It performs all the events in each test case and invokes the test oracle to compare the actual output with the expected output. Events are triggered on the AUT using the native OS API. For example, the windows API *SendMessage* is used for windows applications and Java API *doClick* for Java applications.

Test Oracle Generator: As mentioned earlier, test oracles are used to determine whether or not the AUT executed correctly during testing. In GUIs, we design test oracles that compare the expected output (called *oracle information*) including objects, properties, and values to the actual GUI output. Our model of the GUI in terms of objects/properties is used to represent the oracle information of the GUI after the execution of an event.

For any test case $< e_1; e_2; \ldots e_n >$, the oracle information is represented by a sequence of states $S_1; S_2; \ldots S_n$ that capture the complete (or partial) state of the GUI after each event. Depending on the resources available, DART can collect and compare oracle information using the following parts of the GUI.[3]

Complete: all the properties of all the objects of all the windows in the GUI.

Complete visible: all the properties of all the objects of all the *visible* windows in the GUI. Note that by the term "visible windows", we mean all windows for which the `isVisible` property is TRUE. Windows that are "hidden" behind other overlapping windows are also considered visible, if their `isVisible` property is TRUE.

Active window: all the properties of all the objects of the *active* window in the GUI.

Widget: all the properties of the object (in the current window) on which the current event is being performed.

Later, in the experiments (Section 5), we show that having more details in the oracle information improves the overall test effectiveness of a smoke test suite.

The comparison of the expected and actual states can be done as frequently as once after each event of the test case or less frequently, e.g., after the last event. We expect that reducing the frequency of comparison will reduce the test case execution time as well as space required to store the oracle information (since only the relevant state(s) of the GUI need(s) to be stored). In Section 5, we show that the fault-detection effectiveness of the test suite is *not* impacted by changing the frequency of comparison.

Using the combination of oracle information and frequency of comparison, we define 5 types of oracles for GUI smoke testing. Oracles L1, L2, L3, and L4 represent comparing the *widget*, *active window*, *complete visible*, and *complete* oracle information *after each event* of the test case with the actual GUI respectively. Oracle L5 represents comparing the complete oracle information *after the last event* of the test case with the actual GUI.

Coverage Evaluator and code instrumenter: Although smoke tests are not meant to be exhaustive, we feel that coverage evaluation serves as a useful guide to additional testing, whether it is done for the next build or for future comprehensive testing. In DART, we evaluate conventional code coverage in terms of statements, branches, methods, classes, packages, and files. To collect the coverage information, we use source-level code instrumenters.

[3]The need for these levels is explained in detail in earlier reported work [28, 26].

5 Experiments

Having described the DART process and the modules that make up DART, we now present details of experiments using actual software subjects and smoke test cases to study important characteristics of GUI smoke test cases. We are interested in answering the following questions:

1. What is the fault detection ability of GUI smoke tests?
2. Are GUI smoke tests especially suited to detect certain classes of faults? Are there classes of faults that cannot be detected by smoke test cases?
3. Does the level of detail in the oracle information have any impact on the fault detection effectiveness of a test case?
4. During test case execution, is there any benefit to frequent comparison of expected and actual states?
5. What is the relationship between smoke test cases and the coverage of the underlying code? Since smoke test cases are short (1-3 events), are there large parts of the code that remain unexecuted?

5.1 Experimental Process

To answer the questions we follow the following steps:

1. Choose software subjects with GUI front-ends,
2. Generate smoke test cases and associated oracle information,
3. Use fault seeding techniques to artificially seed faults in the software subjects,
4. Execute all test cases on the software subjects. During execution, compare the actual GUI state to the oracle information,
5. Measure the variables: (1) *Number of Faults Detected*: We record the total number of faults detected by test case, the level of detail that was needed to detect the fault, and the position in the test case when the fault was detected. (2) *Code Coverage*: For each test case, we record statement, branch, method, class, package, and file coverage.
6. Each test case and oracle has different time and space requirements, primarily because of the level of detail of the oracle information and frequency of comparison. We measure the space required to store different levels of oracle information and time to compare.

Subject Applications: The subject applications for our experiments are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[4] and consists of six applications out of which we use four – TerpWord (a small word-processor), TerpSpreadSheet (a spreadsheet application), TerpPaint (an image editing/manipulation program), and TerpCalc (a scientific calculator with graphing capability). They have been implemented using Java. Table 2 summarizes the characteristics of these applications. Note that these applications are fairly large with complex GUIs. The number of widgets listed in the table are the ones on which user events can be executed (e.g., text-labels are not included).

[4] www.cs.umd.edu/users/atif/TerpOffice

Subject Application	Windows	Widgets	LOC	Classes	Methods	Branches
TerpWord	11	132	4893	104	236	452
TerpSpreadSheet	9	165	12791	125	579	1521
TerpPaint	10	220	18376	219	644	1277
TerpCalc	1	92	9916	141	446	1306
TOTAL	31	609	45976	589	1905	4556

Table 2. TerpOffice Applications

	Potential Test Cases			Actual Generated Test Cases		
	Length			Length		
Subject Application	1	2	3	1	2	3
TerpWord	126	1140	12461	126	1140	3880
TerpSpreadSheet	162	2742	56076	126	2742	2318
TerpPaint	215	8077	502133	215	8077	0
TerpCalc	87	7366	623702	87	7366	0
TOTAL	590	19325	1194372	590	19325	6198

Table 3. Number of Smoke Tests Generated

Test Cases: We used DART to automatically generate 5000-8000 smoke test cases for each application. The exact number of test cases for each application is shown in Table 3. The table shows, for each application, the total number of test cases that *could have been generated* and the number of test cases that were *actually generated*. We chose not to generate all length 3 test cases for some applications since we would not be able to run them on one machine in one night, thus defeating the purpose of smoke testing. The total number of test cases for all four applications was 590, 19325, 6198 of lengths 1, 2, and 3 respectively. In practice, if multiple machines are available, the test cases could be distributed. Another way to reduce time is to partition the test suite and run one partition each night.

Oracle Information: We used an automated tool (details beyond the scope of this paper) to generate the oracle information. The key idea of the technique employed by this tool is that it automatically executes a given test case on a software and captures its state (widgets, properties, and values) automatically. By running this tool on the four subject applications for all test cases, we obtained the oracle information. Note that the tool extracted all four levels of oracle information.

Fault Seeding: Fault seeding is a well-known technique used to evaluate fault detection techniques. During fault seeding, known faults are artificially introduced into

Reported Fault in Bug Database	Corrected Code
if (contentArea.getSelectedText() != null)	if (contentArea.getSelectedText() == null)
(a)	(b)

Fault #1	Fault #2
if (orientation != SwingConstants.HORIZONTAL)	if(!printJob.printDialog())
(c)	(d)

Table 4. Seeding GUI Faults

the subject programs. Care is taken so that the artificially seeded faults are similar to faults that naturally occur in real programs due to mistakes made by developers [31, 16].

We define a *GUI fault* as one that manifests itself on the visible GUI at some point of time during the software's execution. We seeded 200 faults in the TerpOffice applications to create 200 faulty versions for each application. We adopted an observation-based approach to seed the faults, i.e., *we observed "real" GUI faults in TerpOffice*. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[5] was used by the developers to report and track faults in TerpOffice version 1.0 while they were working to extend its functionality and developing version 2.0. The reported faults are an excellent representative of faults that are introduced by developers during implementation. Table 4(a) shows an example of a fault reported in our Bugzilla database and Table 4(b) shows the (later) corrected segment of the same code. Table 4(c) and 4(d) show examples of faults seeded into this code.

We seeded exactly one fault in each version. This model is useful to avoid fault-interaction, which can be a thorny problem in these types of experiments and also simplifies the computation of the variable "Number of Faults Detected"; now we can simply count the faulty versions that led to a mismatch between the executing GUI state and the oracle information.

Test Executor: We executed all the smoke tests on all 200 versions of the subject applications. When each application was being executed, we extracted its run-time state and compared it with the stored oracle information. A mismatch was reported as a fault. Note that we ignored widget positions during this process since the windowing system launches the software in a different screen location each time it is invoked. Table 5 shows the total number of objects (O) and their properties (P) that were checked by all the smoke test cases. Note that the oracle L1 examined only 100K GUI widgets during the execution. L4, on the other hand, examined more than 100 times that number. As can be imagined, L4 takes longer to execute and requires more space to store the expected output.

Each test case required approximately 5 seconds to execute. The time varied by application and the number of GUI events in the test case. The total execution time for each application is shown in Table 6. The execution included launching the application under test, replaying GUI events from a test case on it and analyzing the resulting GUI states. The analysis consisted of recording the actual GUI states of the faulty version and determining the result of the test case execution. The test cases executed on four machines (Pentium 4, 2.2GHz, each with 256MB RAM) simultaneously for almost a week. Although much of the execution was automated, we had to restart some machines (and test scripts) because of problems with the JVM.

[5] bugzilla.org

Oracle	L1		L2		L3		L4		L5	
Subject Application	O	P	O	P	O	P	O	P	O	P
TerpWord	28K	333K	1016K	12192K	1824K	21889K	1843K	22121K	370K	4437K
TerpSpreadSheet	21K	253K	1115K	13385K	2211K	26532K	2604K	31243K	626K	7507K
TerpPaint	27K	322K	2801K	33607K	3243K	38921K	3271K	39254K	949K	11382K
TerpcCalc	24K	288K	2361K	28337K	2365K	28385K	3146K	37756K	951K	11417K
TOTAL	100K	1196K	7293K	87521K	9644K	115726K	10865K	130375K	2895K	34743K

Table 5. Number of Objects and Properties

Subject Application	Total Execution Time (sec)
TerpWord	416047
TerpSheet	309411
TerpPaint	129756
TerpCalc	120200
TOTAL	975414

Table 6. Total Execution Time

5.2 Threats to Validity

Threats to external validity are conditions that limit the ability to generalize the results of our experiments to industrial practice. We have used four GUI-based Java applications as our subject programs. Although they have different types of GUIs, this does not reflect a wide spectrum of possible GUIs that are available today. We note that all our applications are extemely GUI-intensive, i.e., most of the code is written for the GUI. The results will be different for applications that have a complex underlying business logic and a fairly simple GUI. Moreover, all our subject programs were developed in Java. Although our abstraction of the GUI maintains uniformity between Java and Win32 applications, the results may vary for Win32 applications.

Threats to internal validity are conditions that can affect the dependent variables of the experiment without the researcher's knowledge. We have used an observation-based approach for seeding faults in the GUI applications. This may have affected the detection of faults by the test cases. Faults not exercised by any test case will go undetected. We made an effort to make the faults as close as possible to naturally occurring faults. Some of these faults might not manifest themselves through the GUI.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are

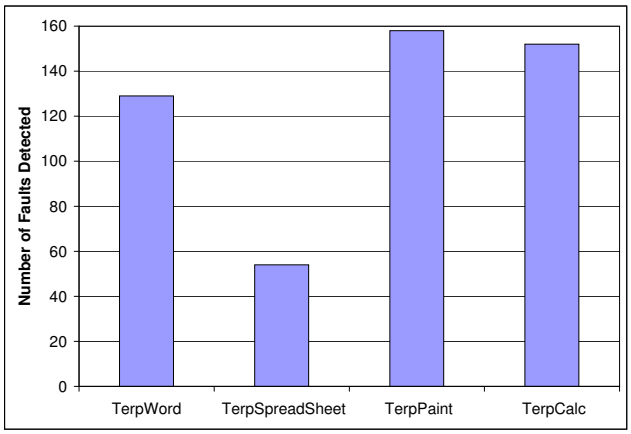


Figure 2. Number of Faults Detected

supposed to measure. For example, in this experiment one of our measures of cost is time. Since GUI programs are often multi-threaded, and interact with the windowing system's manager, our experience has shown that the execution time varies from one run to another. One way to minimize the effect of such variations is to run the experiments multiple number of times and report average time.

The results of our experiments, presented next, should be interpreted keeping in mind the above threats to validity.

5.3 Results

Total faults detected. *Conclusion: Smoke tests are able to detect more than 60% of the faults for most applications*. The column graph in Figure 2 summarizes our results. The x-axis shows the subject applications and the y-axis shows the total number of faults detected. The figure shows that, with the exception of TerpSpreadSheet, smoke tests detected a large number of faults.

We visually examined the execution of the smoke tests on TerpSpreadSheet and found that the weakness was in our test oracle, not the smoke test cases. The smoke test cases executed the lines in which the faults were seeded and the faults manifested themselves on the screen. However, our test oracle did not examine the attributes (e.g., contents, style, font) of the individual cells of the spreadsheet, thereby leading to missed faults. In the future, we will develop new, specialized test oracles for TerpSpreadSheet to detect these missed faults. These test oracles will be domain dependent (i.e., for TerpSpreadSheet); we will develop mechanisms to incorporate them into DART.

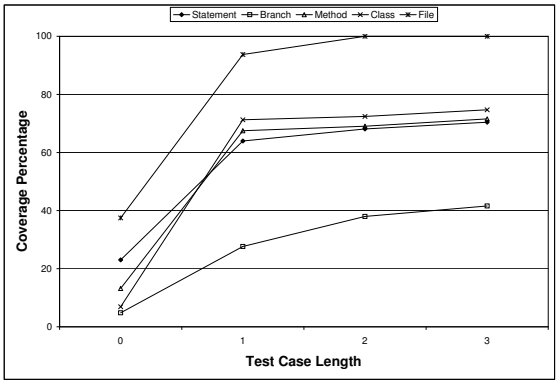
There were other faults that could not be detected because they were seeded in dead code. Some other faults were seeded in code that was part of exception handlers. Our current smoke tests do not specifically target execptions, although we are extending the test cases to use *fault injection* [10] techniques that would lead to exceptions.

Smoke tests and code coverage. *Conclusion: Short smoke tests execute a large percentage of code.* In order to better understand our above results, we examined the code coverage of the smoke tests. The results are shown in Figure 3. The figure shows four line graphs, one for each application. There are 5 lines in each graph for statement, branch, method, class, and file coverage. The x-axis shows the length of the test case and the y-axis shows the percentage coverage. The results show that our smoke tests were able to execute more than 60% of the statements, 40% of branches, 60-75% methods, and 75% classes. Note that some percentage of code is executed even though no events (test case length 0) are performed on the GUI.

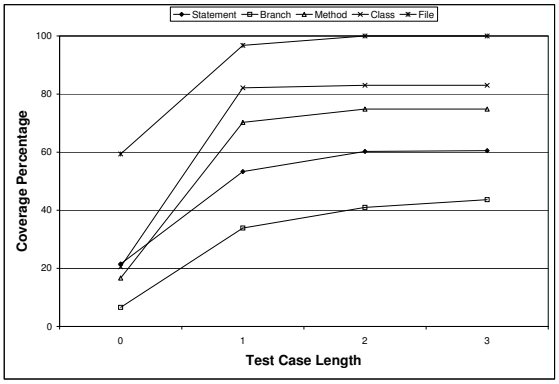
We manually examined the code to find reasons why some parts were not covered. These reasons also helped us to identify classes of faults and their locations that we could not detect. First, we found that that we were unable to execute code related to widgets (e.g., the close button in all windows) that were not ripped by GUI Ripper. We are currently extending the ripper to handle these widgets. Second, as mentioned earlier, we did not generate exceptions, which accounted for a large percentage of missed code. We are currently studying how to generate exceptions using our smoke test cases. Third, since our test cases are replayed using an API that directly communicates with the application, mouse and keyboard events are not generated during replay. Event handlers (e.g., right-click event handler) for such events are not executed. Fourth, since we run our test cases in a controlled environment, i.e., we reset the environment variables (e.g., list of recently accessed files) before executing each test case. Code related to these variables is never executed. Finally, there are events in the GUI that are enabled only after some other event sequence has been executed. If the required event sequence is longer than 3 events, our smoke test cases cannot execute code associated with the disabled events.

Faults detected and test case length. *Conclusion: Longer event sequences are able to detect more faults than shorter ones.* We noted from the code coverage analysis that length 2 and 3 event sequences don't substantially add to the code coverage. We wanted to see whether the length has any impact on fault-detection effectiveness. Our results are summarized in Figure 4. The x-axis shows the length of the test case and the y-axis shows the number of faults. There are four lines in the graph, one for each application. The results show that the number of faults detected grows with test case length. For TerpSpreadSheet, we detected less faults with length 3 test cases than with length 2 test cases because we had chosen a small subset of length 3 test cases for smoke testing (Table 3).

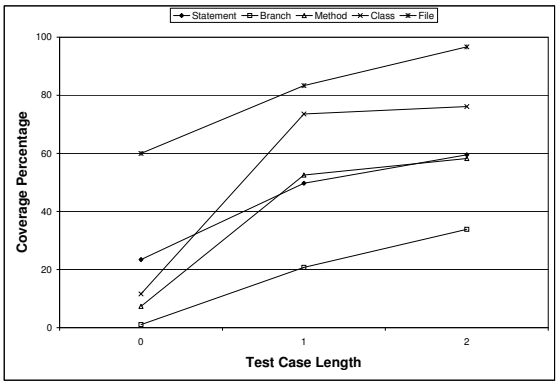
Faults detected, test oracle, and test case length. *Conclusion: The test oracle has a significant impact on the*



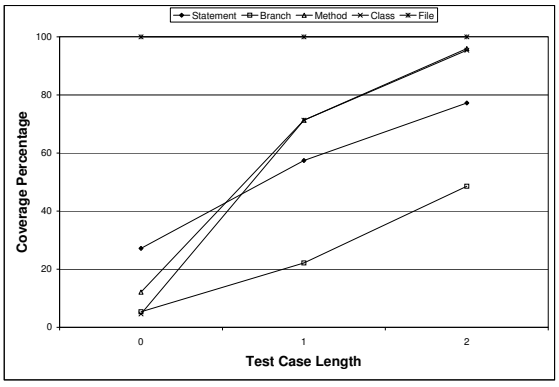
TerpWord



TerpSpreadSheet



TerpPaint



TerpCalc

Figure 3. Code Coverage of the Smoke Tests

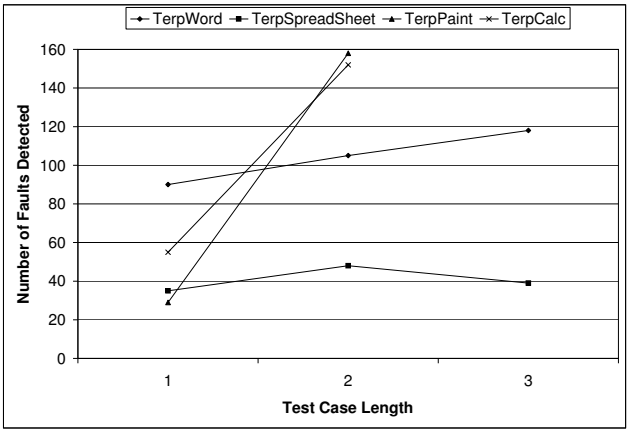
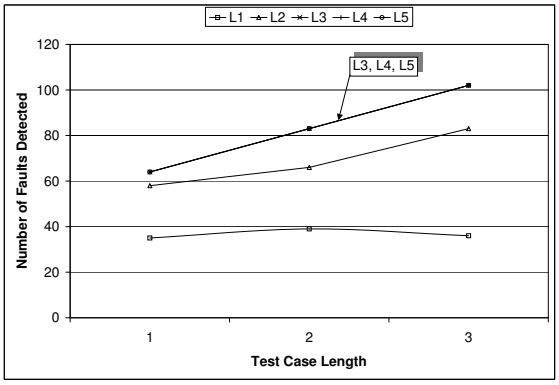


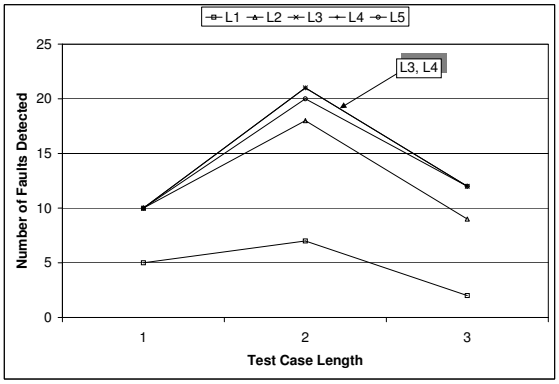
Figure 4. Faults vs. Test Length

fault-detection effectiveness of the smoke tests. Our lack of ability to automatically recognize the TerpSpreadSheet faults led us to study the effects of test oracles on the overall effectiveness of the smoke test cases. We show the results in Figure 5. The figure contains four line graphs, one for each subject application. For each graph, the x-axis shows the length of the test case, and the y-axis shows the number of faults detected. There are 5 lines in each graph, one for each oracle. The results show that oracle L1 was least effective at fault detection, L2 was better than L1, and L3, L4, and L5 were the best and they all performed equally well. Table 5 provides some explanation for this behavior. Examining Table 5, we see that L2 examined significantly more objects than L1 during test case execution. L3 and L4 examined almost the same number of objects because our subject applications don't have many invisible windows. L5 performed as well as L3 and L4 because of the nature of faults and the GUIs. Most of the faults were "persistent", i.e., once they manifested themselves, they remained "detectable" until the end of test case execution, thereby enabling L5 to detect them.

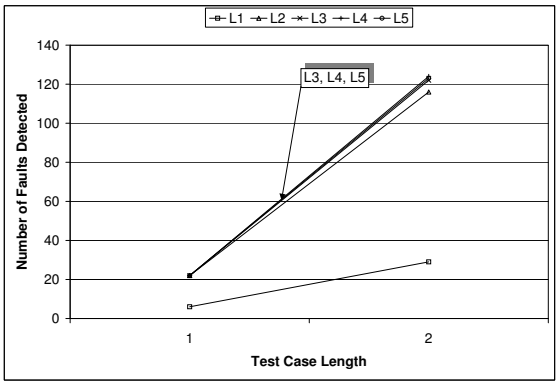
Smoke testing cost. *Conclusion: L5 provides the best combination of cost and fault-detection effectiveness.* Given the above results of test oracles in terms of fault-detection effectiveness, we wanted to observe the cost of deploying each oracle, in terms of time and disk space. Note that we were unable to accurately measure the time for L1. Since the Java Swing API did not allow direct access to the current widget and its properties, our implementation of L1 required accessing the active window, travering the widgets and locating the current widget, and then examining its properties. The time that we obtained was more than that required for L2, which we feel is misleading. Hence we omit reporting the time required for L1. The results are summarized in Tables 7 and 8. The results clearly show that L5 was much cheaper than L2, L3, and L4. As Table 7



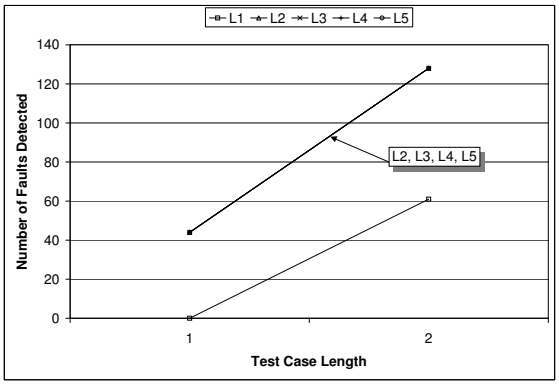
TerpWord



TerpSpreadSheet



TerpPaint



TerpCalc

Figure 5. Fault Detection by Oracle

Subject Application	L2 (ms)	L3 (ms)	L4 (ms)	L5 (ms)
TerpWord	22933	51526	78293	32010
TerpSpreadSheet	18009	33206	59588	21263
TerpPaint	274000	622739	1034536	398611
TerpCalc	89317	103612	126543	44104
TOTAL	404259	811083	1298960	495987

Table 7. Time Required for Each Oracle

	L1(MB)	L2 (MB)	L3 (MB)	L4 (MB)	L5 (MB)
Subject Application					
TerpWord	0.21	7.63	13.71	13.85	2.78
TerpSpreadSheet	0.24	12.63	25.04	29.49	7.09
TerpPaint	0.29	29.89	34.61	34.91	10.12
TerpCalc	0.34	33.19	33.25	44.22	13.38
TOTAL	1.08	83.34	106.61	122.47	33.37

Table 8. Space Required for Each Oracle

shows, the time needed to execute L5 was comparable to that of L2 and the space required was less than that required by L2; L5 detected many more faults than L2.

6 Conclusions

In previous work, we had presented the design of DART, that automatically smoke tested GUI event interactions. In this paper, we extended our previous work and demonstrated, via experiments, several characteristics of GUI smoke tests. We also combined smoke tests with different types of test oracles. Our experimental subjects consisted of four GUI-based applications. We generated 5000-8000 test cases for each application. We showed that (1) short GUI smoke tests with certain test oracles are effective at detecting a large number of faults, (2) there are classes of faults that our smoke test cannot detect, (3) short smoke tests execute a large percentage of code, and (4) the entire smoke testing process is feasible to do in terms of execution time and storage space.

We are currently designing a custom test orcle for TerpSpreadSheet that will examine the contents of the individual cells, hence helping to improve the fault-detection effectiveness of the smoke tests for that application. In the future, we will design an interface for DART that will allow for the definition of such domain-specific test oracles. We will study fault injection techniques for GUIs and incorporate them into our smoke test cases. We will also examine the effects of the execution environment on the fault detection effectiveness of the smoke test cases.

We feel that the GUI-based smoke test cases should be used together with code-based smoke tests. We will study the characteristics of both these types of test cases, evaluate their strengths/weaknesses and devise a mechanism to combine their strengths. We will develop techniques to partition

large smoke test suites, thus enabling the test designer to run one part each night, and combine the individual results.

References

[1] Cruise Control, 2003. http://cruisecontrol.sourceforge.net/.

[2] Current Daily Builds of AceDB, 2003. http://www.acedb.org/Software/Downloads/daily.shtml.

[3] FAST C++ Compilation - IcrediBuild by Xoreax Software, 2003. http://www.xoreax.com/main.htm.

[4] Java Source Code Instrumentation, 2003. http://www.glenmccl.com/instr/instr.htm.

[5] Kinook Software - Automate Software Builds with Visual Build Pro, 2003. http://www.visualbuild.com/.

[6] Mozilla, 2003. http://ftp.mozilla.org/pub/mozilla/nightly/latest/.

[7] Open WebMail, 2003. http://openwebmail.org/openwebmail/download/redhat/rpm/daily-build/.

[8] Positive-g- Daily Build Product Information - Mozilla, 2003. http://positive-g.com/dailybuild/.

[9] WINE Daily Builds, 2003. http://wine.dataparty.no/.

[10] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. Softw. Eng.*, 16(2):166–182, 1990.

[11] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[12] L. Crispin, T. House, and C. Wade. The need for speed: automating acceptance testing in an extreme programming environment. In *Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 96–104, 2001.

[13] M. Finsterwalder. Automating acceptance tests for GUI applications in an extreme programming environment. In *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 114 – 117, May 2001.

[14] J. Grenning. Launching extreme programming at a process intensive company. *IEEE Software*, 18:27–33, 2001.

[15] T. J. Halloran and W. L. Scherlis. High quality and open source software practices. In *Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*, May 2002.

[16] M. J. Harrold, A. J. Offut, and K. Tewary. An approach to fault modelling and fault seeding using the program dependence graph. *Journal of Systems and Software*, 36(3):273–296, Mar. 1997.

[17] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of *SPECIAL SESSIONS: Demonstrations*, page 1042, 1993.

[18] E.-A. Karlsson, L.-G. Andersson, and P. Leion. Daily build and feature development in large distributed projects. In *Proceedings of the 22nd international conference on Software engineering*, pages 649–658. ACM Press, 2000.

[19] B. Marick. When should a test be automated? In *Proceedings of The 11th International Software/Internet Quality Week*, May 1998.

[20] B. Marick. Bypassing the GUI. *Software Testing and Quality Engineering Magazine*, pages 41–47, Sept. 2002.

[21] S. McConnell. Best practices: Daily build and smoke test. *IEEE Software*, 13(4):143–144, July 1996.

[22] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[23] A. M. Memon. Advances in GUI testing. In *Advances in Computers, ed. by Marvin V. Zelkowitz*, volume 57. Academic Press, 2003.

[24] A. M. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the International Conference on Software Maintenance 2003*, pages 410–419, September 2003.

[25] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, pages 260–269, Nov. 2003.

[26] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 164–173. IEEE Computer Society, Oct.12–19 2003.

[27] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.

[28] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, NY, Nov. 8–10 2000.

[29] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb. 2001.

[30] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267, Sept. 2001.

[31] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *International Symposium on Software Testing and Analysis*, pages 195–200, 1996.

[32] K. Olsson. Daily build - the best of both worlds: Rapid development and control. Technical report, Swedish Engineering Industries, 1999.

[33] C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *IEEE Software*, 18:42–50, 2001.

[34] J. Robbins. *Debugging Applications*. Microsoft Press, 2000.

[35] P. Schuh. Recovery, redemption and extreme programming. *IEEE Software*, 18:34–41, 2001.

[36] L. White, H. AlMezen, and N. Alzeidi. User-based testing of GUI sequences and their interactions. In *Proceedings of the 12th International Symposium Software Reliability Engineering*, pages 54 – 63, 2001.