ABSTRACT

Title of dissertation: Developing Cost-Effective Model-Based Techniques for GUI Testing

Qing Xie, Doctor of Philosophy, 2006

Dissertation directed by: Professor Atif Memon
Department of Computer Science
University of Maryland, College Park

Most of today's software users interact with the software through a graphical user interface (GUI), which constitutes as much as 45-60% of the total code. The correctness of the GUI is necessary to ensure the correctness of the overall software. Although GUIs have become ubiquitous, testing GUIs for functional correctness has remained a neglected research area. Existing GUI testing techniques are extremely resource intensive primarily because GUIs have very large input spaces and evolve frequently. This dissertation overcomes the limitations of existing techniques by developing a process with supporting models, techniques, and tools for continuous integration testing of evolving GUI-based applications. The key idea of this process is to create three concentric testing loops, each with specific GUI testing goals, resource usage, and targeted feedback. The innermost fully automatic loop called *crash testing* operates on each code change of the GUI software. The second semi-automated loop called *smoke testing* operates on each day's GUI build. The outermost loop called *comprehensive GUI testing* is executed after a major version of the GUI is available. The primary enablers of this process, also devel-

oped in this dissertation, include an abstract model of the GUI and a set of model-based techniques for test-case generation, test oracle creation, and continuous GUI testing. The model and techniques were obtained by studying GUI faults, interactions between GUI events, and why certain event interactions lead to faults. The continuous testing process and associated techniques are shown to be useful, via several large experiments involving millions of test cases, on both in-house and open-source GUI applications.

Developing Cost-Effective Model-Based Techniques for GUI Testing

by

Qing Xie

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:
Professor Atif Memon, Chair/Advisor
Professor Victor Basili
Professor Rance Cleaveland
Professor Michel Cukier
Professor Adam Porter
Professor Gang Qu

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Professor Atif Memon for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past several years. He has always made himself available for help and advice and there has never been an occasion when I have knocked on his door and he has not given me time. It has been a pleasure to work with and learn from such an extraordinary individual. This experience will benefit me forever.

Thanks are due to Professor Victor Basili, Professor Rance Cleaveland, Professor Michel Cukier, Professor Adam Porter, and Professor Gang Qu for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

I thank all my teachers in schools, colleges, and universities whose dedication and hard work helped lay the foundation for this work.

My colleagues at the Software Testing Group have enriched my graduate life in many ways and deserve a special mention. My interaction with Bin Gan, Xun Yuan, Jaymie Strecker, Cyntrica Eaton, Adithya Nagarajan, Ishan Banerjee, and Lorin Hochstein has been very fruitful.

I would also like to acknowledge help and support from some of the staff members in the department.

I owe my deepest thanks to my family - my mother, father, and brother who have always stood by me and guided me through my career; my husband, Feng, for always

being there to support me and be constant source of encouragement during my Ph.D; my daughter, Katie, for bringing me endless happiness.

I would like to acknowledge financial support from the NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421, for all the projects discussed herein.

Last, but far from least, I want to express my thanks to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Testing is widely recognized as a key quality assurance (QA) activity in the software development process. Although research in testing has received considerable attention in the last two decades [22], testing of *graphical user interfaces* (GUIs), which constitute as much as 45-60% of the total software code [45], has remained until recently, a neglected research area [32]. Because GUI software has become nearly ubiquitous, neglecting the quality of GUI software has the potential to have a negative impact on all of today's software.

A software with a GUI front-end consists of two parts : (1) the underlying code that implements the "business logic" and (2) the GUI front-end that facilitates user interaction with the underlying code. A software user interacts with the GUI by performing *events*, such as button clicks, menu selections, and text inputs. The GUI uses the input events to interact with the underlying code via messages and method invocations. During GUI testing, test cases, modeled as sequences of events are executed on the GUI and its output is compared to an "expected output." The goal of GUI testing is to reveal *GUI faults* (defined as one that manifests itself on the visible GUI at some point of time during the software's execution).

Several researchers have exploited the event-driven nature of GUIs to develop automated model-based GUI testing techniques (*e.g.*, AI planning [38], event-flow graph [32],

complete interaction sequences [62]). However, these techniques have not been adopted by GUI testers because of several problems: (1) the models are expensive to obtain (except for event-flow graphs); they are typically created manually, (2) the number of permutations of all possible GUI interactions (event sequences) with the user is enormous; these techniques test the GUI for a small sub-space of user interactions; it remains unclear whether testing this sub-space reveals any GUI faults, and (3) GUIs are typically developed using agile processes, which are known for their simple planning, short iterations, and are driven by frequent customer feedback. It becomes expensive to update the models and test artifacts (*e.g.*, test cases, test oracles) during frequent software/GUI updates.

Moreover, because modern software is typically developed by multiple programmers, another GUI testing challenge largely ignored by existing techniques is that the programmers are likely to "break" the GUI software during their local code updates. Programmers are generally unwilling and, due to limited resources, unable to setup an expensive GUI testing process for each update. If left undetected, the cascading effect of these updates may lead to integration faults that cause substantial delays during GUI integration testing.

The research presented in this dissertation overcomes the limitations of existing techniques. Specifically, the contributions of this research include:

- the development of new cost-effective, automated GUI testing techniques that are applicable to rapidly evolving GUI software,
- development of new GUI models that are inexpensive to obtain and maintain,
- demonstration of the fault detection effectiveness of the new techniques, and

- development of a continuous GUI testing process that targets feedback to specific developers.

The remainder of the chapter outlines the steps necessary for GUI testing and the challenges that GUI testers face for each step, followed by a discussion of existing GUI testing techniques and their limitations, and a high-level overview of the research presented in this dissertation.

1.1 What is a GUI?

Most of today's software users interact with the software through a GUI. The user typically uses a mouse and a keyboard to interact with GUI *widgets*. Widgets of a GUI include elements such as windows, pull-down menus, buttons, scroll bars, text boxes, and icons. The software user performs *events* on these widgets, such as clicking a button, selecting a menu item, and typing in a text box. These events cause deterministic changes to the state of the software that may be reflected by a change in the appearance of one or more GUI widgets.

The important characteristics of GUIs include their graphical orientation, event-driven input, the widgets they contain, and the properties (attributes) of those widgets. Since GUIs may be used as front-ends to many different types of software applications, the space of all possible GUIs is enormous. It would be extremely difficult to create one model for all possible types of GUIs. Hence, to provide focus, this research models a sub-class of GUIs. Specifically, the GUIs in this sub-class react to events performed only by a single user; the events are deterministic, *i.e.*, their outcomes are completely predictable.

Testing GUIs that react to temporal and non-deterministic events and those generated by other applications is beyond the scope of this research.

1.2 GUI Testing Process

To better understand the complexity associated with GUI testing, this section gives an overview of its steps. Typically, GUI testing involves the following tasks.

1. *Test case generation:* A GUI test case is a sequence of events, *e.g.*, button clicks, menu selections, and text inputs. A tester generates test cases by enumerating sequences of GUI events either manually [57] or by using a model of the GUI [38]. General "common sense" guidelines (*e.g.*, "each GUI function (print, file-open, file-save) is tested at least once") may be used to guide test case generation.

2. *Expected output generation:* The expected output is used to check the correctness of the GUI during test-case execution. The tester specifies the expected output for each GUI event either manually (*e.g.*, via assertions) or by using formal specifications [50]. The expected output may be in the form of screen snapshots, and window positions, titles and contents.

3. *Test case execution and output verification:* Execution of the GUI's test case is done by performing all the input events specified in the test case and comparing the actual GUI's output to the expected output. An assertion violation and/or a mismatch between the expected and actual output is reported as an error.

4. *Coverage analysis:* Once all the test cases have been executed on the GUI, coverage criteria (*e.g.*, "all program statements covered at least once," "all branches covered at least once") are used to evaluate the coverage of the test cases. Testing is considered complete once the coverage criteria have been satisfied.

Because GUIs are typically designed using agile processes and rapid prototyping [46], the above steps may be executed multiple times during the GUI development process to re-test it. Re-testing involves analyzing the changes to the layout of GUI objects, maintaining the test artifacts, and rerunning the test cases. Test artifact maintenance may involve selecting test cases that should be rerun, generating new test cases with their associated test oracles, and deleting *obsolete* test cases (*i.e.*, those that cannot be rerun on the modified GUI).

The above process, as described, is "ideal." However, as is the case with all testing techniques, in practice the above steps present problems, which are described next.

1.3 Challenges of GUI Testing

First, it is difficult to generate test cases because the number of permutations of interactions with a GUI is enormous in that each sequence of GUI events can result in a different state, and a GUI event may, in principle, need to be tested in all of these states. Consequently, the number of test cases required to test the GUI is very large.

Second, it is difficult to specify the expected output for a GUI test case. As is typically the case with reactive software, an event in the test case may lead to an incorrect state in which subsequent events cannot be executed. Execution of the test case must

be terminated as soon as an error is detected. To reveal such problems, GUI test case execution requires that verification and test case execution be interleaved. Hence the expected output needs to be specified for each event in the test case. This is, of course, a resource intensive task. Moreover, it is expensive to check the correctness of the GUI after each event during test case execution.

Third, it is difficult to evaluate the adequacy of GUI test cases. Traditional adequacy criteria are based on code. However satisfying these criteria does not necessarily imply that problematic interactions between GUI events have been tested. New, specialized criteria are needed for GUIs that evaluate the adequacy of tested GUI interactions.

Fourth, it is difficult to perform regression testing of GUIs. Because GUIs are developed using agile processes, they are modified on a continuous basis thereby needing frequent re-testing. Consequently, the previously generated test cases or test oracles may become obsolete when testing the new version of the GUI. Regenerating new test cases and test oracles is either done manually or with model updates (if using model-based techniques); both are resource-intensive activities.

Finally, an orthogonal challenge is that, because modern software development typically involves multiple (geographically distributed) developers working on different parts of the software, there is little direct inter-developer communication [54]. Almost all communication is done via web-based tools such as CVS commit log messages, bug reports, change-requests, and comments [11, 51]. Sub-groups within developer communities often work on loosely coupled parts of the application code [54]. Each developer (subgroup) typically modifies a local "copy" of the code and frequently checks-in changes (and checks-out other developers' changes). Consequently, after making a change, a de-

veloper may not immediately realize that the local change has inadvertently broken other parts of the overall software code [30]. In such situations, the developer needs quick feedback of newly introduced faults, enabling quick fixes. If left undetected, the cascading effect of these faults may lead to wasted debugging cycles during development and expensive quality assurance later. Moreover, intermediate fielded releases of the GUI have questionable quality.

Several researchers have proposed new techniques to address some of the above challenges, specifically for test case generation. A summary of the techniques and their limitations is presented next.

1.4 Existing Approaches and their Limitations

The most popular GUI testing approach is to use semi-automated tools to do limited testing [19, 63]. Examples of some tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module* [2] to create unit tests for GUIs. Other tools include capture/replay tools that "capture" a user session as a test case that can be later "replayed" automatically during regression testing [25]. These tools facilitate only the execution of test cases; creating and maintaining test cases is very resource-intensive.

Several researchers have developed techniques to automate some aspects of GUI testing. In the work by Memon *et al.* [32,38], an automated GUI testing framework called PATHS has been developed. PATHS uses a description of the GUI to automatically generate test cases and test oracles from pairs of initial and goal states by using an AI planner. Although this approach is successful in automating test case generation, the output (*i.e.*,

test cases) largely depends on the choice of tasks given to the planner, which may yield an inadequate test suite. This approach is also resource-intensive because testers have to manually create and maintain an "operators" file for the planner. Moreover, there is no evidence showing that the test cases generated by PATHS are effective at detecting faults.

The other significant work on GUI testing is by White *et al.* [62, 64] who model a GUI in terms of "responsibilities" (user tasks) and their corresponding "complete interaction sequences" (CIS). A CIS is a sequence of GUI objects and selections that may be used to complete a responsibility. Each CIS contains a reduced finite-state machine (FSM) model, which is "traversed" to generate test cases [62]. This technique is very resource-intensive because the test designer has to manually identify the responsibilities and the associated CISs each time the GUI is modified. Moreover, there are no studies demonstrating the fault detection effectiveness of the generated test cases.

Other researchers have developed techniques to address isolated problems of GUI testing. For example, a variable finite state machine based approach to generate test cases has been proposed by Shehady *et al.* [55]. Details of these techniques are presented in Chapter 2. In summary, all of these techniques suffer from relatively similar problems. They are all resource intensive, they address only one specific aspect of GUI testing, the fault detection effectiveness of the test cases generated by these techniques has not been demonstrated, and they handle the same (or in some cases weaker) class of GUIs defined in Section 1.1. Moreover, whenever the GUI is modified, new test cases and associated test oracles have to be recreated/regenerated to substitute the existing obsolete ones. The agile nature of GUI development requires the development of new GUI testing techniques that are themselves agile in that they quickly test each increment of the GUI

during development.

1.5 A New Continuous GUI Testing Process

The primary research contribution of this dissertation is a process with supporting models, techniques and tools for continuous integration testing of GUI-based applications. The key idea of this process is to partition the GUI testing problem via concentric testing loops, each with specific test criteria, GUI testing goals, resource usage, and targeted feedback. This dissertation presents three loops. The innermost loop is executed very frequently and hence is designed to be fully automatic. The goal is to perform a quick-and-dirty, fully automatic integration test of the GUI software with a fixed time interval and give immediate feedback to the developers. The second loop is executed nightly/daily and hence is designed to complete within 8-10 hours; it allows some manual intervention. The third, and outermost loop conducts comprehensive GUI integration testing, may require significant manual effort, and hence is the most expensive. The continuous testing process takes the agile nature of GUI development into consideration. It overcomes the limitations of other model-based techniques that require frequent manual model updates.

An overview of one instance of this process is shown in Figure 1.1. In this particular instance of the concentric-loop-based process, the innermost loop executes a fully automatic process called *crash testing* on each code check-in (*e.g.,* using CVS) of the GUI software [66]. The duration for crash testing is defined by the developer. Software crashes (abnormal terminations) are reported back to the developer who initiated the

check-in. Crash test criteria include covering the entire functionality of the GUI (via a graph model of the GUI called the event-interaction graph described in Chapter 4) and detecting crashes. The second loop executes a semi-automated process called *smoke testing* operates on each day's GUI build [30, 34, 41, 43]. It performs functional "reference testing" (discussed in Chapter 2) of the newly integrated version of the GUI. As is typically the case with reference testing, differences between the outputs of the previous (yesterday's) build and the new build are reported to the developers who contributed to the latest build. Smoke test criteria also involve covering the entire functionality of the GUI; in addition, it requires the detection of differences between two consecutive versions of the software. Finally, the outermost loop executes a process (which may be manual with supporting tools) called *comprehensive GUI testing* after a major version of the GUI is available. Comprehensive GUI test criteria are specific to the goals of the organization in which testing is being performed. In Figure 1.1, the small octagons represent frequent CVS code check-ins. The encompassing rectangles with rounded corners represent daily increments of the GUI. The large rectangle represents the major GUI version. The three loops discussed earlier are shown operating on these software artifacts. In this dissertation, the terms crash testing, smoke testing, and comprehensive GUI testing will be used for the inner, intermediate, and outermost loops, respectively. Even though the work in this dissertation has been shown for three loops, it may be extended to other loops.

Several techniques were developed as part of this research to enable the above process. Each technique is a new research contribution of this dissertation and has been presented in the research literature [30,41–43,65–70]. First, a new GUI model that represents potentially problematic event interactions is developed. This model is obtained by using

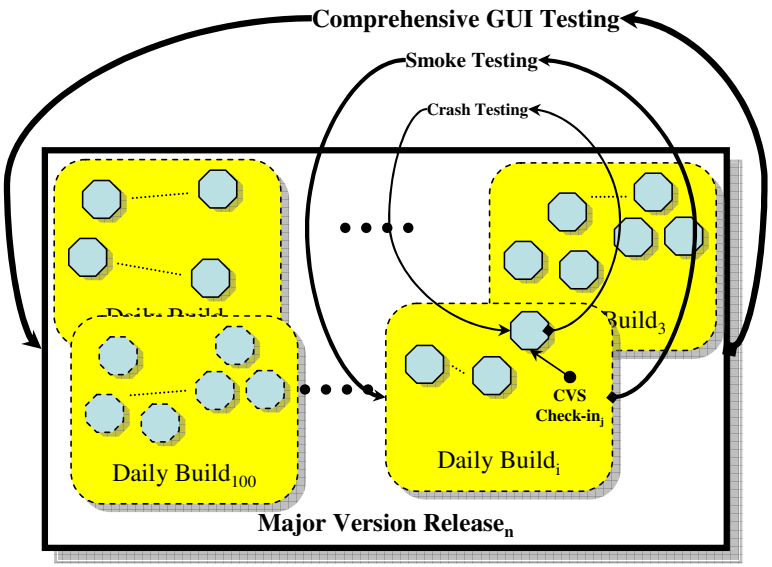


Figure 1.1: Different Loops of Continuous GUI Testing

automated techniques that employ reverse engineering to eliminate manual work [35]. The model is then used to generate test cases, create descriptions of expected execution behavior, and evaluate the adequacy of the generated test cases. Automated test executors "play" these test cases on the GUI and report errors. Second, new test case generation techniques quickly generate "crash" and "smoke" tests that execute very quickly. Third, during smoke testing, which executes a form of reference testing, efficient test oracles enable the process to complete in 8-10 hours. Fourth, new techniques assist developers/testers to make tradeoff decisions during comprehensive testing. Finally, the fault detection effectiveness of all the techniques is empirically evaluated on several open-source GUI subjects developed in-house and downloaded from SourceForge.

Another (implicit) contribution of this dissertation that has an encompassing effect on all aspects of this research is the development of an infrastructure for experimentation in GUI testing. This infrastructure was implemented as an extension of an existing tool

called the GUI Testing frAmewoRk (GUITAR) and GUI subject applications for experimentation. The GUITAR extension allowed the automatic generation and execution of millions of test cases. Several subject applications were seeded with hundreds of artificial faults and used for all the experiments discussed in this dissertation. They have also been shared with other researchers who have used them for their experiments [44].

1.6 Structure of the Dissertation

The next chapter introduces relevant literature and related work. Chapter 3 provides an overview of the continuous GUI testing process. Chapter 4 through Chapter 6 present each GUI testing loop respectively, namely, crash testing, smoke testing, and comprehensive testing, and the techniques developed to support these loops. Finally, Chapter 7 concludes with a discussion of the merits of this research and possible future directions.

Chapter 2

Background and Related Work

The goal of testing is to detect the presence of errors in programs by executing the programs on well-chosen input data. An error is said to be present when either (1) the program's output is not consistent with the specifications, or (2) the test designer determines that the specifications are incorrect. Detection of errors may lead to changes in the software or its specifications. These changes then create the need for re-testing.

Testing requires that test cases be executed on the software under test and the software's output be compared with the expected output by using a test oracle. The input and the expected output are a part of the test suite. The test suite is composed of tests each of which is a triple $< identifier, input, output >$, where $identifier$ identifies the test, $input$ is the input for that execution of the program, and $output$ is the expected output for this input. The entire testing process for software systems is done using test suites.

Information about the software is needed to generate the test suite. This information may be available in the form of formal specifications or derived from the software's structure leading to the following classification of testing.

Black-box testing (also called *functional testing* [7] or *testing to specifications*) is a technique that does not consider the actual software code when generating test cases. The software is treated as a black-box. It is subjected to inputs and the output is verified for conformance to specified behavior. Test generators that support black-

box testing require that the software specifications be given as rules and procedures. Examples of black-box test techniques are equivalence class partitioning, boundary value analysis, and cause-effect graphing.

White-box testing (also called *glass-box testing* [7] or *testing to code*), as the name suggests, is a technique that considers the actual implementation code for test case generation. For example, a *path oriented* test case generator selects a program's execution path and generates input data for executing the program along that path. Other popular techniques make use of the program's branch structure, program statements, code slices, and control flow graphs (CFG).

No single technique is sufficient for complete testing of a software system. Any practical testing solution must use a combination of techniques to check different aspects of the program.

Zhu et al. [71] provide a comprehensive survey of existing testing techniques. One classification of techniques presented therein is based on the source of information used to specify the testing criteria. This classification defines testing as either *specification based*, *program based*, or *interface based*. Of interest to this research is the interface based testing that specifies testing criteria in terms of the type and range of software input without reference to any internal features of the program code or the specifications. Interface based testing remains an open area for research.

Automated software testing research has received significant attention in the last three decades. There are several books that describe the wide spectrum of techniques available for automated testing [8, 15, 18, 20, 24, 49, 56]; it is impossible to summarize all

the material here. The research presented in this dissertation develops new cost-effective, model-based GUI testing techniques to realize the continuous GUI testing process shown in Figure 1.1. The research spans the areas of GUI representation, test case generation, test oracle creation, test coverage, regression testing, rapid feedback-based QA mechanisms, and fault seeding. This chapter introduces relevant terms and existing approaches used in these areas and provides pointers to additional sources of detailed information. It should be noted that all existing GUI testing techniques, including ones developed in this dissertation, handle the same (or sometimes weaker) class of GUIs defined in Section 1.1.

2.1 GUI Representation

Several researchers have developed different types of GUI representations for specific testing tasks. The GUI representation that is used as a starting point for this research has been developed by Memon *et al.* [32]. Hence, it will be discussed in this section separately; other representations will be discussed coupled with their specific techniques. The representation consists of two parts: (1) the GUI's state in terms of GUI widgets, their properties, values, and the events that can be performed on the GUI and (2) the space of all possible interactions with GUI. The remainder of this section presents an overview of this representation.

2.1.1 GUI's State

A GUI is modeled as a set of *widgets* W (*e.g.*, `label`, `form`, `button`, `text`), a set of *properties* P of those widgets (*e.g.*, `background-color`, `font`, `caption`),

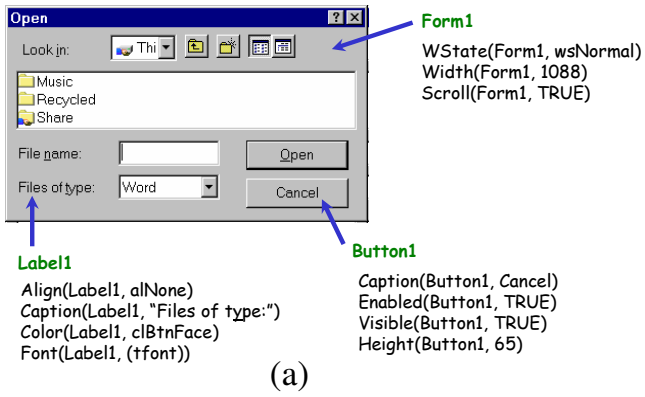


Figure 2.1: (a) `Open` GUI, (b) its Partial State

and a set of *values* V (*e.g.*, `red`, `12pt`, `"GUI"`) associated with the properties. Each GUI will use certain types of widgets with associated properties. At any point during its execution, the GUI can be described in terms of the specific widgets that it currently contains and the values of their properties.

For example, consider the `Open` GUI shown in Figure 2.1(a). This GUI contains several widgets, two of which are explicitly labeled, namely `Button1` and `Label1`; for each, a small subset of properties is shown. Note that all widget types have a designated set of properties and all properties can take values from a designated set.

The set of widgets and their properties is used to create a model of the *state* of the GUI. The *state* of a GUI at a particular time t is the set S of triples $\{(w_i, p_j, v_k)\}$, where $w_i \in W$, $p_j \in P$, and $v_k \in V$. The state of the GUI of Figure 2.1 (a) is shown in Figure 2.1 (b).

With each GUI is associated a distinguished set of states called its *valid initial state*

set. A set of states S_I is called the *valid initial state set* for a particular GUI iff the GUI may be in any state $S_i \in S_I$ when it is first invoked. The state of a GUI is not static; *events* $\{e_1, e_2, \ldots, e_n\}$ performed on the GUI are used to change its state. Events may be stringed together into sequences. Note that not all combinations of events need to be tested; only those that are allowed by the structure of the GUI are tested.

2.1.2 Event-flow Graphs

Memon *et al.* [38] model the space of all possible valid user interactions with the GUI as an event-flow graph (EFG). More formally, an EFG for a GUI G is a 4-tuple $<$**V**, **E**, **B**, **I**$>$ where:

1. **V** is a set of vertices representing all the events in G. Each $v \in$**V** represents an event in G.
2. **E** $\subseteq$ **V** $\times$ **V** is a set of directed edges between vertices. Event e_j `follows` e_i (or equivalently $e_j =$ `follows`(e_i)) iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in$ **E** iff the event represented by v_y `follows` the event represented by v_x.
3. **B** $\subseteq$ **V** is a set of vertices representing those events of G that are available to the user when the GUI is first invoked.
4. **I** $\subseteq$ **V** is the set of events that invoke other windows.

Note that an event-flow graph is not a state machine. The nodes represent events in the GUI (not states) and the edges represent the `follows` relationships (not state transitions). An example of an EFG for the `Main` and `Replace` windows of the MS NotePad

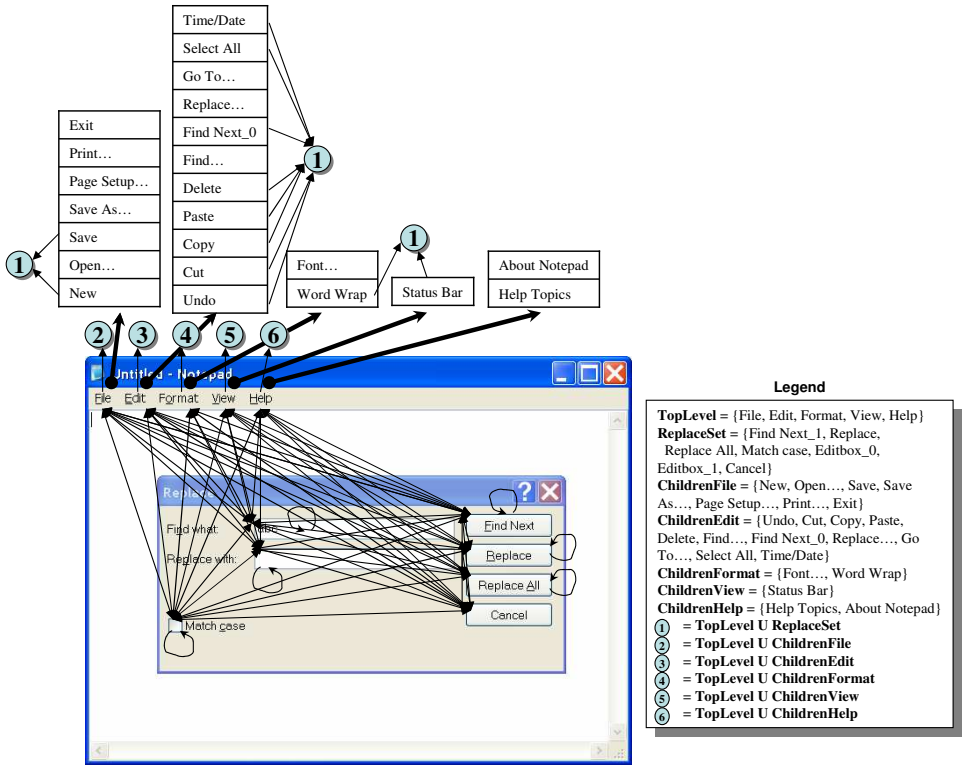


Figure 2.2: Example of an Event-Flow Graph

software is shown in Figure 2.2. Events (corresponding to each widget) are shown as labeled boxes. The labels show a meaningful unique identifier for each event. Directed edges show the `follows` relationship between events. For increased readability, only some of the edges are shown. Sets of events are defined and listed in a Legend. For example `TopLevel` is a set containing the events `File`, `Edit`, `Format`, `View`, and `Help`. Similarly ① is a set containing all the events in `TopLevel` and `ReplaceSet`. Note that `Editbox_0` and `Editbox_1` in `ReplaceSet` represent the two events used to edit the text boxes in the Replace window. An edge from `Copy` to ① represents a number of edges, from `Copy` to each event in ①. According to this EFG, the event `Cancel` can be executed immediately after the event `Find Next`; event `Match case` can be executed after itself; however, event `Replace` cannot be executed immediately after event

`Cancel`.

The concepts of events, widgets, properties, and values are used to formally define a GUI test case. A **GUI test case** T is a pair $< S_0, e_1; e_2; \ldots; e_n >$, consisting of a state $S_0 \in S_I$, called the *initial state for T*, and an event sequence $e_1; e_2; \ldots; e_n$ such that e_i `follows` $e_{i-1}, 2 \leq i \leq n$.

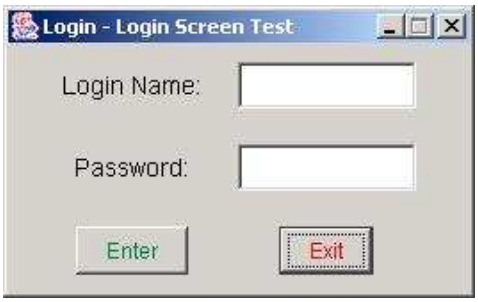
2.2 Test Case Generation

This section presents GUI test case creation/generation techniques partitioned into two categories: (1) manual approaches and (2) model-based approaches; for each approach, the advantages and limitations are presented.

2.2.1 Manual Approaches

As mentioned Briefly in Chapter 1, there are several GUI testing tools used for limited testing [19, 63]. Examples of tools include extensions of *JUnit* such as *JFCUnit*, *Abbot*, *Pounder*, and *Jemmy Module* [2] that help testers to manually create unit tests for GUIs, and capture/replay (record/playback) tools [25] that "capture" a user session that can be "replayed" automatically during regression testing. These tools provide very little automation [33], especially for *creating* test cases, as demonstrated next.

In order to test the GUI with unit testing tools, testers have to write JFCUnit test cases to simulate various types of event activities. Figure 2.3 shows (a) a simple login screen example that needs to be tested, and (b) part of a JFCUnit test case for this screen. The test case consists of five parts: (1) specifying and writing the input to the text fields,



(a)

```
public void testLoginScreen() {
    ...

    // (1) type in "qing" into the login name textbox, and "whatever" to the password textbox
    getHelper().sendString( new StringEventData( this, userNameField, "qing" ) );
    getHelper().sendString( new StringEventData( this, passwordField, "whatever" ) );

    // (2) click on "Enter" button
    getHelper().enterCickAndLeave( new MouseEventData( this, enterButton ) );

    // (3) waiting for response
    DialogFinder dFinder = new DialogFinder(null);
    dFinder.setWait(0);

    // (4) login screen window is disposed
    showingDialogs = dFinder.findAll( loginScreen );
    assertEquals( "Number of dialogs showing is wrong", 0, showingDialogs.size( ) );
    assertTrue( "Login screen is showing still", !loginScreen.isShowing( ) );

    // (5) main GUI window shows up
    FrameFinder fFinder = new FrameFinder(null);
    showingWindows = fFinder.findAll();
    assertEquals( "Number of windows showing is wrong", 1, showingWindows.size( ) );
     …
}
```

(b)

Figure 2.3: (a) A Simple GUI and (b) Example of a JFCUnit Test Case

(2) hitting the `Enter` button, (3) waiting for a response from the GUI, (4) checking if the login window is disposed, and (5) checking for the next expected window. This test case is extremely simplified for illustration. As can be imagined, coding a more realistic test case is extremely labor-intensive; the tester must predict several possible outcomes and event permutations.

Some of the manual effort required to develop test cases may be reduced by using Capture/replay tools. These tools (also called record/playback tools) operate in two modes: *Record* and *Playback*. In the *Record* mode, tools such as *CAPBAK* and *TestWorks* [57] record mouse coordinates of the user actions as test cases (test scripts). In the *Playback* mode, the recorded test cases are replayed automatically. The problem with such tools is that because they store mouse coordinates, test cases break even with the slightest changes to the GUI layout. Tools such as *Winrunner* [4], *Abbot* [3], and *Rational Robot* [5] avoid this problem by capturing GUI widgets rather than mouse coordinates. Although playback is automated, significant effort is involved in creating the test scripts, detecting failures, and editing the tests to make them work on the modified version of the software. Experience with GUI testing shows that a tester cannot use these tools to develop a test suite that covers a significant portion of the GUI (an extremely resource-intensive task) for continuous testing [32]. Test cases obtained from capture/replay tools are very fragile and most of them become unusable after a few GUI modifications [32]. Test cases become unusable for the modified GUI either because the input event sequence can no longer execute on the GUI or because the expected output stored with the test case becomes obsolete.

2.2.2 Model-based Approaches

Several model-based approaches have been used for GUI test case generation including state-machines, AI planning, event-flow graphs, and genetic algorithms. They all address the GUI testing problem for the same subclass of GUIs defined earlier in Section 1.1. Each of these approaches is presented next.

State-machine based approaches: It is relatively easy to model a GUI with a finite-state machine (FSM): each user action leads to a new state and each transition models an event. A path in the FSM represents a test case and the FSM's states are used to verify the software's state during test case execution. Several FSM based models have been used to generate test cases [9, 12, 17]. However, a major limitation of this approach, which is especially important for GUI testing, is that FSM models have scaling problems when applied to GUI test case generation [53]. Slight variations such as variable finite state machine (VFSM) models have been proposed by Shehady *et al.* [55]. These variations help scalability but verification checks need to be inserted manually at points determined by the test designer.

White *et al.* [62] model a GUI in terms of "responsibilities" (user tasks) and their corresponding "complete interaction sequences" (CIS). A CIS is a sequence of GUI objects and selections that may be used to complete a responsibility. For each CIS, a reduced FSM model is constructed. This FSM may be "traversed" to generate test cases. This technique helps to address the scalability challenge of using FSM models to generate test cases because the number of CIS sequences increases linearly with the size of the GUI. However, the technique requires a substantial amount of manual work on the part of the

test designer, who has to manually identify and maintain the responsibilities and associated CISs. Moreover, there are no studies demonstrating the fault detection effectiveness of the test cases.

AI planning based approaches: Memon *et al.* [32, 38] have developed an automated GUI testing framework called PATHS that uses AI planning to generate test cases. PATHS uses a description of the GUI to automatically generate test cases from tasks (pairs of initial and goal states) by iteratively invoking the planner. First, a test designer defines *planning operators* in terms of preconditions and effects. The test designer then describes tasks by identifying a set of initial and goal states. Finally, PATHS generates a test suite to achieve the goals.

While this approach is successful at automating test case generation, it has several limitations: (1) there is no evidence showing that PATHS generates test cases that are effective at detecting faults; (2) the test case generator is largely driven by the choice of tasks given to the planner; a poorly chosen set of tasks will yield an inadequate test suite; (3) the test oracle compares the expected and actual output once after each event, making test execution very slow; (4) the planner uses an "operators" file, which is resource intensive to create and maintain; (5) the coverage criteria used by PATHS require a prohibitively large test suite; and (6) regression testing is performed by repairing test cases that have become unusable for the modified GUI [39]; the fault detection effectiveness of the repaired test cases has not been demonstrated. The associated test oracles need to be re-created in some cases.

Event-flow graph based approaches: An EFG (described in Section 2.1.2) may be used to generate GUI test cases. A straightforward way to generate test cases is to start

from a known *initial state* of the GUI (*e.g.*, the state in which the software starts) and use a graph traversal algorithm to enumerate the nodes during the traversal of the EFG. If the event requires text input, *e.g.*, for a text-box, then its value is read from a database, initialized by the software tester. A sequence of events $e_1; e_2; \ldots; e_n$ is generated as output that serves as a GUI test case. This straightforward approach works well in certain situations. However, the number of event sequences grows very rapidly with length. It becomes infeasible to generate and execute all possible event sequences beyond $length$ (*i.e.*, number of events) > 2.

Genetic algorithm based approaches: Test cases have been generated to mimic novice users [27]. The approach uses an expert to generate the initial path manually and then uses genetic algorithm techniques to generate longer paths. The assumption is that experts take a more direct path when solving a problem using GUIs whereas novice users often take longer paths. Although useful for generating multiple test cases, the technique relies on an expert to generate the initial test case. The final test suite depends largely on the paths taken by the expert user.

2.3 Test Oracles

Once test cases have been generated, they are executed. A test oracle is a mechanism for determining whether or not the output from the GUI is equivalent to the expected output derived from the software's specifications. Several researchers have discussed the difficulty of creating test oracles for programs that have a large volume of output [14, 16, 60]; this is the case with GUI software, where each GUI screen is a part of

the program's output. A popular mechanism used as a GUI test oracle is based on *reference testing* [58, 59]. Actual outputs are recorded when the GUI software is executed for the first time. The recorded outputs are later used as expected output for regression testing. This is a popular technique used for regression testing of GUI-based software [61]. For example, testers may use capture/replay tools (discussed earlier) to assert specific widgets and some values of their properties for reference testing.

The only work on automated GUI test oracles has been done by Memon *et al.* [36, 39]. Figure 2.4 shows their design of the automated GUI test oracle. The *oracle information generator* automatically derives the *oracle information* (expected state) using either a formal specification of the GUI or by using another version of the GUI software, *e.g.*, for reference testing. Likewise, the *actual state* (also described by a set of widgets, properties, and values triples) is obtained from an *execution monitor*, which uses techniques such as screen scraping [37] and/or querying to obtain the actual state of the executing GUI. An *oracle procedure* then automatically compares the two states and determines if the GUI is executing as expected. A mismatch between the actual and expected states is called a GUI error.

Memon *et al.* have shown that the test oracle contributes significantly to test effectiveness and cost [36]. They create different types of oracles by varying the oracle information and procedure. Four types of oracle information in terms of *widget*, *active window*, *visible windows*, and *all windows* are created to represent the expected state of the widget on which the current event is being executed, all widgets that are part of the current active window, all currently visible windows, and all windows respectively. The oracle procedure may be invoked as frequently as once after each event of the test case

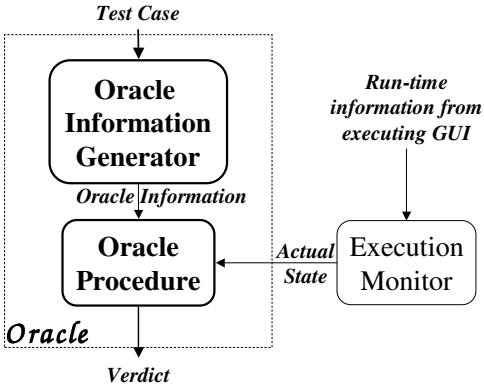


Figure 2.4: An Overview of the GUI Oracle

or after the last event. The limitation of their work is that the correlation between test effectiveness and cost has not been studied.

2.4 Test Coverage Criteria

Test coverage criteria provide measurements of test quality. Testing is considered complete once the coverage criteria have been satisfied.

The only coverage criteria for GUI testing have been presented by Memon *et al.* [40]. These criteria are based on the EFG. *Intra-component criteria* are used to evaluate the adequacy of tests on all the events in one window. *Inter-component coverage criteria* are used to evaluate the adequacy of test sequences that go across different windows.

Intra-component criteria include event coverage, event-interaction coverage and length-n event-sequence coverage. Event coverage requires each event in one window (each node in the EFG) to be executed at least once. Event-interaction coverage requires the interactions among all possible pairs of events in the window (each edge in the EFG) to be executed at least once. Length-n event-sequence coverage captures the contextual

impact on events; it requires all event-sequences of length equal to n (for values of n from 1 to a predetermined number i) be executed at least once. Event coverage and event-interaction coverage are two special cases of length-n event-sequence coverage, where n is 1 and 2 respectively.

Inter-component criteria consist of invocation coverage, invocation-termination coverage, and inter-component length-n event-sequence coverage. Invocation coverage requires that each GUI window be opened at least once. Invocation-termination coverage requires that each window be opened and immediately closed at least once. The inter-component length-n event-sequence coverage requires testing all length n event-sequences that start with an event in one window and end in another window. These test coverage criteria are successful in guiding GUI test case generation/selection; the limitation is that a test suite that satisfies these criteria for $n > 3$ for a non-trivial GUI is prohibitively large.

2.5 Regression testing

Regression testing is performed whenever modifications are made to either the software implementation or specifications. Regression testing is done to provide confidence that modifications have not adversely impacted the software's quality. However, it is not practical to test the modified software by rerunning all the test cases. Regression testing involves reusing some of the results from prior test runs. The main decisions involved are (1) which test cases to rerun, and (2) what new test cases to generate based on the changes made to the software. Regression testing for GUIs is extremely difficult because

the layout of GUI objects changes constantly, resulting in a large number of *obsolete* test cases.

White [61] developed a Latin square method to reduce the size of the regression test suite. The underlying assumption is that it is sufficient to check pairwise interactions between menu items of the GUI; each menu item needs to appear in at least one test case. This strategy seems promising since it too deals with GUI events. However, the technique needs to be extended to GUI items other than menus.

White *et al.* [64] also extended their CIS (described in Section 2.2) to develop a selective regression approach based on identifying the changed and affected objects and CISs. Besides the significant amount of manual effort needed to identify responsibilities and CISs for the GUI, this technique also requires the use of complex memory diagnostic tools, such as *Memory Doctor* and *WinGauge* to assist in fault detection.

Memon *et al.* [39] have presented a new regression testing technique for GUIs, which repairs test cases that have become unusable for the modified GUI. The first step is to determine the usable and unusable test cases from a test suite after a GUI modification, followed by identifying the unusable test cases that can be repaired so they can execute on the modified GUI. The last step is to repair the test cases. The idea is to maintain test cases rather than generate new ones since test cases generation is very time consuming and tedious. However, the fault detection effectiveness of the repaired test cases has not been evaluated.

2.6 Rapid Feedback-based QA mechanisms

There are several rapid feedback-based mechanisms to help manage the quality of evolving software developed by multiple developers. These mechanisms improve the quality of software via continuous, rapid quality assurance during evolution. They differ in the level of detail of feedback that they provide to targeted developers, their thoroughness, their frequency of execution, and their speed of execution.

Immediate-Feedback: For example, some mechanisms (*e.g.*, integrated with CVS) provide immediate feedback at change-commit time by running select test cases, which form the *commit validation suite*. Developers can immediately see the consequences of their changes. For example, developers of NetBeans perform several quick validation steps when checking into the NetBeans CVS repository.[1] In fact, some web-based systems such as Aegis [1] will not allow a developer to commit changes unless all commit-validation tests have passed. This mechanism ensures that changes will not stop the software from "working" when they are integrated into the software baseline.

Smoke Testing: Other, slower mechanisms include "daily building and smoke testing" that execute more thorough test cases on a regular (*e.g.*, nightly) basis at central server sites. Daily builds (also called nightly builds) and smoke tests [26, 29, 48] have become widespread [21, 52]. They have been used for a number of large-scale commercial and open-source projects. For example, Microsoft used daily builds extensively for the development of its Windows NT operating system [29]. The GNU project continues to use daily builds for most of its projects. For example, during the development of the

[1] `http://www.netbeans.org/community/guidelines/commit.html`

Ghostscript software, daily builds were used widely. During daily 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" [28]). Typically *unit tests* [52] and sometimes *acceptance tests* [13] are executed during smoke testing. Such tests are run to (re)validate the basic functionality of the system [28]. Smoke tests exercise the entire system; they don't have to be an exhaustive test suite but they should be capable of raising a "something is wrong here" alarm. A build that passes the smoke test is considered to be "a good build." As is the case with all testing techniques, it is quite possible that problems are found in a good build during more comprehensive testing later or after the software has been fielded. In smoke testing, developers do not get instant feedback; rather they may be e-mailed the results of the daily builds and smoke tests.

Continuous Testing: Another, still higher level of continuous QA support is provided by mechanisms such as Skoll [31] that continuously run test cases, for days and even weeks on several builds (stable and beta) of the evolving software using user-contributed resources over the Internet. For example, the ACE+TAO is tested continuously by Skoll; results are summarized in a web-based virtual scoreboard.[2] All these mechanisms are useful, in that they leverage multiple resources to detect defects early during software evolution. Moreover, since feedback is directed towards specific developers (*e.g.*, those who made the latest modifications), QA is implicitly and efficiently distributed. However, none of these mechanisms has been used for GUI testing.

[2] `http://www.dre.vanderbilt.edu/scoreboard/`

2.7 Fault Seeding

Fault seeding is a well-known technique used to introduce known faults into programs [23, 47]. During fault seeding, classes of known faults are identified, and several instances of each fault class are artificially introduced into the subject program code at relevant points to create fault-seeded versions. Test cases are then generated and executed simultaneously on the fault-seeded versions and the original subject application. A test case fails if there is a mismatch between the original software's GUI state and the fault-seeded version's GUI state.

Note that using fault seeding is a popular way to simulate the process of fault detection by a test case. In a real testing scenario, a tester creates a test case together with a description of an "expected outcome" for the software. A software that does not execute as expected *fails* on the test case; otherwise it *passes*. By using a "golden version" of the software and fault-seeded versions, the creation of descriptions of "expected outcomes" is side-stepped for each test case. A fault-seeded version that behaves exactly like the golden version on an input is observationally equivalent to the original software; hence the input (*i.e.*, the test case) has been unable to "reveal the fault" that was seeded in the code to create the fault-seeded version. The advantages and disadvantages of using fault -seeding for this type of study are well-known [23, 47]. Note that researchers have shown that artificial faults are good representatives of actual software faults [6].

2.8 Summary

This chapter presented an overview of existing techniques, some of which serve as the foundation for the concepts developed in this dissertation. In particular, the EFG concept is extended to construct event-interaction graphs (EIG), which serve as the basis for "crash" and "smoke" test cases. The definition of GUI's state in terms of widgets, properties, and values, and the idea of reference testing are adapted to create efficient test oracles for the smoke testing process. Ideas from rapid feedback-based QA mechanisms are used throughout the continuous GUI testing process.

Chapter 3

A Continuous GUI Testing Process

Figure 1.1 presented a high-level overview of the continuous GUI testing process developed in this dissertation. This chapter provides additional details, presents criteria for each loop, and describes the steps/activities developed to realize each testing loop.

3.1 Innermost Loop

The innermost loop is the most frequently executing process shown in Figure 1.1. The goal of this loop is to create test cases that can quickly test major parts of the GUI automatically without any human intervention within a predetermined time interval. More specifically, the loop should use techniques to generate and execute test cases and oracles that satisfy the following criteria.

- The test cases should be generated quickly on-the-fly and executed. The test cases are not saved as a suite; rather, a throwaway set of test cases that require no maintenance is obtained.
- The test cases should broadly cover the GUI's entire functionality.
- It is expected that new changes should be made to the GUI before the testing process is complete. Hence, the process should be terminated and restarted each time a new change is made. The test cases should detect major problems in the fixed time

interval.

- As the GUI code may be changed by another developer before all the tests have been executed, the process should ensure that all tests that cover the entire GUI are executed over a series of code changes.

- The test oracle should be automated.

3.2 Intermediate Loop

The intermediate loop (shown in Figure 1.1) that executes on a daily basis is more complex than the innermost loop and requires additional effort on the part of the test designer. It also executes for a longer period of time. Moreover, the goal of this loop is to determine whether the software "broke" during its latest modifications. More specifically, the techniques used in this loop should satisfy the following criteria.

- The test cases should be generated and executed quickly, *i.e.*, in one night.

- The test cases should provide adequate coverage of the GUI's functionality. The goal is to raise a "something is wrong here" alarm by checking that GUI events and event-interactions execute correctly.

- As the GUI is modified, many of the test cases should remain usable. Earlier work showed that GUI test cases are very sensitive to GUI changes [32]. The goal here is to design test cases that are robust, in that a majority of them remain unaffected by changes to the GUI.

- The test suite should be divisible into parts that can be run (in parallel) on different machines.
- The test oracle should be automated.

3.3 Outermost Loop

The outermost loop is the most expensive, and hence the least frequently executed testing loop during GUI evolution. This loop largely depends on the resources that the software companies/organizations are willing to spend, and the expertise of the developers/testers. Moreover, different companies/organizations may face different requirements and budget spending plans. Recognizing these constraints for this loop, instead of developing new testing techniques, this dissertation will provide a set of guidelines that may be used with all existing techniques to assist during the execution of this loop.

Because GUI development is iterative, valuable resources may be conserved by employing a model-based approach for this loop.

3.4 Instantiating the Loops

Due to some of the criteria of the innermost loop (full automation), this dissertation develops a new testing technique called crash testing that can be executed by the loop. Crash testing is essentially a two-stage code commit with an automated GUI testing intervention step. A developer who has made a change to a part of the GUI code "checks-in" the changes. An instance of the crash testing process is automatically launched at the server that hosts the code repository (in general, this could be a dedicated computer that

is linked to the repository server). A reverse engineering technique [35] is used to automatically obtain a model of the GUI. This model is used to generate crash test cases, which are then automatically executed on the newly modified GUI. "Software crashes" are reported back to the specific developer who checked-in the changes along with the test cases that caused the crash. The developer debugs the GUI and resubmits the changes. Only the previously failed test cases are re-executed; if they pass, the code changes are made permanent in the repository. If they fail, new crash information is reported to the developer and the previous steps are repeated. Note that this process does not require any manual intervention; it is fast and gives a very specific type of feedback to the developer involved, *i.e.*, whether the software crashed or not.

Crash test criteria include full coverage of an abstract model of the GUI called the event-interaction graph; the test oracle detects only crashes. Details of the crash testing process are presented in Chapter 4. An empirical study presented therein shows that the crash testing process is efficient in that it can be performed automatically, and useful, in that it helps to detect crashes. The feedback from crash testing is quickly provided to the specific developer who checked in the latest GUI changes. The developer can debug the code and resubmit the changes before the problems effect other developers' productivity.

Similarly, due to some of the criteria of the intermediate loop, a form of reference testing, called smoke testing is executed every night. Smoke testing is launched to ensure that changes made to the GUI during a 24 hour period (this interval length is tunable) are integrated properly. The smoke testing process is launched automatically; it employs a reverse engineering technique (similar to the one used for crash testing) to obtain a GUI model, which is used for test case generation. The previous version is used as a *test oracle*

(a mechanism that determines whether a software being tested is executing correctly). As test cases are executed automatically on the latest GUI version, its state after each event is compared to the baseline and mismatches are reported. Although the process described thus far is fully automatic, the mismatches (that are reported to all developers involved in the latest changes) need to be examined manually to weed out false positives. False positives are expected to exist because the software has been modified, leading to expected changes between the new and baseline version.

Smoke test criteria also include full coverage of the event-interaction graph model; in addition, it requires that differences between the latest and previous GUIs be reported. Experiments involving smoke testing are provided in Chapter 5. Results of these experiments show that smoke testing is effective at detecting a large number of GUI faults. Testers have to examine the test results and manually eliminate false positives, which may arise due to changes made to the GUI. The combination of smoke and crash testing ensures that "crash bugs" will not be transmitted to the smoke testing loop. If not weeded out earlier, such bugs lead to a large number of failed and unexecuted test cases, causing substantial delays.

The outermost loop uses a collection techniques that are referred to as comprehensive GUI testing. In general, the goal of comprehensive GUI testing is to develop and execute a "thorough" test suite that looks for errors beyond crashes and differences between the latest and previous versions. In Chapter 6, tradeoffs between test case length, test suite size and event composition are studied with the goal of designing a "good" comprehensive test suite. The event-driven nature of GUI software is exploited to determine positions in test cases where test developers can insert assertions (for the test oracle) and

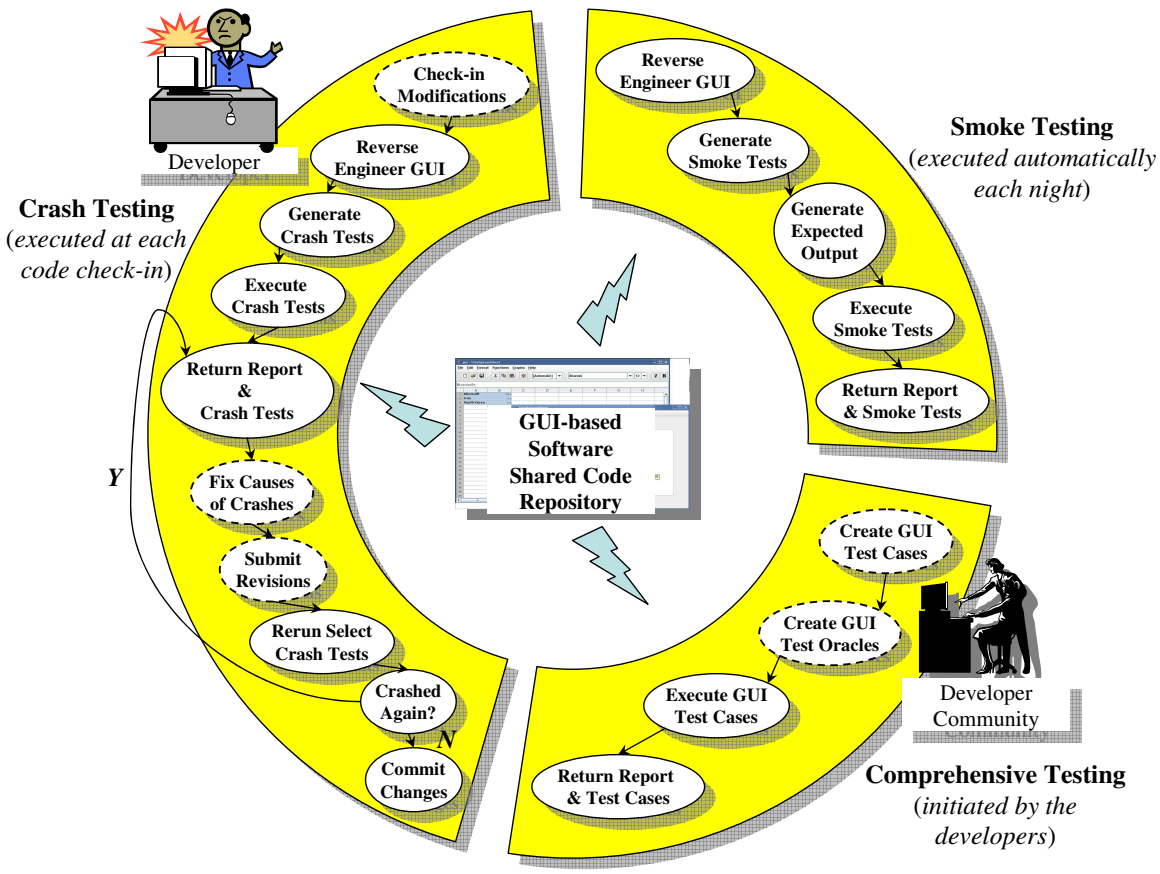


Figure 3.1: Activities to Support Continuous GUI Testing

get maximum fault-detection effectiveness.

The activities described above to realize each testing loop are summarized in Figure 3.1. The dashed ovals are the activities that are performed by the developer, and are hence resource intensive. Other activities are performed by automated tools developed in this dissertation.

3.5 Summary

This chapter described details of the criteria and processes needed to support the continuous GUI testing process. The process consists of three concentric loops. Techniques specific to each loop, crash testing, smoke testing, and comprehensive testing were described. Crash testing operates fully automatically to detect crashes in the newly checked-in updates; smoke testing tests the current GUI version against its previous version on a daily basis; comprehensive testing executes a "thorough" test suite to look for errors beyond crashes and differences between latest and previous versions. Details of each process are presented in subsequent chapters.

Chapter 4

Crash Testing

Crash testing is the most frequently executed process during continuous GUI testing. The GUI is tested automatically every time a code change is made, The goal is not to exhaustively test the GUI; rather, it is to quickly test the software for crashes by checking that each GUI event and interactions between them work correctly. As the code may be changed by another developer before all the crash tests have been executed, hence requiring restarting of the process, a simple rotation-based scheme (described in Section 4.6) is used to ensure that the entire GUI is tested over a series of code changes.

The criteria for crash testing (discussed in Chapter 3) present a number of challenges. One significant challenge is to find a small number of test cases (event interactions) that cover the entire GUI, can be executed very quickly, yet are effective at detecting faults in the software. Another challenge is to develop an automatic test oracle that detects crashes without human intervention. The first challenge was handled by developing a new model of the GUI called an event-interaction graph (EIG). This model was obtained by empirically studying GUI faults and interactions between GUI events that lead to faults; EIGs were then used for automated test case generation. The second challenge was handled by developing a mechanism to determine whether the GUI software crashed during the execution of a test case.

The remainder of this chapter presents a new concept called the minimized effective

event context, details of a pilot study used for the development of the EIG model, and application of the model to crash testing and evaluation on eight GUI-based applications.

4.1 Minimized Effective Event Context

The overall goal of developing the EIG model is to use it to generate potentially problematic event sequences, *i.e.*, test cases that reveal faults. This section takes the first step to obtaining such a model by introducing a new concept called the minimized effective event context (MEEC) of an event X. Intuitively, the MEEC of X is the *shortest* (in terms of number of events) event sequence that needs to be executed before X detects a GUI fault in a failed test case. Subsequent sections will empirically study the structure of the MEEC and use it to create the new model called an EIG.

Given a test case $< S_0, e_1; e_2; \ldots; e_n >$ (defined in Section 2.1.2) that has failed at event e_i, *i.e.*, the expected and actual states mismatched immediately after e_i was executed, not all of e_i's preceding events $e_1; \ldots; e_{i-1}$ in the test case would contribute to the failure, suggesting that some of these events may be removed. Because a failure would lead to the debugging process that would ultimately cause the fault (F), *i.e.*, the reason for the failure, to be fixed, a failed test case will be referred to as having "detected the fault". In general, not all events may be removed since they are necessary to establish the context in which e_i detected the fault. Hence a subsequence $e_j; \ldots; e_k$ (for $1 \leq j \leq (n-1)$ and $(j+1) \leq k \leq (n-1)$) of $e_1; \ldots; e_n$) is sufficient for e_i to detect the fault. The resulting test case would be $< S_0, e_j; \ldots; e_k; e_i >$. Care must be taken that event e_j can be executed in the test case's starting state S_0, and e_i can be executed in the state S_k resulting

from the execution of e_k. The *effective event context* (EEC) of an event in terms of a test case and a fault F detected by event e_i is formally defined as:

- **Definition:** Given a test case $< S_0, e_1; e_2; \ldots; e_n >$, the *EEC* of event $e_i \in \{e_1; e_2; \ldots; e_n\}$ that has detected a fault F is the pair $(S_0, e_j; \ldots; e_k)$, for $1 \leq j \leq (n-1)$ and $(j+1) \leq k \leq (n-1)$ of $e_1; \ldots; e_n$ such that e_j can be executed in S_0 and e_i can be executed in S_k. □

Note that an event may have multiple EECs. The MEEC is defined as the shortest EEC needed to detect the fault.

- **Definition:** Given a test case $< S_0, e_1; e_2; \ldots; e_n >$ and a fault F that was detected by an event $e_i \in \{e_1; e_2; \ldots; e_n\}$, the *MEEC* of e_i is the shortest EEC to detect the fault F. □

It is difficult to guess the structure and length of MEECs for typical GUI test cases and faults. In this chapter, a bottom-up approach is used to understand MEECs. In particular, a pilot study of real failed GUI test cases is conducted on several GUI applications, the MEEC for each test case is extracted, and the characteristics of MEECs are used to create EIGs. The study is described next.

4.2 Pilot Study - Understanding the MEEC

In this study, the following questions need to be answered:

1. How many event sequences is a user allowed to execute in a typical GUI-based software application?

2. Do GUI events interact? Is it sufficient to test each event once?
3. When a GUI test case (*i.e.*, consisting of a sequence of events $e_1, e_2; e_3; \ldots; e_n$) reveals a fault at event e_i,[1] what is the role of the context established by preceding events $e_1; e_2; e_3; \ldots; e_{i-1}$? Which of the preceding events are actually needed for fault detection?
4. What is the structure of the MEEC?

4.2.1 Study Procedure

The above questions are answered using the following process.

Step 1: Take different GUI-based software subjects.

Step 2: Artificially seed faults in them.

Step 3: Generate test cases; each test case is of the form $< S_0, e_1; e_2; \ldots; e_n >$, where S_0 is the initial state of the GUI in which the event sequence $e_1; e_2; \ldots; e_n$ is executed.

Step 4: Execute each test case on each fault-seeded version. A test case fails if there is a mismatch between the fault-seeded version's GUI state and the original software's GUI state. Record the event at which the mismatch was observed.

Step 5: For each test case $< S_0, e_1; e_2; \ldots; e_n >$ that failed at event e_i, $1 \leq i \leq n$, compute the shortest subsequence $e_l; \ldots; e_k$, for $k < i$, such that e_i still fails on the same fault-seeded version.

[1] Recall that the GUI test case is executed one event at a time; a fault may be detected during the execution of one of the events.

4.2.2 Step 1: Study Subjects

Several requirements had to be satisfied when selecting the subject applications. First, access to the source code, CVS development history, and bug reports (for oracle creation, described later) was needed. Second, the applications needed to be "GUI-intensive," *i.e.*, ones without complex back-end code. The GUIs of such applications are typically static, *i.e.*, not generated dynamically from back-end data. Finally, the applications needed to be non-trivial, consisting of several windows and widgets.

The study subjects 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[2] and includes TerpWord (a word-processor with drawing capability), TerpCalc (a scientific calculator with graphing capability), TerpPaint (an imaging tool), TerpPresent (a presentation tool), and TerpSpreadSheet (a compact spreadsheet program). They have been implemented using Java. Table 4.1 summarizes the characteristics of these applications. Note that these applications are fairly large with complex GUIs. With the exception of TerpCalc, all the applications are roughly the size of MS WordPad. The number of widgets listed in the table are the ones on which user input events can be executed (*i.e.*, text-labels are not included). The LOC are the number of statements in the programs. The `Help` menu is also not included since the help application is launched in a separate web browser. Most of the code written for the implementation of each application is for the GUI. None of the applications have complex underlying "business logic". This property of the subject

[2] `http://www.cs.umd.edu/users/atif/TerpOffice`

Subject Application	Windows	Widgets	LOC	Classes	Methods	Branches
TerpWord	**11**	**126**	**4893**	**104**	**236**	**452**
TerpSpreadSheet	**9**	**159**	**12791**	**125**	**579**	**1521**
TerpPaint	**10**	**215**	**18376**	**219**	**644**	**1277**
TerpCalc	**1**	**85**	**9916**	**141**	**446**	**1306**
TerpPresent	**12**	**328**	**44591**	**230**	**1644**	**3099**
TOTAL	**43**	**913**	**90567**	**819**	**3549**	**7655**

Table 4.1: TerpOffice Applications

applications is especially important for seeding GUI faults (discussed later) since almost the entire code is for the GUI; there is no need to distinguish between GUI-code and business-logic-code during fault seeding.

4.2.3 Step 2: Fault Seeding

Several issues need to be addressed when creating the fault-seeded versions. First, care is taken so that the artificially seeded faults are similar to faults that occur in programs due to mistakes made by developers. As defined earlier, a *GUI fault* is defined as one that manifests itself on the visible GUI at some point of time during the software's execution. A history-based approach was adopted to seed GUI faults, *i.e.*, "real" GUI faults were observed and used from real applications. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[3] 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. The classes of faults are summarized next in one short statement; the example of each class is provided in Table 4.2. Note that the row

[3] `http://bugs.cs.umd.edu`

number in the table corresponds to the numbering below.

1. Modify relational operator (>, <, >=, <=, ==, !=);
2. Invert the condition statement;
3. Modify arithmetic operator (+, -, *, /, =, ++, –, +=, -=, *=, /=);
4. Modify logical operator (&&, ||);
5. Set/return different boolean value (`true`, `false`);
6. Invoke different (syntactically similar) method;
7. Set/return different attributes;
8. Modify bit operator (&, |, ∧, &=, !=, ∧=);
9. Set/return different variable name;
10. Set/return different integer value;
11. Exchange two parameters in a method;
12. Set/return different string value.

Fault Type	Original Code	Mutated Code
1	if (this.row > y.row)	if (this.row **<** y.row)
2	if (newValue)	if (**!**newValue)
3	prev = index+1;	prev = index**-**1;
4	if (done || border == null ||	if (done **&&** border == null ||
5	if(contentArea.closeDocument(true))	if(contentArea.closeDocument(**false**))
6	int rowLimit = model.getRowCount() - 1;	int rowLimit = model.**getColumnCount()** - 1;
7	int style = Font.ITALIC;	int style = Font.**BOLD**;
8	style |= Font.BOLD;	style **&=** Font.BOLD;
9	buttonPanel.add(okButton);	buttonPanel.add(**cancelButton**);
10	int size = 12;	int size = **15**;
11	tmp = data.substring(0, i2);	tmp = data.substring(**i2,0**);
12	if(findString.equals("")) { return; }	if(findString.equals(**" "**)) { return; }

Table 4.2: Classes of Seeded Faults

Second is to determine *where* the faults will be seeded in the code. The code of the applications was partitioned by functionality into *functional units*. The functional units

for each subject application are shown in Column 1 of Table 4.3. The names indicate the roles of the functional units. The total number ($N_{i,j}$) of opportunities for seeding each type of fault (j) in each functional unit (i) was then counted. The number of faults seeded in each functional unit was proportional to the opportunities. The faults were seeded at *equal distances* across the functional units, *i.e.*, if the number of opportunities was N and the number of seeded faults was M, then the faults were seeded approximately at every (N/M) opportunity. Table 4.3 shows the number of seeded faults per functional unit.

Fun. Units	Number of Faults Seeded				
	TerpPaint	TerpWord	TerpCalc	TerpSpreadSheet	Total
File Operation	32	35	2	14	83
Business Logic	36	52	61	85	234
Search/Find Function	0	22	0	4	26
Clipboard Operation	1	7	9	29	46
Preference Setting	131	83	128	65	407
OK/Cancel Dialogs	0	1	0	3	4
TOTAL	200	200	200	200	800

Table 4.3: Seeded Faults Classified by Functionality

Finally one common issue with GUI fault-seeding is that some faults will never be manifested as failures on the GUI. Hence, a large enough number of faults is seeded so that useful results are obtained even if some of them are not manifested. A total of 200 faults were seeded in each application. Only one fault was introduced in each version. This model is useful to avoid fault-interaction. Four graduate students seeded the faults independently. These students had taken a graduate course in software testing and were familiar with popular testing techniques.

4.2.4 Step 3: Test-Case Generation

To address Question 1, this step computes the total number (by length) of event sequences that may be executed on the subject applications. The results are summarized in Figure 4.1. The x-axis shows the length of the event sequence; the y-axis (logarithmic scale) shows the number of sequences. This result shows that, for the subject applications, the number of event sequences grow exponentially with length. It would be extremely expensive to generate and execute all event sequences beyond $length > 3$.

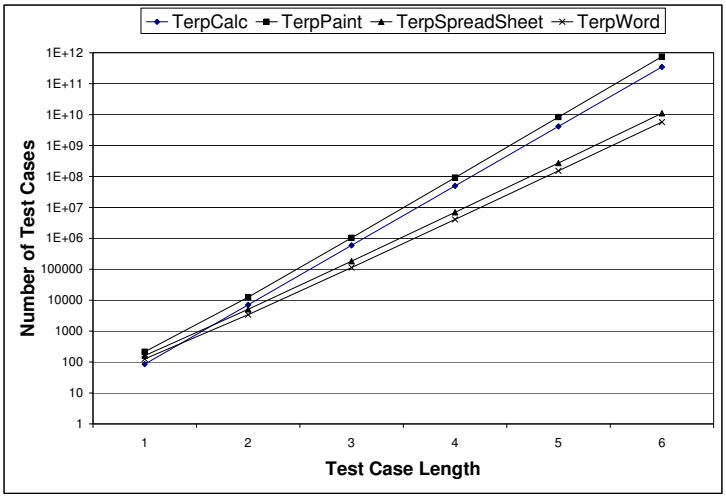


Figure 4.1: Total Number of Event Sequences

As the total number of potential event sequences (and hence the number of test cases) is enormous, in this study, a reasonable subset was generated. Any process that is used to select the subset may have an impact on the results of the study. Hence, to minimize threats to external validity, a process that GUI testers commonly use in practice, *i.e.*, to generate test cases that cover each event at least once, was chosen.

An existing EFG-based approach was used to generate test cases for all applications. In summary, for each subject program, the EFG is traversed from one event that can be

executed in an initial GUI state S_0; a list of events that may be executed in S_0 are created, and one event is chosen. The event sequence becomes iteratively longer by selecting another event using the `follows` relationship encoded in the EFG's edges. Whenever possible, events that had not already been used were selected.

The above algorithm is able to generate a large number of long test cases that contain all the events in the software. All these test cases were executed on the fault-seeded versions, storing only those that failed, and discarding the rest. As expected, not all these test cases failed. In all, 1119 test cases failed. The longest of these test case had 50 events and the shortest one had 1 event.

Figure 4.2 shows the event distribution of all the test cases. The figure shows four column graphs; the x-axis shows all the events in each application; the y-axis shows the number of times a particular event was executed by a test case. It is found that the test cases had good event coverage.

Step 4: Test Execution

A test executor was designed to executed the entire test suite automatically on the subject applications and all the fault-seeded versions. It performed all the events in each test case and compared the fault-seeded version's GUI state with the original software's GUI state. Events were triggered on the GUI using the native OS API. The test cases executed on four machines (Pentium 4, 2.2GHz, each with 256MB RAM) simultaneously for more than a week. Although much of the execution was automated, some machines (and test scripts) had to be restarted because of problems with the Java virtual machine

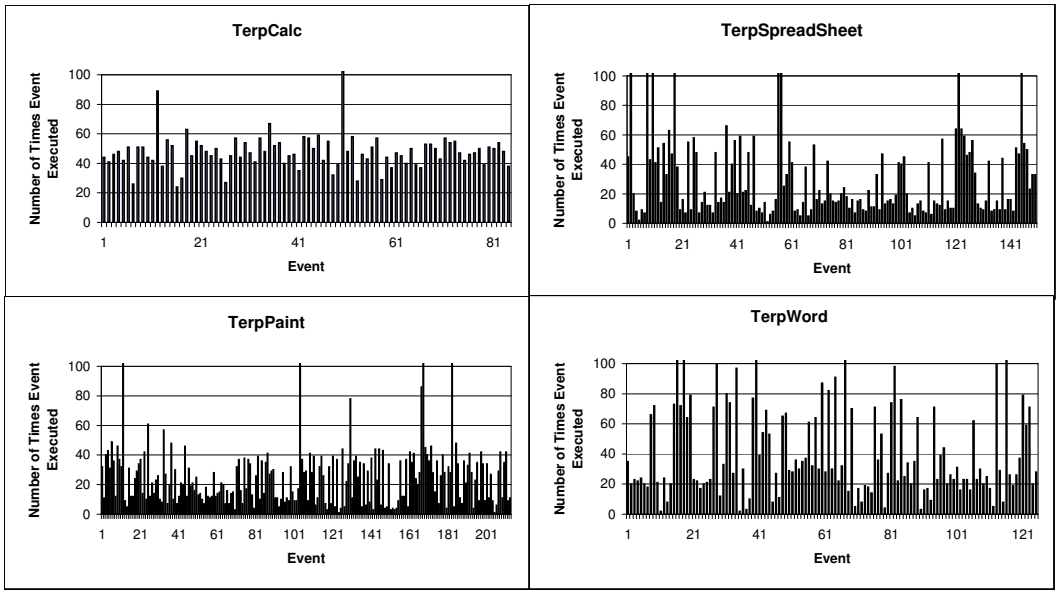


Figure 4.2: Event Distribution

(JVM).

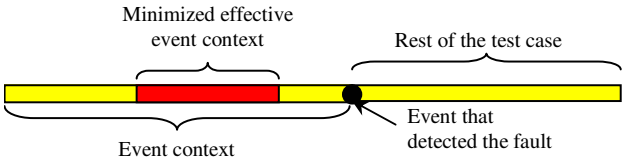
It was observed that there were many complex interactions between events in the subject applications. If a test case failed at event e_x for fault-seeded version M_y, e_x did not cause a failure for the same fault-seeded version M_y in many other test cases. Moreover, if e_x occurred in test case T_i multiple number of times, it caused test case failure for M_y at only one point. This observation showed that the context of an event seriously affected its ability to reveal a fault. This data is summarized into 4 plots shown in Figure 4.3. The x-axis in these plots represents individual events in the GUI. The dotted line shows the number of times a particular event existed in some failed test case. The solid line shows the number of times the event caused the failure. While many events caused the failure for one or more fault-seeded versions, the same event failed to cause the failure in many instances. For example, event #120 in TerpWord caused 300 test

cases to fail; however, the same event, although it existed in 600 other test cases, did not cause a failure. Also, note that results of failed test cases for at least one fault-seeded version were presented; there were many test cases that did not cause test case failure for even a single fault-seeded version but comprised of events that were otherwise successful in other contexts at causing failures. Although the above discussion reinforces popular belief that the software state plays an important role during testing, it also shows that it is important to consider state when generating test cases.

4.2.5 Step 5: Studying Predecessor Events

For a failed test case $< S_0, e_1; e_2; \ldots; e_n >$, in which the event e_x (for $1 \leq x \leq n$) caused the failure for fault-seeded version $\mathcal{M}$, all possible subsequences of $e_1; \ldots; e_{x-1}$ were created keeping only those in which the first event can be executed in S_0 and e_x `followed` the last event. Test cases were then obtained by appending e_x to the chosen subsequences. Starting from the shortest of these test cases, they were executed on the same fault-seeded version on which the original test case failed, stopping when one failed. The predecessor events in this (shortest) test case form the MEEC.

Each test case is shown as a horizontal line with 2 levels of shading. The dark band shows the MEEC.



The above shaded-horizontal-line visualization is stacked for all test cases per ap-

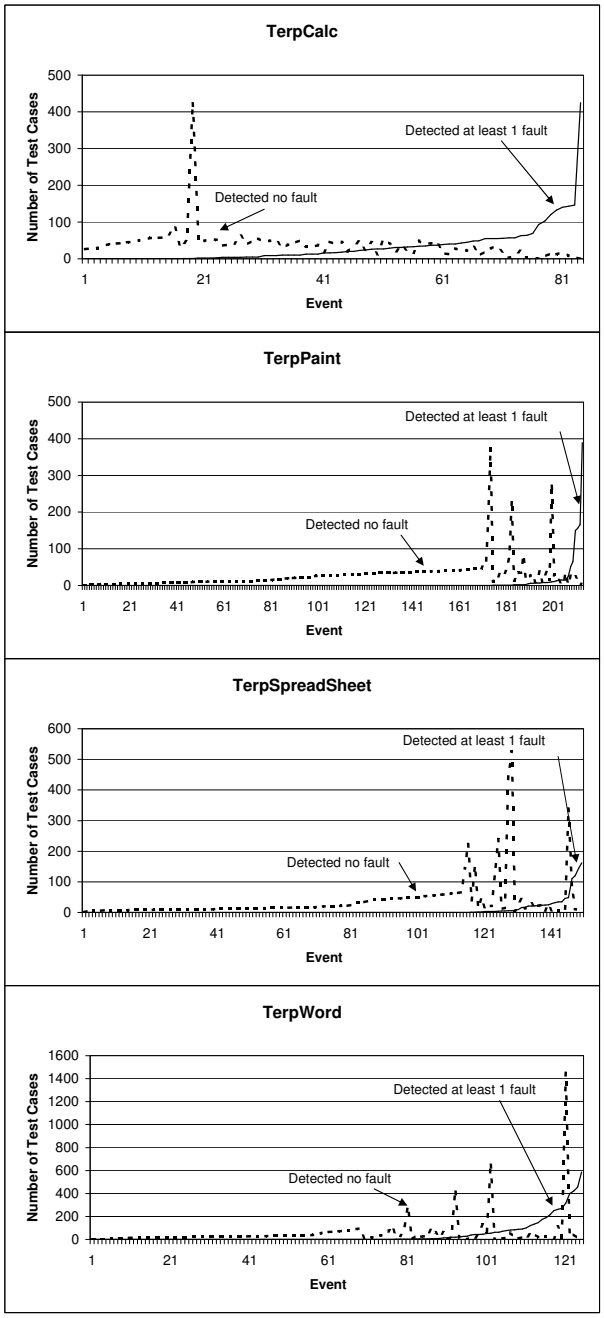


Figure 4.3: Events Interactions

plication to summarize the results. Figure 4.4 shows the results for TerpCalc. The x-axis shows the event number in the test case. The y-axis represents failed test cases. If a test case failed for two fault-seeded versions, then it is counted twice, since it may result in a different MEEC. The result for TerpCalc shows that the average length of the MEEC for TerpCalc was 2.21 events. Even though the entire test case was long (50 events in many cases), large parts of the test case were in fact useless for fault detection. If all the events were ignored except those in the MEEC, all the faults are still able to be detected. Figures 4.5, 4.6, and 4.7 show the same results for TerpPaint, TerpSpreadSheet, and TerpWord respectively. The average length of the MEEC was 3.57, 4.62, and 3.86 respectively.

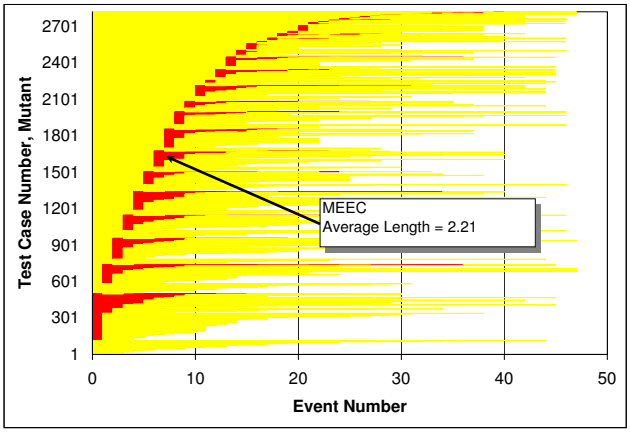


Figure 4.4: MEEC for TerpCalc

4.3 Dissecting the MEEC

To further understand the structure of MEECs, a classification of GUI events is created:

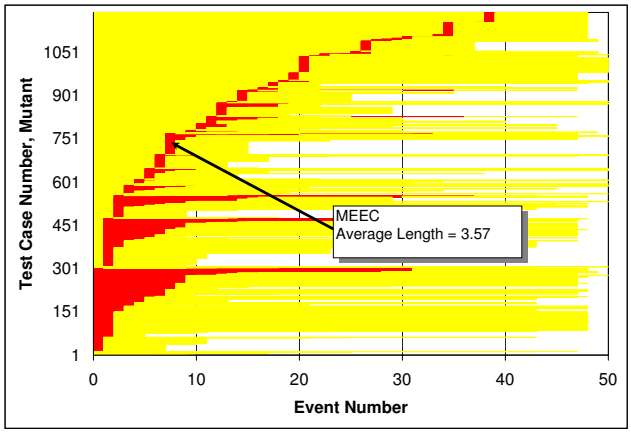


Figure 4.5: MEEC for TerpPaint

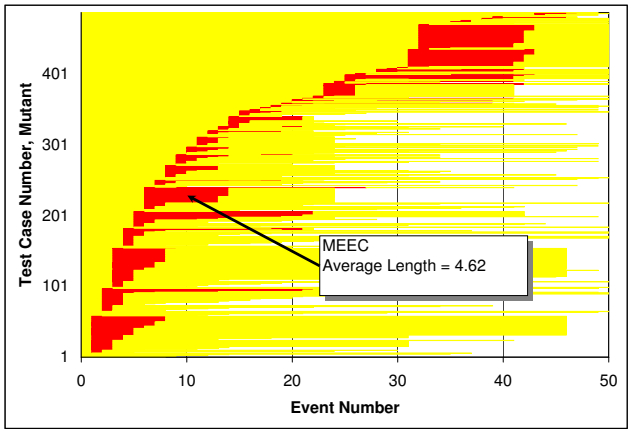


Figure 4.6: MEEC for TerpSpreadSheet

- *reachability events* (denoted by a symbol `R`) that are used to open windows, and open/close menus. One subset of `R` of interest is `W`, the set of events that open windows.
- other events that are used to manipulate the structure of the GUI include *termination events* (`T`) that close windows.
- events that do not manipulate the structure of the GUI are called *system-interaction events* (denoted by symbol `S`).

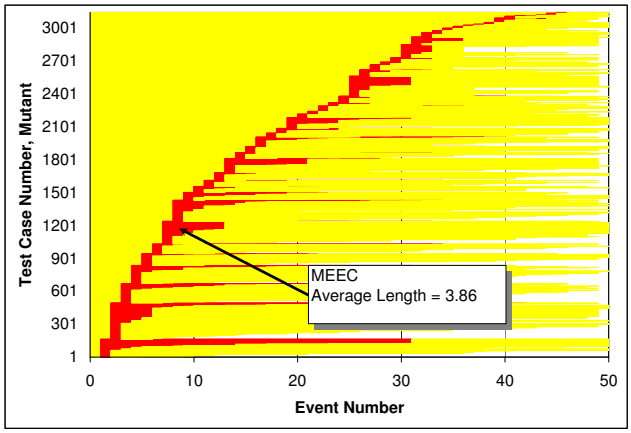


Figure 4.7: MEEC for TerpWord

The above event classes were then used to create an abstraction of the MEECs – each event was replaced by one of the symbols `S`, `T`, or `R`, depending on its function in the GUI. The resulting strings were then compactly represented using regular expressions. The result of this compaction process yielded four regular expressions R*, R*S, R*SR+, and R*SR*(SR*)+, each of which was assigned a "pattern ID" 1, 2, 3, and 4 respectively. The failed test cases were then partitioned by pattern ID. Note that the MEECs did not contain any event of class `T`.

The result of this overall process is shown in Table 4.4. Column `MEEC Structure` of this table shows the regular expression. Column e_x shows the type of event that caused the failure. Column `# Failures` show the number of failed test cases. Note that a failure is not a crash; it is a mismatch between the actual and expected state of the GUI. Each failed execution was "debugged" to determine the seeded fault that caused the failure. The number of faults is shown in Column `# Faults`. The numbers in `# Faults` are somewhat misleading because a single fault may be manifested as multiple failures. Consequently, multiple test cases may detect the same fault, causing each pattern to be

Pattern ID	MEEC Structure	e_x	# Failures	# Faults	Unique Faults	% Unique Faults
TerpCalc						
			Total Faults Detected		**96**	
			Total Test Case Failures		**1275**	
1	**R***	**S**	676	37	37	**38.5%**
		W	6	1	1	**1.0%**
2	**R*S**	**S**	431	54	40	**41.7%**
		W	1	1	0	**0.0%**
3	**R*SR+**	**S**	19	10	3	**3.1%**
4	**R*SR*(SR*)+**	**S**	142	42	15	**15.6%**
TerpPaint						
			Total Faults Detected		**23**	
			Total Test Case Failures		**115**	
1	**R***	**S**	78	14	14	**60.9%**
		W	31	6	6	**26.1%**
2	**R*S**	**S**	2	1	1	**4.3%**
		T	4	2	2	**8.7%**
TerpSpreadSheet						
			Total Faults Detected		**11**	
			Total Test Case Failures		**318**	
1	**R***	**S**	259	4	4	**36.4%**
		T	5	1	1	**9.1%**
		W	20	3	3	**27.3%**
2	**R*S**	**S**	17	3	3	**27.3%**
3	**R*SR+**	**S**	17	2	0	**0.0%**
TerpWord						
			Total Faults Detected		**33**	
			Total Test Case Failures		**544**	
1	**R***	**S**	152	9	9	**27.3%**
		T	61	4	4	**12.1%**
		W	296	15	15	**45.5%**
2	**R*S**	**T**	22	3	2	**6.1%**
3	**R*SR+**	**S**	2	1	1	**3.0%**
		T	4	1	1	**3.0%**
4	**R*SR*(SR*)+**	**S**	7	2	1	**3.0%**

Table 4.4: Regular Expression Table

counted several times; which is why the sum of all faults does not add to the `Total Faults Detected` value.

An alternative measure, shown in the column `Unique Faults`, shows the number of faults that were detected by test cases with Pattern i *but not with Pattern* $i-1$. This measure will be useful when developing new test case generation techniques based on these results. The main idea of defining this measure is that it is less expensive to generate event sequences using Pattern $i-1$ than with Pattern i.

Table 4.4 illustrates several important points. First many faults (38.5%) in TerpCalc were detected by test cases with Pattern 1, *i.e.*, zero or more events from `R` were followed

by an event in S; only one fault was detected when using an event of type W after zero or more R events. Pattern 1 was also effective in TerpPaint (87%), TerpSpreadSheet (72.7%), and TerpWord (84.8%); event types W and T for e_x played more significant roles in these applications. Second, Pattern 2 was very effective for TerpCalc (41.7%) when e_x was an S type of event; the same pattern was less effective for other applications. Third, Patterns 3 and 4 were not very effective in any application, except TerpCalc.

This analysis showed that parts, with well-defined structures, of test cases are in fact sufficient for fault detection in GUIs. The results of this analysis also gives insights into how to generate test cases for GUIs. For the classes of faults used in this study, the set of subject applications, and the test cases that were generated:

- a large number of faults are detected with test cases that execute a number of structural events (*i.e.*, R*), followed by either a window opening event (W) or a system-interaction event (S); the structural events are needed to simply "reach" the event that caused the failure. According to Pattern 1, it is important to test all S, W and T events at least once. In terms of EFGs, this essentially means that each node of type S, T, and W is covered by the test suite.
- for Pattern 2, it is important to test interactions between event pairs (S, S), (S, T), and (S, W). In terms of EFGs, this means that the test suite should cover such edges.
- for Pattern 3, it is important to test specific types of paths between two S events, and S and T events. These paths should only contain R types of events.
- for Pattern 4, it is important to test paths between multiple S events, where each path contains only R type of events.

Because Patterns 3 and 4 require the computation of paths between pairs of events, which can be computationally expensive, a new structure (EIG) that models these paths is developed. The next section formally describes an EIG and outlines a method to transform an EFG to EIG.

4.4 Threats to Validity

The results of the previous study, should be interpreted keeping in mind the following threats to validity.

Threats to external validity are conditions that limit the ability to generalize the results of studies to industrial practice. Several such threats are identified in this study. First, four GUI-based Java applications have been used as subject programs. Although they have different types of GUIs, this does not reflect a wide spectrum of possible GUIs that are available today. Moreover, the applications are extremely GUI-intensive, *i.e.*, most of the code is written for the GUI. The results will be different for applications that have complex underlying business logic and a fairly simple GUI. Second, all the subject programs were developed in Java by students, which might be more bug-prone than software implemented by professionals. Finally, although the 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 study without the researcher's knowledge. Every effort was made to seed faults that were as close as possible to naturally occurring faults. A history-based approach was

used for seeding faults in the GUI applications. This may have affected the detection of faults by the test cases. Faults that are not manifested on the GUI will go undetected.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this study one of the measures of cost is time. Since GUI programs interact with the windowing system's manager, the execution time of an event varies from one run to another. One way to minimize the effect of such variations is to run the studies multiple number of times and report average time. Since each event is executed several times (at least 80 times in different test cases), this threat has been adequately handled.

4.5 Event-Interaction Graph

Several terms are first defined to develop event-interaction graphs (EIG) for a GUI. An event-flow path represents a sequence of events that can be executed on the GUI. Formally, an event-flow-path is defined as follows:

- **Definition:** There is an *event-flow-path* from node n_x to node n_y iff there exists a (possibly empty) sequence of nodes $n_j; n_{j+1}; n_{j+2}; \ldots; n_{j+k}$ all in the event-flow graph E such that $\{(n_x, n_j), (n_{j+k}, n_y)\} \subseteq edges(E)$ and $\{(n_{j+i}, n_{j+i+1}) for\ 0 \leq i \leq (k-1)\} \subseteq edges(E)$. □

The function $edges$ takes an EFG as an input and returns a set of ordered-pairs, each representing an edge in the EFG. The notation $< n_1; n_2; \ldots; n_k >$ is used for an event-flow path. Several examples of event-flow paths from the EFG of Figure 2.2 are: $< File; Edit; Undo >, < File; MatchCase; Cancel >, < MatchCase; Editbox\_1; Replace >,$

and $< MatchCase; FindNext; Replace >$. Only those event-flow-paths that start and end with system-interaction and termination events, without any intermediate system-interaction and termination events are studied.

- **Definition:** An event-flow-path $< n_1; n_2; \ldots; n_k >$ is *interaction-free* iff none of $n_2, \ldots, n_{k-1}$ represent system-interaction or termination events. □

Of the examples of event-flow paths presented above, $< File; Edit; Undo >$ is interaction-free (since $Edit$ is neither a system-interaction event nor a termination event) whereas $< MatchCase; Editbox\_1; Replace >$ is not (since $Editbox\_1$, an event used to edit one of the text boxes in the Replace window, is a system-interaction event).

The *interacts-with* relationship between system-interaction events is that, two system-interaction events may interact if a GUI user may execute them in an event sequence without executing any other intermediate system-interaction event. The same holds true for termination events.

- **Definition:** A system-interaction (or termination) event e_x *interacts-with* system-interaction (or termination) event e_y iff there is at least one interaction-free event-flow-path from the node n_x (that represents e_x) to the node n_y (that represents e_y). □

For the EFG of Figure 2.2, the above relation holds for the following pairs of events: $\{(New, Date/Time)$, $(FindNext\_1, WordWrap)$, $(Editbox\_0, Editbox\_1)$, and $(Delete, Cancel)\}$. The interaction-free event-flow-paths for these pairs are $< New; Edit; Date/Time >$, $< FindNext\_1; Format; WordWrap >$, $< Editbox\_0; Editbox\_1 >$, and $< Delete; Cancel >$ respectively. Note that an event

may interact-with itself. For example, the event `MatchCase` interacts with itself. Also note that "e_x interacts-with e_y" does not necessarily imply that "e_y interacts-with e_x." For example, in the EFG, even though $Replace$ interacts-with $Cancel$, the event $Cancel$ does not interact-with $Replace$.

The interacts-with relationship is used to create the event-interaction graph. This graph contains nodes, one for each system-interaction and termination event in the GUI. An edge from node n_x (that represents e_x) to node n_y (that represents e_y) means that e_x interacts-with e_y. The event-interaction graph (EIG) for the EFG of Figure 2.2 is shown in Figure 4.8. All the events that are not part of the EIG have been crossed out. Note that the space of event-sequences has reduced considerably since only the system-interaction and termination event interactions are marked in this graph.

The algorithm to convert an EFG to an event-interaction graph is shown in Figure 4.9. The procedure `GenerateEIG` takes as input an EFG, represented as a set of nodes *N* and a set of edges *E*. It removes all non-system-interaction event nodes and their associated edges from the given EFG. At the termination of the procedure, the event-interaction graph is obtained, represented as a set of nodes $\mathcal{N}$ and a set of edges $\mathcal{E}$. $\mathcal{N}$ and $\mathcal{E}$ are initialized to *N* and *E* (lines 2-3). When traversing all edges of the EFG, a list of nodes start(*n*) on the edges that start from the node *n* (except itself) is obtained for all nodes. Similarly, a list of nodes end(*n*) that end with the node *n* (except itself) for all nodes (lines 4-6) is computed. For each node *n* of the EFG (line 7), all new edges (n_x, n_y) are added to $\mathcal{E}$ if there is an interaction-free path $< n_x; n; n_y >$ in the EFG (lines 8-11); start(n_x) and end(n_y) are updated to add n_y and n_x in the lists respectively if n_x and n_y are not the same node (lines 12-14). Accordingly, *n* is removed from the start and end

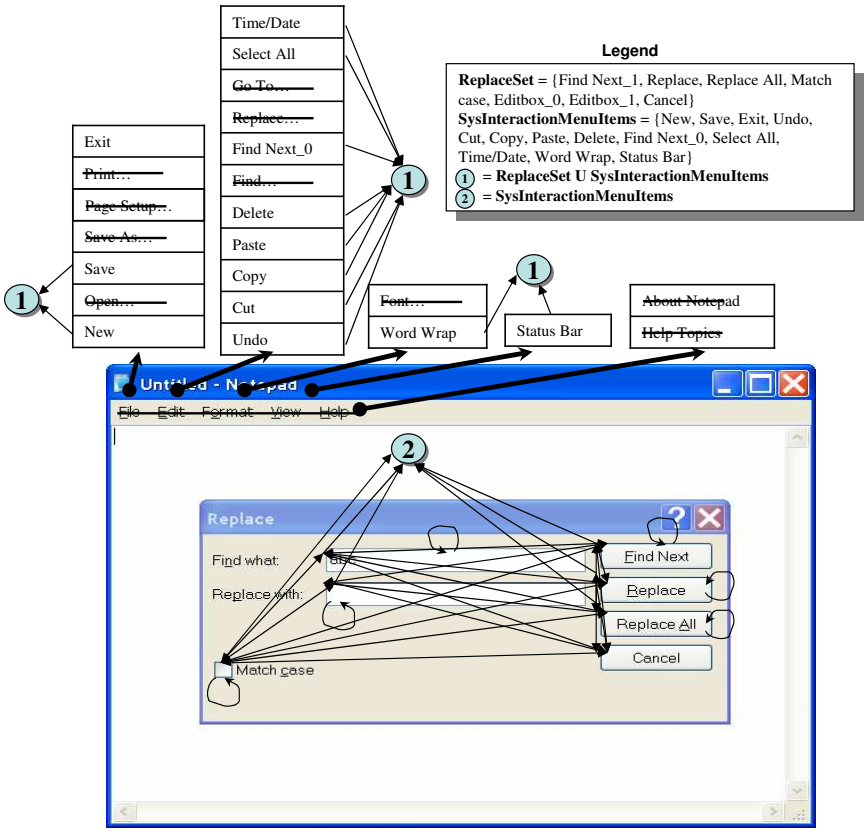


Figure 4.8: EIG for the EFG of Figure 2.2

lists (lines 15-18). Finally, n is removed from $\mathcal{N}$ (line 19); all edges associated with n are removed from $\mathcal{E}$ (lines 20-21).

The event-interaction graph may be traversed in a number of ways to generate sequences of events. For example, all length 1 event sequences may be generated by simply enumerating all the nodes in the graph. For the event-interaction graph of Figure 4.8, the set of length 1 sequences $\{New, Save, Undo, Cut, Copy, Paste, Delete, FindNext\_0, SelectAll, Time/Date, WordWrap, StatusBar, MatchCase, Editbox\_0, Editbox\_1, FindNext\_1, Replace, ReplaceAll, Cancel\}$ is obtained. All length 2 event sequences may be generated by enumerating each node with its adjacent node, *i.e.*, each edge in the EIG. For the event-interaction graph of Figure 4.8, sequences such as $< New; Undo >$,

```
N /* Nodes set of EIG */                                                  
E /* Edges set of EIG */                                                  
PROCEDURE :: GenerateEIG(                                                 
  Event Flow Graph (N, E)) {                                          1   
 N = N                                                                2   
 E = E                                                                3   

  FORALL n ∈ N DO                                                     4   
   start(n) = { n_i | (n, n_i) ∈ E, and n ≠ n_i }                     5   
   end(n) = { n_i | (n_i, n) ∈ E, and n ≠ n_i }                       6   

  FORALL n ∈ N DO                                                     7   
   IF EventType(n) ≠ system-interaction or termination                8   
    FORALL n_x ∈ end(n) DO                                            9   
     FORALL n_y ∈ start(n) DO                                        10   
      E = E ∪ (n_x, n_y)                                             11   
      IF  n_x ≠ n_y                                                  12   
       start(n_x) = start(n_x) ∪ {n_y}                               13   
       end(n_y) = end(n_y) ∪ {n_x}                                   14   
    FORALL n_x ∈ end(n) DO                                           15   
     remove n from start(n_x)                                        16   
    FORALL n_y ∈ start(n) DO                                         17   
     remove n from end(n_y)                                          18   
    remove n from N                                                  19   
    remove all edges (n, n_i) from E                                 20   
    remove all edges (n_i, n) from E                                 21   
}                                                                         
```

Figure 4.9: Generate Event-Inteaction Graph from Event-Flow Graph

$< FindNext\_1; Replace >$, and $< ReplaceAll; Exit >$ are obtained.

The remaining question is how to execute the generated event sequences. At execution time, other events needed to "reach" the events are automatically generated, yielding an executable test case. For example, the sequence $< New; Undo >$ will expand to the test case $< File; New; Edit; Undo >$ during execution.

4.6 Crash Test Cases

The test cases obtained using EIG are complete, in that they can be generated and executed automatically on the GUI. Crashes found during test execution may be used to identify problems in the software. In this research, a "crash" is defined as an abnormal termination of the software; this can be detected by the script used to execute the test cases. These test cases and "test oracle" form the crash test cases.

Once the event-interaction graph for a GUI is obtained, it can be annotated in several ways. For crash testing, a boolean flag is associated with each edge in the graph. During crash testing, once a test case that "covers" an edge is generated, the associated boolean flag is set. This prevents the same test case from being generated again, until all the edges have been covered. If the crash testing process is interrupted, *e.g.*, when a new version of the software has been checked-in, or the time interval specified for the innermost loop has been completed, the flags for each edge are retained across event-interaction graph versions.

The next section evaluates crash test cases for several programs.

4.7 Feasibility Studies - Evaluating Crash Test Cases

Two studies were conducted on the TerpOffice applications and four popular GUI-based Open Source Software (OSS) to evaluate the crash testing process. The first study demonstrates the effectiveness of the crash test cases and the usefulness of the overall process on four of the TerpOffice applications. The second study applies crash testing to several versions of four open-source applications with the goal of demonstrating that crash tests may be used to reveal problems in fielded GUI-based software; some of these problems persist across different versions of the software.

4.7.1 Feasibility Study - Crash Testing on TerpOffice Applications

This study was conducted to demonstrate the usefulness of the crash tests. More specifically, the following questions needed to be answered:

1. How long does it take to run the crash test cases?
2. How many times does a GUI software crash on the test cases?
3. How many crash-causing bugs are there in a GUI software?
4. How many more crashes do the test cases reveal that are generated using EIG than that using EFG?
5. Since the crash testing process is expected to be terminated as soon as the GUI is modified again or the time interval has been completed, which could give a very small window of time to run the test cases, how many test cases must be run to completion for effective testing?
6. When rotating test cases during frequent GUI modifications, how effective is the

annotated event-interaction graph approach?

Study Procedure

The following process was used for this study:

1. Choose software subjects with GUI front-ends.
2. Generate event-interaction graphs.
3. Generate crash test cases on-the-fly, executing each automatically on the subject applications.

The time taken to execute the test cases and the number of software crashes were reported.

Step 1: Software Subjects: The subjects are part of TerpOffice described in Section 4.2.

Step 2: Generate Event-Interaction Graphs: For each application, an EIG is generated. The sizes of the event-interaction graphs are shown in Table 4.5. As noted earlier, the crash tests will cover all nodes and all edges in the EIG. Test cases were generated by picking the two events on each edge and using a shortest-path algorithm to "reach" these events from the application's main window. In this study, more than 39K crash tests were generated and executed.

Step 3: Test-Case Generation and Execution: All the crash test cases were generated and replayed on the subject applications one-by-one. The execution consisted of performing each event, such as a clicking-on-buttons, opening-menus, selecting-items, checking-boxes, etc. If a text-box needed input, then the values were read from a database. The database was initialized with several types of inputs: negative and positive integers, text

Subject Application	Nodes	Edges
TerpWord	112	1253
TerpSpreadSheet	145	3246
TerpPaint	200	8699
TerpCalc	82	6561
TerpPresent	294	19918
TOTAL	833	39677

Table 4.5: Sizes of Event-Interactions Graph

strings, special characters, very long strings, and floating-point numbers. No customization was done for any particular text-field.

The time needed to run all the test cases is shown in Figure 4.10. It took approximately 3-6 hours to execute all the crash test cases for TerpCalc, TerpSpreadSheet and TerpPaint, and 12 hours for TerpPresent. No manual work was required. This result answers Question 1.

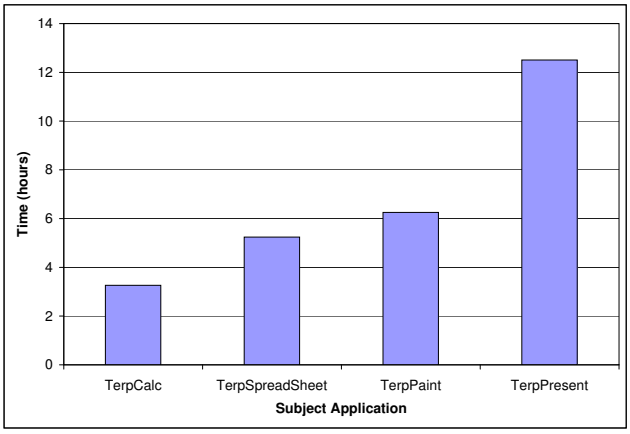


Figure 4.10: Total Execution Time

Results: Crashes Reported: Figure 4.11 shows the total number of test cases that led to a software crash. The large number of crashes reported was very encouraging, especially since this version of TerpOffice was considered to be stable, and had been tested and

debugged using a test suite of 5000+ GUI test cases; in addition, it also has at least one JUnit test case per Java method. This result answers Question 2.

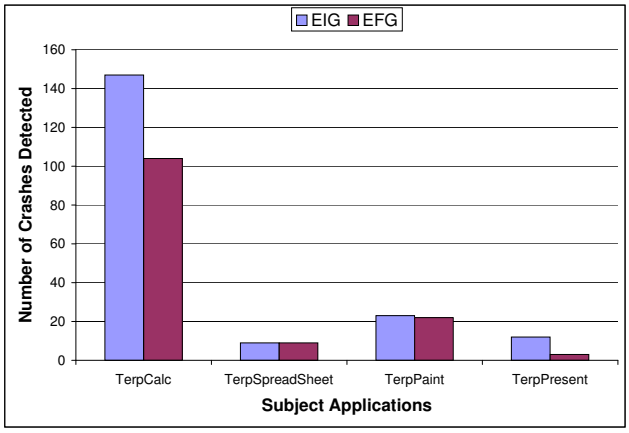


Figure 4.11: Number of Software Crashes

All the system exception messages were manually examined and the number of bugs (the term "bug" will be used to mean a "fault in the code") in the code that had led to the crashes was computed. Figure 4.12 summarizes the results. Although TerpCalc had crashed on a large number (140+) of test cases, the crashes were due to only 3 bugs in the code. Also, TerpPaint had crashed on only 23 test cases but the number of underlying bugs was 13, a surprisingly large ratio. The ratios between crashes and bugs are in fact due to the location of the bugs; if frequently executed code contains the crash-causing bug, then a large number of test cases will result in a crash. This result answers Question 3.

To address Question 4, the EFGs of each software were used to generate new test cases; nodes and edges of the EFG were covered. Test cases were generated by picking the two events on each edge and using a shortest-path algorithm to "reach" these events from the application's main window. The test oracle remained unchanged. The results

of the execution of these EFG-based test cases are shown in Figure 4.11 and Figure 4.12. The results clearly show that although EFGs are able to reveal bugs, the EIG was able to reveal additional bugs. This result answers Question 4.

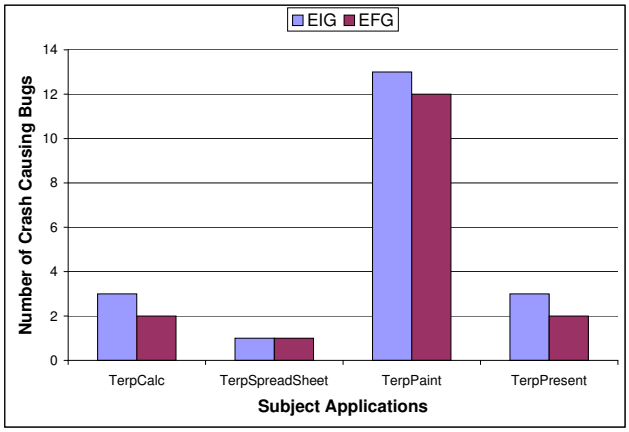
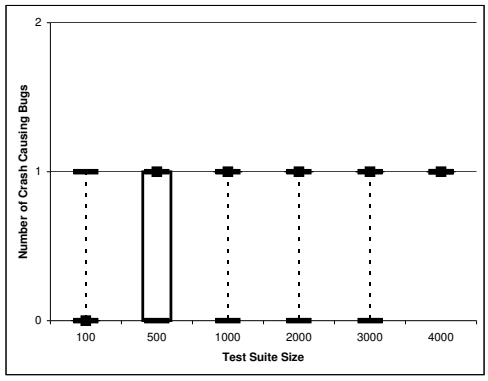


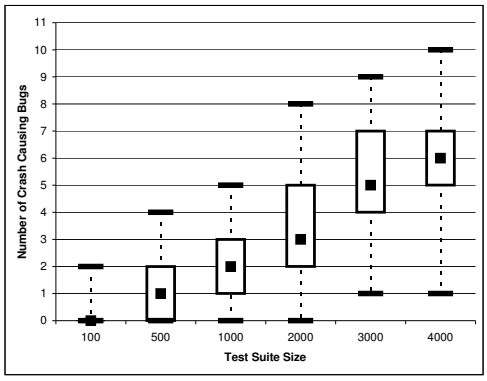
Figure 4.12: Number of Crash-Causing Bugs

Figure 4.10 showed that running all crash tests can take up to 12 hours. Since it may not be realistic to have such a long time between GUI code changes, and the time interval specified by the developer may be short, the impact of number of test cases on the number of crash-causing bugs detected was studied. Because all the test cases had already been run, it was not necessary to regenerate and re-execute new test cases; the effect of different number of test cases was simulated by treating the existing crash test suite as a *test pool* and selecting different number of test cases from them. More specifically, for each subject application, the test pool was used to create 1200 test suites: 200 of each size 100, 500, 1000, 2000, 3000, and 4000. Each suite was obtained independently using random selection without replacement.

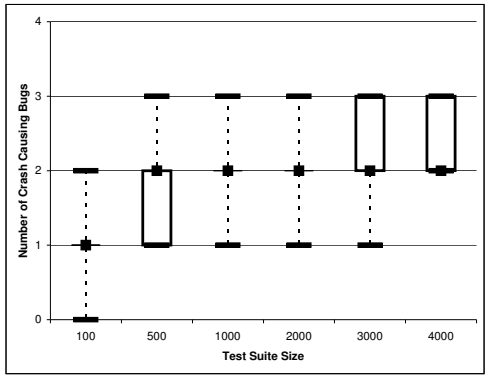
As there are 200 test suites of each size, the results are shown in the form of box-plots. The box-plots provide a concise display of each distribution. The black square



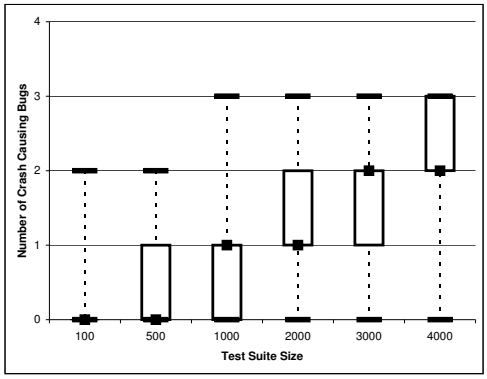
TerpSpreadSheet



TerpPaint



TerpCalc



TerpPresent

Figure 4.13: Number of Bugs vs. Number of Test Cases

inside each box marks the median value. The edges of the box mark the first and third quartiles. The whiskers extend from the quartiles and cover the entire distribution. The median in the box-plots of Figure 4.13 shows that the number of bugs revealed increases with test suite size; however, as the overlaps between box-plots show, the number of bugs does not grow significantly with test suite size. Hence, even a small number (a few hundreds) of crash tests are sufficient to find software problems. This result answers Question 5.

To address Question 6, each software's EIG was transformed into an annotated EIG. Recall that the annotated EIG algorithm ensures that all crash tests are executed across multiple code changes. The following steps were performed:

1. Start with the original (faulty) subject applications. The number of crash-causing bugs in each application is already known.
2. Set a time interval $\mathcal{N}$ between software changes.
3. Use two techniques to generate and execute as many crash test cases as possible in this interval. For the first technique (called Random), a crash test case is randomly selected (without replacement), making sure that each test case was selected only once in one interval. For the second annotated-EIG approach (called Memory), boolean flags are updated on the event-interaction graph edges.
4. Examine the crashes reported and eliminate the revealed bugs.
5. Repeat Steps 3 and 4 until there are no more bugs.

Four values of $\mathcal{N}$ were used, *i.e.*, 15, 30, 60, and 90 minutes. The above steps were repeated 200 times; the results are the medians of 200 values.

The results are summarized in Figure 4.14 through Figures 4.16 . Each graph has two lines,[4] one for the control "Random" and the other for "Memory." The x-axis shows the intervals between code changes, the first one being the start interval 0. The y-axis shows the number of bugs remaining in the subject applications. Note that the result for TerpSpreadSheet is not shown since it has only one bug, making its results uninteresting.

As the graphs show, the memory-based rotation technique does better than the random technique, *i.e.*, all bugs are removed much sooner. In the case of TerpPaint, the random technique does not even eliminate all bugs. This result answers Question 6.

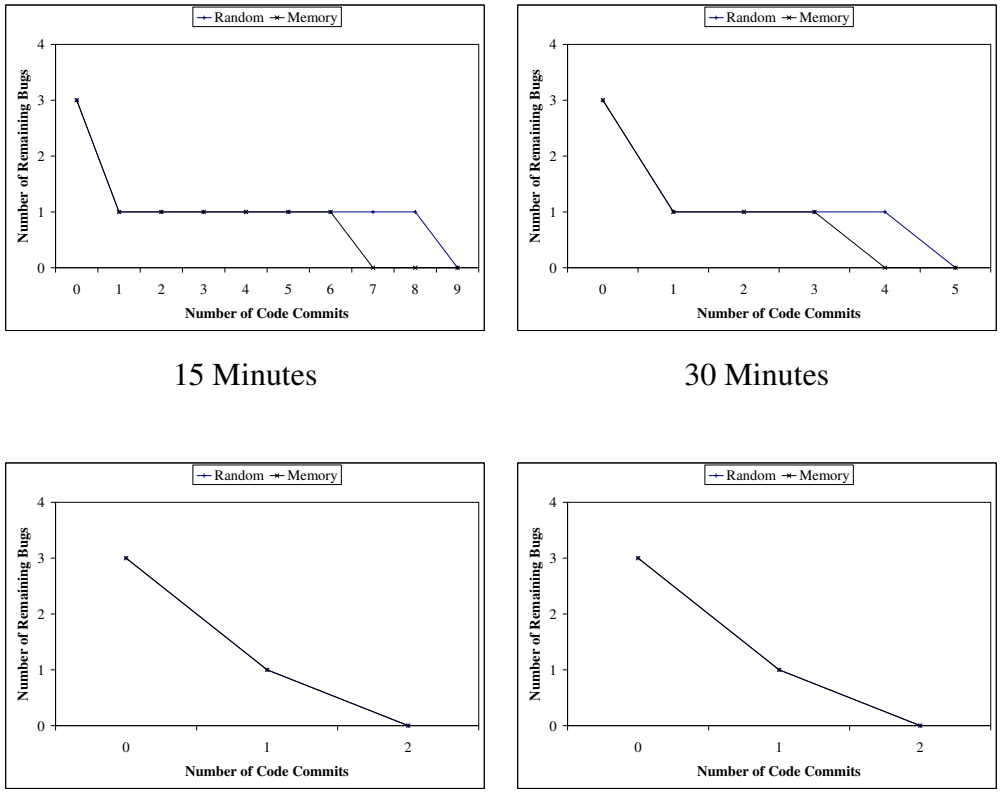


Figure 4.14: Effectiveness of the Rotating Algorithm for TerpCalc

[4]If only one line is visible, they are overlapping

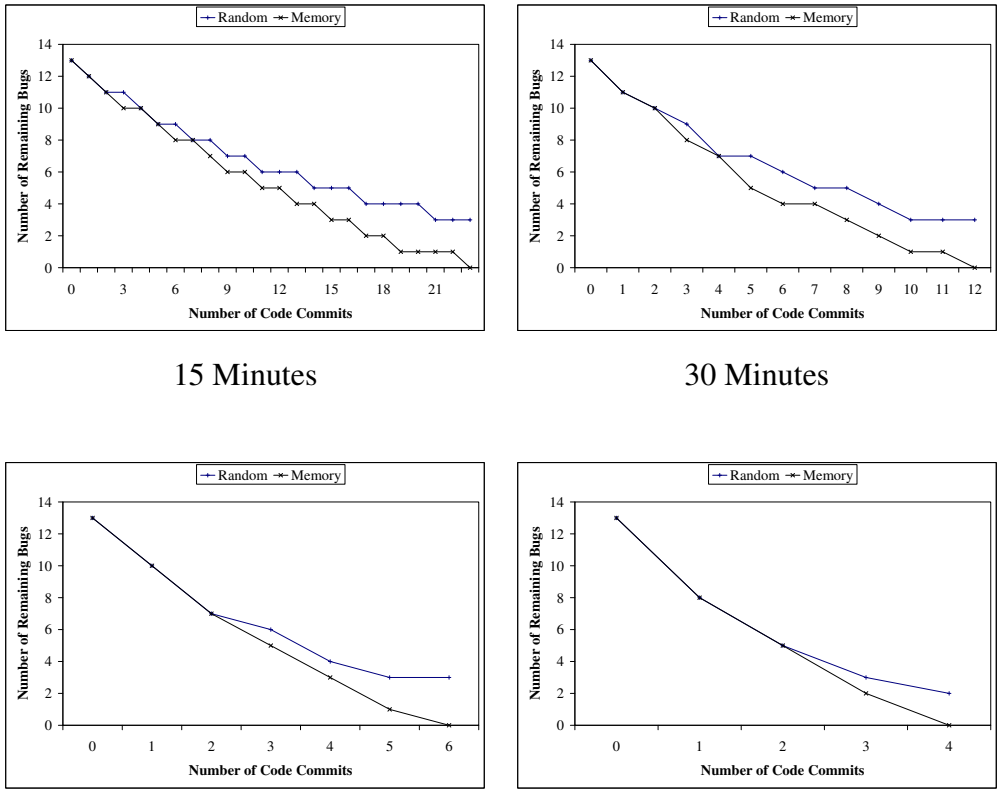
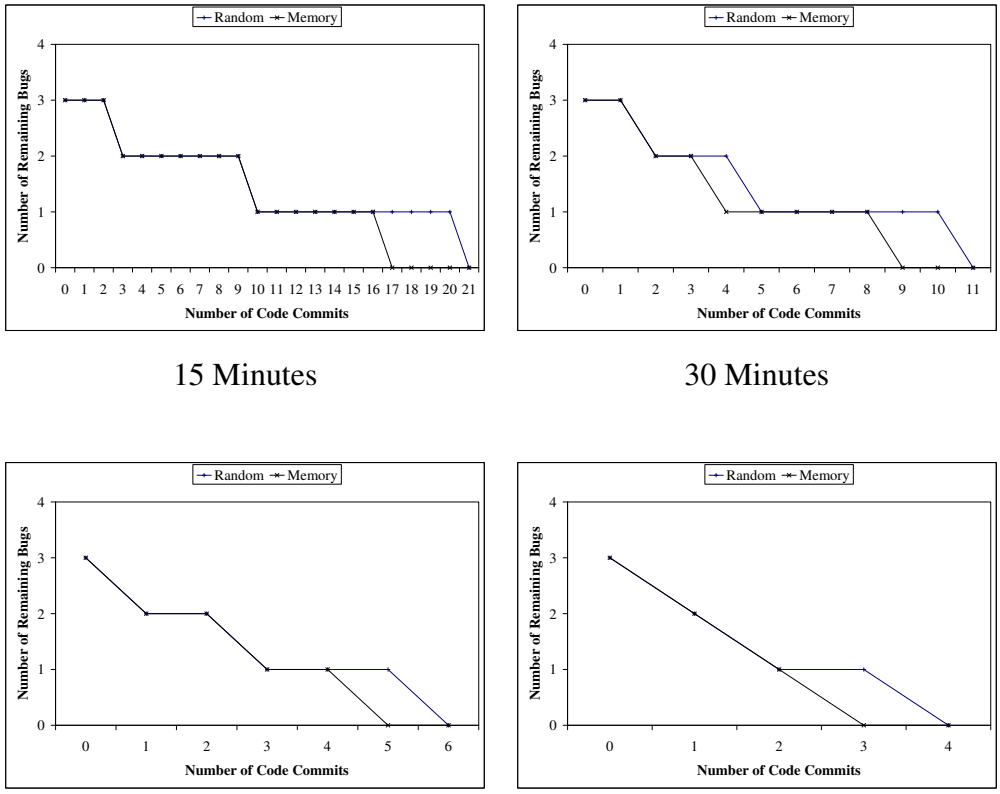


Figure 4.15: Effectiveness of the Rotating Algorithm for TerpPaint



15 Minutes

30 Minutes

60 Minutes

90 Minutes

Figure 4.16: Effectiveness of the Rotating Algorithm for TerpPresent

The questions posed in this study were answered using only four subject applications. These results may not hold for other GUI applications. Hence, at best, the answers to the six questions may be used to formulate hypotheses that need additional empirical evidence.

4.7.2 Feasibility Study - Crash Testing for Open-Source Applications

The goal of the second study is to determine whether fielded GUI-based open-source software (OSS), developed by a community of developers, have faults that may be detected using this approach. More specifically, the following questions need to be answered:

1. Do popular web-based community-driven GUI-based OSS have problems that can be detected by crash testing?

2. What is the nature of the crashes?

3. Do these problems persist across multiple versions of the OSS?

4. What are the common cases of crashes that can be detected by crash testing?

To answer these questions and to minimize threats to external validity, this study was conducted using several fielded GUI-based OSS downloaded from SourceForge.net. The fully-automatic crash testing process was executed on them and problems were reported. Previous versions of these applications were also downloaded to see how long these problems have been in the code. Note that only the versions that the developer community chose to make available online were tested. These applications are expected to

have undergone some QA before release.

Subject Applications: The following four applications with GUIs developed using Java Swing were chosen:

1. **FreeMind**[5], which is a premier free mind-mapping[6] software written in Java. It has an all time activity of 99.72%. Versions 0.0.2, 0.1.0, 0.4, 0.7.1, 0.8.0RC5 and 0.8.0 were tested.

2. **GanttProject**[7], which is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets). It has an all time activity of 98.12%. Versions 1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, and 2.pre1 were tested.

3. **JMSN**[8], which is a pure Java Microsoft MSN Messenger clone, including Instant messaging, File Send/Receive, msnlib (for developers), and additional chat log, etc. It has an all time activity of 98.93%. Versions 0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, and 0.9.9b1 were tested.

4. **CrosswordSage**[9], which is a tool for creating (and solving) professional looking crosswords with powerful word suggestion capabilities. When tested, it had an activity percentile (last week) of 98.21%. Versions 0.1, 0.2, 0.3.0, 0.3.1, 0.3.2, and 0.3.5 were tested.

The first three of the above applications were chosen due to their popularity, active community of developers, and high all-time activity. Crossword Sage was chosen since

[5]http://sourceforge.net/projects/freemind

[6]http://en.wikipedia.org/wiki/Mind_map

[7]http://sourceforge.net/projects/ganttproject

[8]http://sourceforge.net/projects/jmsn

[9]http://sourceforge.net/projects/crosswordsage

Subjects	Versions						
	0.0.2	**0.1.0**	**0.4**	**0.7.1**	**0.8.0RC5**	**0.8.0**	**Total**
FreeMind	1550	1964	4118	13658	50872	52216	124378
	1.6	**1.9.11**	**1.10.3**	**1.11**	**1.11.1**	**2.0.pre1**	**Total**
GanttProject	1240	3705	3878	4015	4015	4414	21267
	0.9a	**0.9.2**	**0.9.5**	**0.9.7**	**0.9.8b7**	**0.9.9b2**	**Total**
JMSN	1015	1107	1156	1218	1591	1777	7864
	0.1	**0.2**	**0.3.0**	**0.3.1**	**0.3.2**	**0.3.5**	**Total**
CrosswordSage	101	134	818	818	876	1524	4271
Total							157780

Table 4.6: Number of Test Cases Generated for Each Version of Each Application

Subjects	Versions						
	0.0.2	**0.1.0**	**0.4**	**0.7.1**	**0.8.0RC5**	**0.8.0**	**Total**
FreeMind	2	5	4	4	5	4	10
	1.6	**1.9.11**	**1.10.3**	**1.11**	**1.11.1**	**2.0.pre1**	**Total**
GanttProject	3	4	3	3	3	3	8
	0.9a	**0.9.2**	**0.9.5**	**0.9.7**	**0.9.8b7**	**0.9.9b2**	**Total**
JMSN	2	2	1	2	3	3	4
	0.1	**0.2**	**0.3.0**	**0.3.1**	**0.3.2**	**0.3.5**	**Total**
CrosswordSage	0	0	3	3	2	5	6
Total							28

Table 4.7: Number of Crashes Detected for Each Version of Each Application

it is fairly new (it was registered in mid-Sep. 2005) with several versions. All the above applications were tested on the Windows 2000 Professional platform.

The overall process executed on each version without any human intervention in 5-8 hours; one machine per application. The reverse engineering, model creation, test case generation steps took 2-3 minutes per application. The test cases execution took the remaining time.

To answer Question 1, a total of 157780 test cases (Table 4.6) were generated for FreeMind, GanttProject, JMSN, and CrosswordSage; these test cases revealed total 28 bugs (Table 4.7). Note that the the individual version bugs do not sum to the number in the Total column because a bug was counted several times if it was detected in different versions.

To address Question 2, all the crash logs were manually examined and the bugs in the code that caused the crash were identified. The analysis of the results is summarized next. Note that version numbers are shown in parenthesis. Each listed bug will be referred by its *bug number* in later discussions.

FreeMind: 1. NullPointerException when trying to open a non-existent file (0.0.2, 0.1.0);

2. FileNotFoundException when trying to save a file with a very long file name (0.0.2, 0.1.0, 0.4);

3. NullPointerException when clicking on some buttons on the main toolbar when no file is open (0.1.0);

4. NullPointerException when clicking on some menu items if no file is open (0.1.0, 0.4, 0.7.1, 0.8.0RC5);

5. NullPointerException when trying to save a "blank" file (0.1.0);

6. NullPointerException when adding a new node after toggling folded node (0.4);

7. FileNotFoundException when trying to import a non-existent file (0.4, 0.7.1, 0.8.0RC5, 0.8.0);

8. FileNotFoundException when trying to export a file with a very long file name (0.7.1, 0.8.0RC5, 0.8.0);

9. NullPointerException when trying to split a node in "Edit a long node" window (0.7.1, 0.8.0RC5, 0.8.0);

10. NumberFormatException when setting non-numeric input while expecting a number in "preferences setting" window (0.8.0RC5, 0.8.0);

Gantt Project: 1. NumberFormatException when setting non-numeric inputs while expecting a number in "New task" window (1.6);

2. FileNotFoundException when trying to open a non-existent file (1.6);

3. FileNotFoundException when trying to save a file with a very long file name (1.6, 1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

4. NullPointerException after confirming any preferences setting (1.9.11);

5. NullPointerException when trying to save the content to a server (1.9.11);

6. NullPointerException when trying to import a non-existent file (1.9.11, 1.10.3, 1.11, 1.11.1, 2.pre1);

7. InterruptedException when trying to open a new window (1.10.3);

8. Runtime error when trying to send e-mail (1.11, 1.11.1, 2.pre1);

JMSN: 1. InvocationTargetException when trying to refresh the buddy list (0.9a, 0.9.2);

2. FileNotFoundException when trying to submit a bug/request report because the submission page doesn't exist (0.9a, 0.9.2, 0.9.5, 0.9.7, 0.9.8b7, 0.9.9b2);

3. NullPointerException when trying to check the validity of the login data (0.9.7, 0.9.8b7, 0.9.9b2);

4. SocketException and NullPointerException when stopping a socket that has been started (0.9.8b7, 0.9.9b2);

Crossword Sage: 1. NullPointerException in Crossword Builder when trying to delete a word (0.3.0, 0.3.1);

2. NullPointerException in Crossword Builder when trying to suggest a new word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

3. NullPointerException in Crossword Builder when trying to write a clue for a word (0.3.0, 0.3.1, 0.3.2, 0.3.5);

4. NullPointerException when loading a new crossword file (0.3.5);

5. NullPointerException when splitting a word (0.3.5);

6. NullPointerException when publishing the crossword (0.3.5);

The above list of severe problems show that fielded GUI-based OSS developed by a community of developers have problems that are quickly uncovered using the crash testing process. Since the overall process is completely automatic, crash testing, integrated with CVS, can discover these problems before they are found by users.

To answer Question 3, the history of each bug was studied. Figure 4.17 gives an overview of bug history across versions of each application. The x-axis represents the versions; the y-axis uses the bug numbers assigned earlier. Each bug that led to one crash is represented by a small filled circle; bugs that led to multiple crashes are represented by an asterisk. If the same bug persisted across multiple versions, the circles (or asterisks) are connected by a horizontal line. For example, many crashes are caused by Bug#3 in FreeMind (several toolbar buttons should be disabled if there is no file opened).

Figure 4.17 shows that many bugs are persistent across versions. For example, Bug#4, #7, #8, #9 and #10 in FreeMind persisted across several versions before they were discovered and fixed. The same observation holds for the other applications. In fact, Bug#3 in GanttProject appeared in the first version tested (version 1.6 was chosen because it is the first version with default language English); it exists in all versions, including the latest version. This result answers Question 3.

To answer Question 4, the reasons for the crashes were studied. Four reasons were identified for these crashes: (1) *Invalid text input.* Many crashes were detected because the software does not check the validity and size of text input. For example, some text boxes in GanttProject and FreeMind expect an integer input; providing a string resulted in

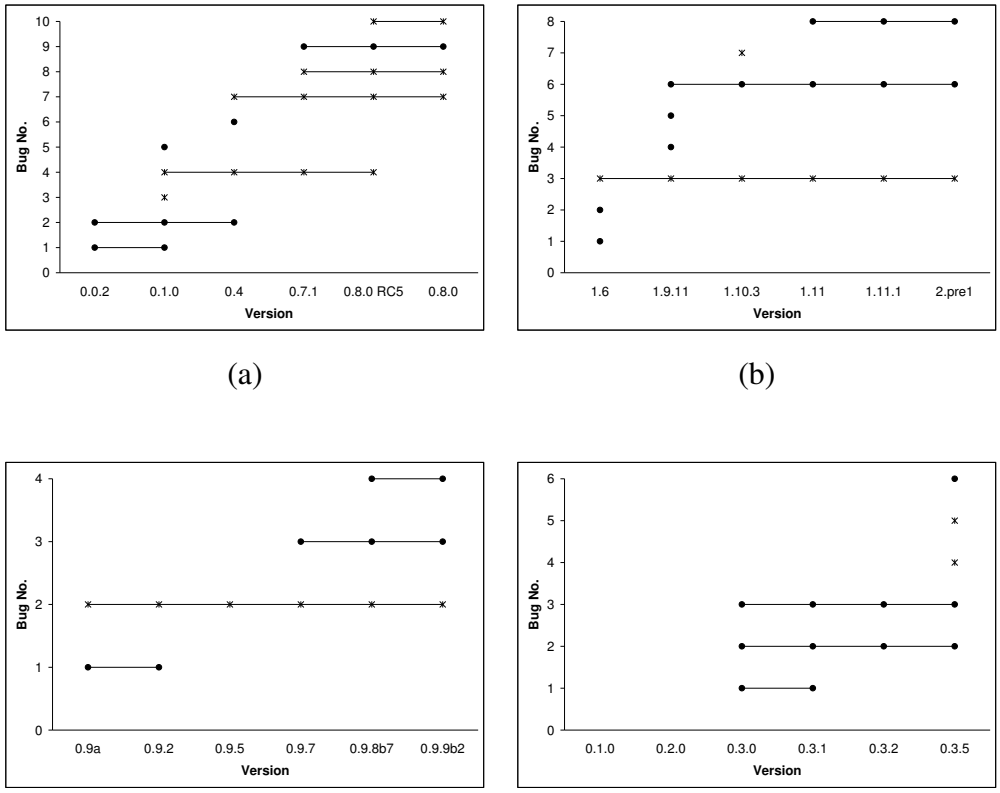


Figure 4.17: Bug History Over Versions

a crash. In some instances, a "very long" text input also resulted in a crash, such as providing a "very long" text input as the file name while saving such a file sometimes leads to FileNotFoundException. (2) *Widget enabled when it should be disabled.* One challenge in GUI design is to identify allowable sequences of interactions with widgets and to disallow certain sequences. Designers often disable certain widgets in certain contexts. In these open-source applications, it is found that several instances of widgets were enabled when they should really have been disabled. When the crash tests executed the incorrectly enabled widget in an event sequence, the software crashed. (3) *Object declared but not initialized.* Some of the crashes were Java NullPointerExceptions. It turned out that as

the software was evolving, one developer, not seeing the use of an object, commented out a part of the code, which was responsible for object initialization. Another developer continued to use the object in another part of the code. The software crashed when the uninitialized object was accessed. (4) *Obsolete external resources.* Some of the crashes in JMSN were caused by test cases that were trying to retrieve information from a web page that is no longer available. This result answers Question 4.

As mentioned earlier, the questions posed in this study were also answered using only four OSS applications. Consequently, these results may not hold for other GUI applications.

The crash test cases exhibited the following properties that satisfy the criteria for the innermost loop (presented in Chapter 3).

1. The test cases can be generated automatically and executed very quickly.

2. All system-interaction and termination events are executed; most of the GUI's functionality is covered.

3. The rotation-based scheme ensures that the entire GUI is tested over a series of code changes.

4. The abnormal termination of the program (serves as the "test oracle") can be determined fully automatically.

Additional Lessons Learned

Since SourceForge has a bug reporting/tracking tool for each project, some bugs were reported. For example, Bug#4 in FreeMind for version 0.8.0RC5 was reported (bug #1245216 in SourceForge[10]). In response to the report, the developers fixed this bug in release 0.8.0. This showed that the bugs found by the crash testing were relevant. All other bugs will be reported, especially the ones in the latest versions of all the applications.

Figure 4.17 leads to another observation. There are fewer bugs in the first version than in later versions. For example, there are two crash-causing bugs in Version 0.0.2 of FreeMind. Typically, the first version of an OSS is relatively simple and is developed by a small group of core developers. This version typically undergoes QA before its first release; hence it is reasonably stable. Versions 0.1.0 and 0.2.0 of CrosswordSage have no bugs because they are very simple. The only change that was made from Version 0.1.0 to Version 0.2.0 was a new help document. As the developer community grows, the application becomes more complex and prone to bugs. For example, Bug#10 in FreeMind was first introduced when a new "preference setting" functionality was added. Similarly, there was a new feature added to Version 0.3.0 of Crossword Sage; this new feature introduced some bugs that were detected. There were more features added in Version 0.3.5; bugs were detected in the added part of code.

By default, all the applications were tested in one machine configuration on Windows 2000 Professional. It is observed that altering this "default" configuration helps to uncover more bugs. In a preliminary study, GanttProject was tested in a new configura-

[10]http://sourceforge.net/tracker/index.php?func=detail &aid=1245216&group_id=7118&atid=107118

tion with a much lower memory setting than the default configuration. Bug#4 and Bug#7 surface only in this low memory configuration. In case of Bug#4, the application tries to repaint all the GUI windows/widgets after the preferences setting have changed; in low memory, this causes a substantial delay for the user. Any event performed during the slow repainting process causes an uncaught NullPointerException exception. In case of Bug#7, the application requires additional time to open new windows; if a user performs a new event during this time, the result is an uncaught InterruptedException exception.

A surprising result is that some bugs existed across applications. This was due to shared open-source GUI components. For example, Bug#2 in FreeMind and Bug#3 in GanttProject are identical since both these applications share a *FileSave* component. This component throws a FileNotFoundException when given a very long file name, which cannot be handled by the Windows operating system. This particular bug does not show up after Version 0.4 of FreeMind; however, the same bug still shows up when the user tries to *export* a file with a very long file name. This observation shows that OSS that use shared components must "sanitize" inputs before passing them to the shared components.

4.8 Conclusions

This chapter presented the innermost loop called crash testing for continuous integration testing of GUI-based software. It operates on each code check-in of the GUI software and performs a quick-and-dirty, fully automatic integration test of the GUI software; feedback is directed to the developer who initiated the check-in. This chapter evaluated the crash testing process in two studies involving four TerpOffice applications and

four popular OSS. The studies showed that (1) the crash testing approach helps to find integration problems in GUI-based software quickly, (2) test cases generated using EIG reveals more crashes than those that are generated using EFG, (3) several problems persist across multiple versions of OSS, (4) errors surface in different OSS that share problematic open-source GUI components, and (5) the first version (likely created by a core group of developers) of most OSS is relatively stable; problems surface as additional developers add functionality. Post-study analysis revealed that most of these problems are caused by incorrect integration of different parts of the OSS.

Chapter 5

Smoke Testing

This chapter describes the second loop (smoke testing) of the continuous GUI testing process. Smoke testing is more complex than crash testing in that (1) it operates on each day's GUI build, testing a set of changes, (2) its goal is to do functional "reference testing" of the newly integrated version of the GUI, not just detecting crashes, (3) it requires additional effort on the part of the test designer who has to identify false positives, and (4) it requires additional information to be specified in the feedback, *i.e.*, the exact mismatches that led to test case failures.

Smoke testing shares several criteria with crash testing, *i.e.*, the test cases should be generated and executed quickly and they should cover the GUI's entire functionality. The differences are that smoke testing should maintain a test suite that is largely reusable across GUI versions and is divisible, and because it is a form of reference testing, it requires a test oracle to compare the current version's output with that of the previous version.

Due to the similarity of some crash and smoke testing criteria, the event-interaction graph (EIG) model is reused to generate the event sequence part of the smoke test cases. An advantage of reusing the EIG model is that each test case consists of only system-interaction events and termination events; changes to the GUI layout, such as moving events from one window to another and changing the menu structure, leave most of the

test cases unaffected. Other events are generated on-the-fly *during* test execution. Hence, the first four criteria for smoke testing (Section 3.2) are already satisfied.

The last criterion is related to the test oracle. Smoke testing requires a test oracle that can be used to compare the two GUI versions. The most straightforward approach is to compare the entire GUI's state after each event of the smoke test case. An experiment in Section 5.2 will show that this test oracle is useful but expensive.

To reduce cost and retain fault detection effectiveness, the remainder of this chapter develops different types of test oracles and studies the amount of oracle information that should be specified in the expected output and the frequency at which it should be compared for effective testing.

5.1 Designing Different Test Oracles

As mentioned in Section 2.3, a GUI test oracle consists of *oracle information* and *oracle procedure*. Different types of GUI test oracles may be created by varying the *oracle information* and *oracle procedure*.

5.1.1 Oracle Information

The oracle information is a description of the GUI's expected state for a test case. Recall from Section 2.1.1 that the GUI's state is a set of triples of the form (w_i, p_j, v_k), where w_i is a widget, p_j is a property of w_i, and v_k is a value for p_j. Hence the oracle information for a test case is a sequence of these sets. Note that oracle information has been deliberately defined in very general terms, thus allowing the creation of different

instances of oracles. The least descriptive oracle information set may contain a single triple, describing one value of a property of a single widget. The most descriptive oracle information would contain values of all properties of all the widgets, *i.e.*, the GUI's complete expected state. In fact, all the non-null subsets of the complete state may be viewed as a spectrum of all possible oracle information types, with the single triple set being the smallest and the complete state being the largest. The following three types of oracle information will be considered in this research:

1. **widget (LOI1):** the set of all triples for the single widget w associated with the event e_i being executed. The constraint is written as $(\#1 == w)$, where $\#1$ represents the first element of the triple. If applied to a triple with "w" as its first element, the constraint would evaluate to `TRUE`; in all other cases, it would evaluate to `FALSE`. Figure 5.1 shows an example of the oracle information. The test case contains the `Cancel` event in the `Find` window. The complete expected state S_i of the GUI after `Cancel` has been executed is also shown. For the *widget* level test oracle information, only the (boxed) triples relevant to `Cancel` are stored.

2. **active window (LOI2):** the set of all triples for all widgets that are a part of the currently active window W. The constraint is written as $(inWindow(\#1, W))$, where $inWindow(a, b)$ is a predicate that is `TRUE` if widget a is a part of window b.

3. **all windows (LOI3):** the set of all triples for all widgets of all windows. Note that the constraint for this set is simply `TRUE` since it is the complete state of the GUI.

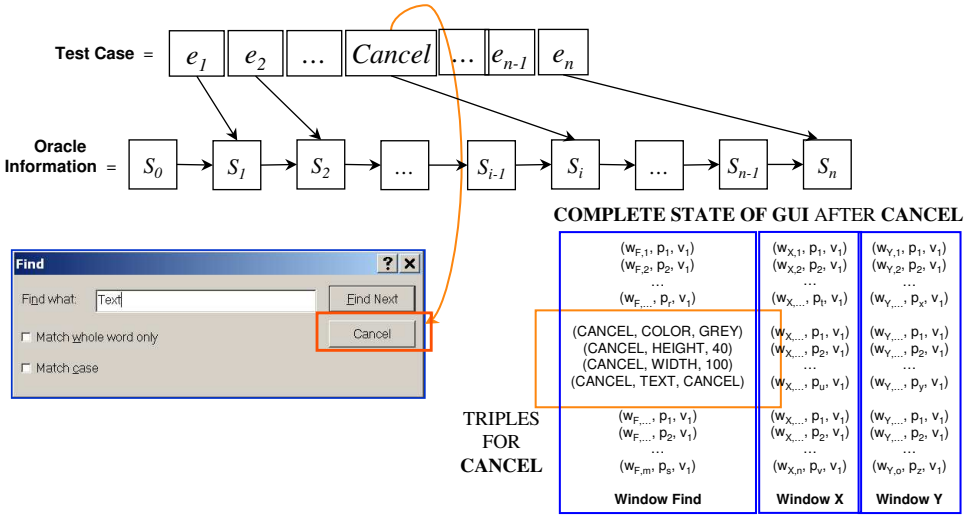


Figure 5.1: Oracle Information for the `Cancel` Event

For brevity, the terms LOI1 to LOI3 will be used for the above three *levels of oracle information*. Note that although only three instances of test oracle information have been specified, the specification mechanism is general and may be used to specify many other instances.

5.1.2 Oracle Procedure

The oracle procedure is the process used to compare the oracle information with the executing GUI's actual state. It returns `TRUE` if the actual and expected states match, `FALSE` otherwise. Formally, a **test oracle procedure** is a function ζ(**OI**, **AS**, $\mathbf{C}_{OI}$, $\mathbf{C}_{AS}$, Φ) $\longrightarrow$ {`TRUE`, `FALSE`}, where **OI** is the oracle information, **AS** is the actual state of the executing GUI, $\mathbf{C}_{OI}$ is a boolean constraint on **OI**, $\mathbf{C}_{AS}$ is a boolean constraint on **AS**, and Φ is a comparison operator. ζ returns `TRUE` if **OI** and **AS** "match" as defined by Φ; `FALSE` otherwise.

The oracle procedure may be invoked as frequently as once after every event of

ALGORITHM :: `OP(`

AS$_i$: Actual state; */* for event e_i */*	1
OI$_i$: Oracle information; */* for event e_i */*	2
C$_{AS}$: Boolean Constraint; */* on actual state */*	3
OPF $\subseteq \{1, 2, 3, \ldots, n\}$ */* oracle procedure freq. */*	4
i: event number; */* current event index* $1 \leq i \leq n$ **/*) $\{$	5
`IF` ($i \in$ **OPF**) `THEN` */* compare? */*	6
RETURN(FILTER(**OI**$_i$, **C**$_{AS}$) == **AS**$_i$)	7
`ELSE` RETURN(TRUE)$\}$	8

Figure 5.2: Oracle Procedure Algorithm

the test case or less frequently, *e.g.*, after the last event. The algorithm for the oracle procedure is shown in Figure 5.2. Note that this specific implementation `OP` of ζ takes extra parameters i and **OPF** that account for this frequency; i is the event number in the test case and **OPF** is a set of numbers that specify when the comparison is done. Also note that Φ is hard-coded to "set equality", hence omitted from `OP`'s parameters (Line 7 of Figure 5.2). $\mathbf{C}_{OI}$ is also omitted since **OI** has already been filtered before `OP` is invoked. `OP` takes five parameters described earlier. The comparison process is straightforward – if the GUI needs to be checked at the current index i of the test case (LINE 6), then the oracle information is filtered[1] using the constraint $\mathbf{C}_{AS}$ to allow for *set equality* comparison. The constraint $\mathbf{C}_{AS}$ (not $\mathbf{C}_{OI}$) ensures that the result of the filtering is compatible with $\mathbf{AS}_i$. The oracle procedure returns `TRUE` if the actual state and oracle information sets are equal.

Note that it is important to provide the constraint $\mathbf{C}_{AS}$ and the set **OPF** to completely specify the oracle procedure. The definition of `OP` is now used to specify six different instances of test oracles.

- **L1:** *After each event of the test case*, compare the set of all triples for the single widget w associated with that event. The constraint $\mathbf{C}_{AS}$ is written as $(\#1 == w)$ and **OPF** = $\{1, 2, 3, \ldots, n\}$. Note that $\mathbf{C}_{AS}$ is first used to select relevant triples for the actual state and then later to filter the oracle information. **L1** is shown in Figure 5.3; it compares the state triples relevant to the widget $\mathbf{W_x}$.

[1]Note that this filtering is unnecessary if `OP` is invoked by the test case executor, since it already filters the oracle information. The filtering step is included here for completeness.

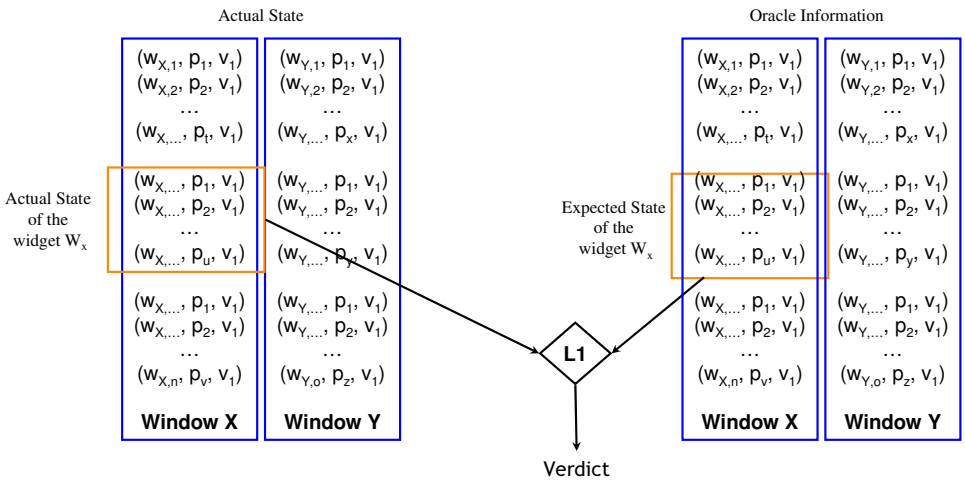


Figure 5.3: **L1** Compares Widget-Relevant Triples after Each Event in the Test Case

- **L2:** *After each event of the test case*, compare the set of all triples for all widgets that are a part of the currently active window W. The constraint $\mathbf{C}_{AS}$ is written as $(inWindow(\#1, W))$ and **OPF** = $\{1, 2, 3, \ldots, n\}$.

- **L3:** *After each event of the test case*, compare the set of all triples for all widgets of all windows. The constraint $\mathbf{C}_{AS}$ is written as `TRUE` and **OPF** = $\{1, 2, 3, \ldots, n\}$.

- **L4, L5, L6:** *After the last event of the test case*, compare the set associated with the current widget, active window, and all windows, respectively. **OPF** = $\{n\}$ for all these oracles.

Even though only six instances of oracles have been developed, the definition of `OP` is very general and may be used to develop a variety of test oracles.

5.2 Evaluating the GUI Test Oracles

Having presented the design of GUI test oracles and ability to specify multiple oracles, different oracles are now compared via an experiment.

5.2.1 Research Questions

Two notations are introduced: $C(T, L)$ – the cost of executing a test case T with oracle L; $F(T, L)$ – the number of faults that test case T detects when using test oracle L.

The way that the test oracles L1 to L6 are defined leads to some immediate observations. First, it is noted that L1 to L3 have been defined with increasing complexity (as have L4 to L6), which will have a direct impact on their relative cost (*i.e.*, time to generate/execute); L3 will be most expensive and L1 the least expensive (hence, for all test cases T, $C(T, L3) \geq C(T, L2) \geq C(T, L1)$). Similarly, $C(T, L6) \geq C(T, L5) \geq C(T, L4)$. Also, it is obvious that $C(T, L4) \leq C(T, L1)$, $C(T, L5) \leq C(T, L2)$, and $C(T, L6) \leq C(T, L3)$. Second, T with oracle L3 is expected to reveal more faults than T with oracle L1 or L2, simply because L3 "looks at" a larger set of GUI widgets during T's execution (*i.e.*, $F(T, L3) \geq F(T, L2) \geq F(T, L1)$); it can certainly do no worse. Similarly, T with L6 is expected to reveal more faults than with either L4 or L5 (*i.e.*, $F(T, L6) \geq F(T, L5) \geq F(T, L4)$). It is, however, not clear how L1 compares to L4 in terms of T's fault-detection effectiveness, *i.e.*, is $F(T, L4) < F(T, L1)$ or is $F(T, L4) = F(T, L1)$? (similar questions can be asked about the pairs (L2, L5) and (L3, L6)). Also, even though the above relationships have been presented as "obvious," the magnitude of these relationships needs further study to determine practical signifi-

cance. For example, even though, in theory, the relationship $C(T, L5) \leq C(T, L2)$ holds, *how much more* does L2 cost? Is the additional cost worth the extra faults that may be found (if any) when using L2? Answers to these questions will demonstrate the practical significance of using different test oracles.

In particular, the following questions need to be answered to show the relative strengths of the test oracles and to explore the cost of using different types of oracles.

- **Q1:** What effect does the oracle information have on the fault-detection effectiveness of a test case? Is the additional effectiveness worth the cost?
- **Q2:** What effect does the invocation frequency of a test oracle have on the fault-detection effectiveness of a test case? Is the additional effectiveness worth the cost?
- **Q3:** What combination of oracle information and procedure provide the best cost-benefit ratio?

While answering the above questions, the situations in which generating/using a complex (more expensive) oracle are justified will also be informally studied. For example, if a tester has only short test cases (and/or a small number of test cases), will the test results improve if complex oracles are used? This question will be referred to as **Q4**.

5.2.2 Modeling Cost and Fault Detection Effectiveness

One factor of cost is the time needed to execute a test case with a given oracle; this time is directly proportional to the number of comparisons of $(widget, property, value)$ triples during test case execution. Hence, the number of *widget comparisons* done (during execution of test case T) by test oracle L is used as a measure of cost. The notation

$\mathcal{C}(T, L)$ is used for this measure. For example, $\mathcal{C}(T, L4) = 1$ for all test cases, since L4 involves comparing the triples for a single widget.

Because the impact of using different test oracles is studied on each test case, the fault-detection effectiveness is modeled on a per test case basis. $\mathcal{F}(T, L)$ of a test case T is defined as the number of faults it detects with test oracle L. Obviously, a higher value of $\mathcal{F}$ is desirable but at a reasonable cost. A more appropriate measure called the "number of faults detected per comparison" (ξ) is computed as:

$$\xi(T, O) = \begin{cases} \frac{\mathcal{F}(T,L)}{\mathcal{C}(T,L)} & \text{if } \mathcal{C}(T, L) > 0, \\ undefined & \text{if } \mathcal{C}(T, L) = 0. \end{cases}$$

The second case of the definition is included only for completeness; as long as T is a non-empty sequence, $\mathcal{C}(T, L)$ will be positive. The ξ value gives a good measure of the relative cost and benefit of test oracles. A test oracle that performs very few comparisons yet reveals a large number of faults will have a high ξ value, which is desirable due to the larger number of faults that it detects. However, ξ has several weaknesses. First, a test oracle that performs very few (say x) (*e.g.*, $x = 1$ for L4) comparisons and reveals too few faults (say y) will have a higher ξ value than one that performs more comparisons (*e.g.*, $10x$) but detects more faults (*e.g.*, $5y$). However, the latter oracle may be more desirable. In practice, the cost of missing a fault may be extremely high. Indeed, in particular domains, a tester may be willing to spend considerable resources to detect even a single fault. In such domains a test oracle with a high average $\mathcal{F}$ value is clearly desirable. Second, all faults are given equal weight in this model. The ξ formula can be easily modified if the "severity" of faults is to be considered; in this experiment all faults are

considered to be of equal severity. Although ξ suffers from some of these problems, it provides an adequate starting point for oracle comparison. Recognizing the weaknesses of the cost/benefit model, details of the actual number of faults detected are presented; readers can interpret the results for their particular domains/situations.

To answer Q1, $\mathcal{F}$ and ξ values for oracles L1–L3 and L4–L6 will be compared. To answer Q2, the $\mathcal{F}$ and ξ values for the oracle pairs (L1,L4), (L2,L5), and (L3,L6) will be compared. For Q3, the average ξ values of all oracles will be compared. Finally, for Q4, the impact of test case length and their number on the ξ values for each oracle will be studied.

5.2.3 Experimentation Procedure

Four TerpOffice applications (TerpPaint, TerpPresent, TerpWord, TerpSpreadSheet) were selected, and, for each application, the following steps were performed:

Step 1: generate test cases,

Step 2: generate different levels of oracle information,

Step 3: execute the test cases on the application using different oracle procedures. Measure the following variables:

Number of Faults Detected: A "fault is detected" if the expected and actual states mismatch.

Number of Comparisons: This is the number of widget comparisons between the expected and actual states for each oracle.

Step 4: from the execution results, eliminate test runs that were affected by factors be-

yond control, *e.g.*, those that crash the subject application irrespective of the test oracle used.

Details of these steps are discussed in subsequent sections.

Step 1: Generate Test Cases

600 test cases were generated for each application. The number 600 was chosen because the test cases could be executed in a reasonable amount of time; 100 fault-seeded versions of each application were selected from the original pool of fault seeded versions (Section 4.2.3); with 600 test cases, 100 versions, and 4 applications, there are total 240K test runs; since an average test run takes 30 seconds, the experiment would run for months. The number 600 allowed the experiment to be kept within the realm of practicality.

Each GUI's EFGs were used to generate test cases. Since one of the question is to study the role of test case length in GUI testing (Q4), an algorithm was used that allowed the control of the length of the test case by specifying a limit on the graph traversal. Hence, a set of buckets of test cases by length were created. One of the problems with automated GUI testing is the creation and execution of long test cases. Experience with GUI testing tools has shown that test cases longer than 20 events typically run into problems during execution, mostly due to timing issues with windows rendering. As events in a test case are executed, the test case replayer keeps track of GUI state information for each event. For long sequences, the overhead of keeping track of this information significantly affects the performance of the JVM, which is also responsible for executing the subject application. After 20 events, window rendering becomes so slow that events are executed

even before the corresponding widget is available, resulting in uncaught exceptions. Because of this limitation of this tool, the GUI length was capped at 20, *i.e.*, there are 20 buckets, one for each length. Since no bucket should be favored, an equal number of test cases, *i.e.*, 30, was generated per bucket. In all there were 600 test cases per application.

Step 2: Generate Oracle Information

The next step was to obtain the oracle information for each test case. The oracle information was obtained from the "correct" version of the subject application and used to test the other versions of the application. An automated tool was implemented to create this oracle information. This tool automatically executes a given test case on a software system and captures its state (widgets, properties, and values) by using the Java Swing API. Due to the limitations of this API, only 12 properties can be extracted for each widget. The oracle information was obtained by running this tool on the four subject applications for all 600 test cases. Note that the tool extracted all three levels of oracle information.

Step 3: Oracle Procedure and Test Executor

All 600 test cases were executed on all 100 versions of each subject application (hence there were 60,000 runs per application). When each application was being executed, its run-time state based on the six oracles were extracted and compared with the stored oracle information and reported mismatches. The attribute "set equality" was used to compare the actual state with the oracle information. Note that widget positions were

ignored during this process since the windowing system launches the software at a different screen location each time it is invoked.

Each test case required between 5 and 60 seconds to execute. The time varied by application and the number of GUI events in the test case. The total execution time was slightly less than one month for each application.

The resulting data can be viewed as a (hypothetical) table (hereafter referred to as the "data table") for each application. Each row of this table represents the result of executing each test case on each fault-seeded version. Hence the table has $600 \times 100 = 60,000$ rows. It has 6 columns, one for each test oracle. Each entry of the table is a boolean value (`Match`/`Mismatch`) indicating whether at least one mismatch occurred (the fault was detected) during test case execution when using the corresponding oracle.

Step 4: Cleaning up the Data Table

During test execution, two factors that were independent of test oracle caused filtering out some of the rows in the data table. These factors include the impact of seeded faults on software execution and interactions between test cases and faults. The former is due to the way a fault is manifested during execution. The latter is due to test-case design, whether the test case caused the execution of the program statement in which the fault was seeded, and whether the seeded fault was manifested on the GUI. Each of these issues are listed and discussed next:

1. *Effect of fault on software execution:* several test cases (during execution) crashed specific fault-seeded versions, irrespective of the test oracle. These test cases ex-

ecuted properly on other versions. Such crashes were eliminated from the data. There were 954, 1595, 2302, and 4829 crashes for TerpPresent, TerpWord, TerpPaint, and TerpSpreadSheet respectively. Each of these *(test case, fault-seeded version)* pairs caused the filtering of one row in the table.

2. *Fault design:* several faults were never detected by even a single test case. These faults are "unobserved." There were 58, 5, 43, and 1 unobserved faults for TerpPresent, TerpWord, TerpPaint, and TerpSpreadSheet respectively. These faults are discarded from the data. For each such fault, a maximum of 600 table rows were filtered out, one for each test case.
3. *Test case design:* one test case in TerpPaint did not detect even a single fault for any oracle. This test case was eliminated, causing the filtering of 57 rows, one for each of the remaining fault-seeded versions of TerpPaint.
4. Finally, a large number of test cases did not detect certain faults for any test oracle. These rows were eliminated from the table.

Subject Applications	**TerpPresent**		**TerpWord**		**TerpPaint**		**TerpSpreadSheet**	
Total Rows		60000		60000		60000		60000
Filtering Steps	*#*	*Rows Filtered*	*#*	*Rows Filtered*	*#*	*Rows Filtered*	*#*	*Rows Filtered*
1 *Crashes*		954		1595		2302		4829
2 *Unobserved faults*	58	34800	5	3000	43	25800	1	600
3 *Test cases not detecting any faults*	0	0	0	0	1	57	0	0
4 *Test cases not detecting specific faults*		20566		54013		31779		52323
Remaining Rows		**3680**		**1392**		**62**		**2248**

Table 5.1: The Data Table Cleanup Steps

These "filtering steps" are also shown in Table 5.1. Note that they were executed in the order presented. Also note that after the last filtering step, some fault-seeded versions

may have been filtered out entirely, since test cases either crashed them or did not detect the faults.

The remaining data, which is used for the analysis, are the rows of the data table that contain at least one `Mismatch` entry. These rows represent test runs that yielded a successful fault detected for at least one test oracle. That is, the test case successfully executed the program statement in which the fault was seeded and the fault manifested as a GUI error. This data is relevant to the results since it helps to compare test oracles. Note that other entries may be useful for other analyses, *e.g.*, to study characteristics of test cases, which are beyond the scope of this work.

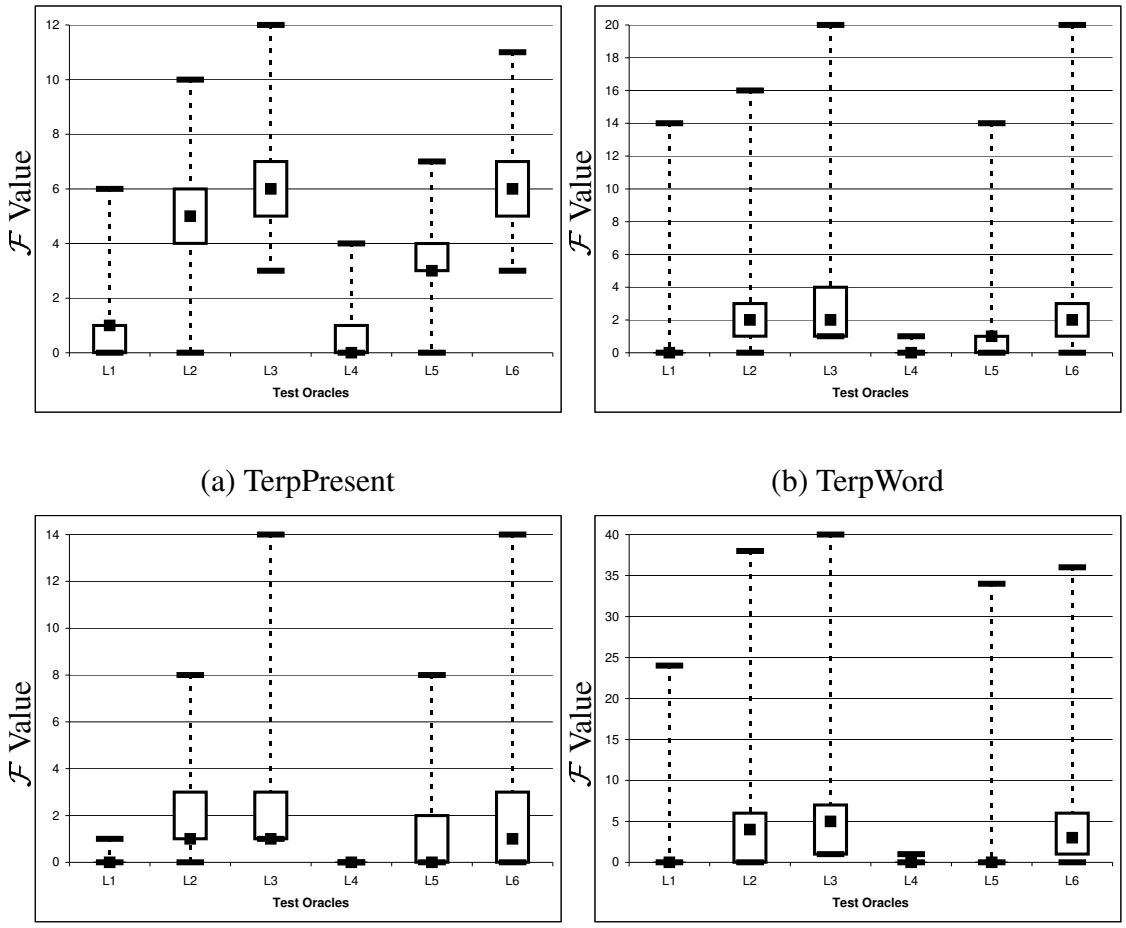
The number of test cases that appeared in at least one row of the resulting data table were 600 for TerpPresent, 424 for TerpWord, 18 for TerpPaint, and 358 for TerpSpreadSheet. Similarly, the number of faults that appeared in at least one row in the table were 25 for TerpPresent, 82 for TerpWord, 18 for TerpPaint, and 83 for TerpSpreadSheet. These numbers will be used in the analyses presented. It should be noted that the threats to validity stated in Section 4.4 also hold for this experiment.

5.2.4 Results

Fault-Detection Effectiveness

Recall that $\mathcal{F}(T, L)$ was defined as the number of faults detected by test case T when using oracle L. This value is computed from the data table as $\mathcal{F}(T, L) = \sum_{f \in F} DT(T, f, L)$, where the function DT returns 1 if the the entry for column L in the row corresponding to test T and fault f is `Mismatch`; 0 otherwise. F is the set of all

faults in the data table.



(a) TerpPresent (b) TerpWord

(c) TerpPaint (d) TerpSpreadSheet

Figure 5.4: Distribution of $\mathcal{F}$ Values by Test Oracle

The $\mathcal{F}$ values for each test case are summarized as box-plots in Figure 5.4. There are four box-plots in Figure 5.4, one for each subject application. For example, Figure 5.4(a) shows the results for TerpPresent. This plot contains six boxes, corresponding to the six test oracles. The x-axis lists the oracles and the y-axis shows the $\mathcal{F}$ values. From visual examination of the graph, it is noted that L2 (mean $\mathcal{F}$ value = 5) does better than L1 (mean $\mathcal{F}$ = 1). However, L3 (mean $\mathcal{F}$ = 6) is very close to L2. Comparing L4, L5, and

L6, note that the difference between L4 and L5 is not as stark as the difference between L1 and L2; moreover, L6 does better than L5 (which was not the case for L2 vs. L3). Comparison of L1 to L4 (mean $\mathcal{F} = 0.5$) shows that L1 does better than L4. Similarly L2 does better than L5 (mean $\mathcal{F} = 3.5$). However, L3 and L6 are very close. The results for the other applications are more or less similar; the only visual difference is that L3 does better than L6 for these applications.

In summary, visual examination of the box-plots suggest that the "effectiveness order" of test oracles (as measured by their mean $\mathcal{F}$ values) is $\{$L3, L6, L2, L5, L1, L4$\}$, *i.e.*, L3 is the best and L4 is the worst. This result suggests that the oracle information and execution frequency does have an impact on fault-detection effectiveness. Checking the entire state as opposed to only the active window is effective if the oracle is invoked after the last event in the test case. If, on the other hand, the oracle is invoked after each event, then checking only the active window does well. With the exception of TerpPresent, checking the current widget seems ineffective.

As demonstrated above, box-plots are useful to get an overview of data distributions. However, valuable information is lost in creating the abstraction. For example, it is not clear *how many* test cases detected specific numbers of faults. This is important to partially address Q4. Even though L3 and L6 more or less showed similar results in the box-plots, *do more test cases* detect more faults with L3 than L6? If this is the case, a tester who has a small number of test cases may get better results with L3 and L6.

The number of test cases that detected specific numbers of faults for different test oracles is shown in Figure 5.5. It shows six histograms for TerpPresent, one for each test oracle. The x-axis represents the $\mathcal{F}$ values; the y-axis shows the number of test cases

that had the particular $\mathcal{F}$ values. There are several important points to note about these plots. First, they have an $\mathcal{F} = 0$ column (the first dark column; in some cases this column is very tall; in these cases, it has been chopped – the number adjacent to the top of the column represents its height); this column is important since it accounts for test cases that detected faults with at least one test oracle but not with the current oracle. Second, the sum of all the columns is equal to the number of test cases in the "filtered" data table.

To allow easy visual comparison, the same x-axis and y-axis scales are used for all six plots. For TerpPresent, there is a larger number of test cases have a larger $\mathcal{F}$ value for L3 and L6. In fact, the zero column for L3 and L6 contains no test cases, *i.e.*, all test cases detected at least one fault when using L3 and L6. The zero column is tallest for L4, followed by L1. Hence a large number of test cases did not detect even a single fault when using L1 and L4. In case of TerpWord (Figure 5.6), approximately 60 test cases did not detect even a single fault for L6. Moreover, the column corresponding to $\mathcal{F} = 1$ for oracle L3 is shorter than that of L2; however, a larger number of test cases have higher $\mathcal{F}$ values. For TerpPaint (Figure 5.7), the oracle L4 detected no faults, represented by a single zero column of height 18. For TerpSpreadSheet (Figure 5.8), L3 did significantly better than L2, indicated by a taller $\mathcal{F} = 1$ column; L2 has a very tall $\mathcal{F} = 0$ column.

The probability that a test case will detect a larger number of faults with L3 is high. It was also noted that oracle L6 does reasonably well. Oracle L4 has the largest number of test cases with zero faults detected. In summary, a tester with a small number of test cases can improve overall fault detection effectiveness by using oracle L3. This result partly answers Q4.

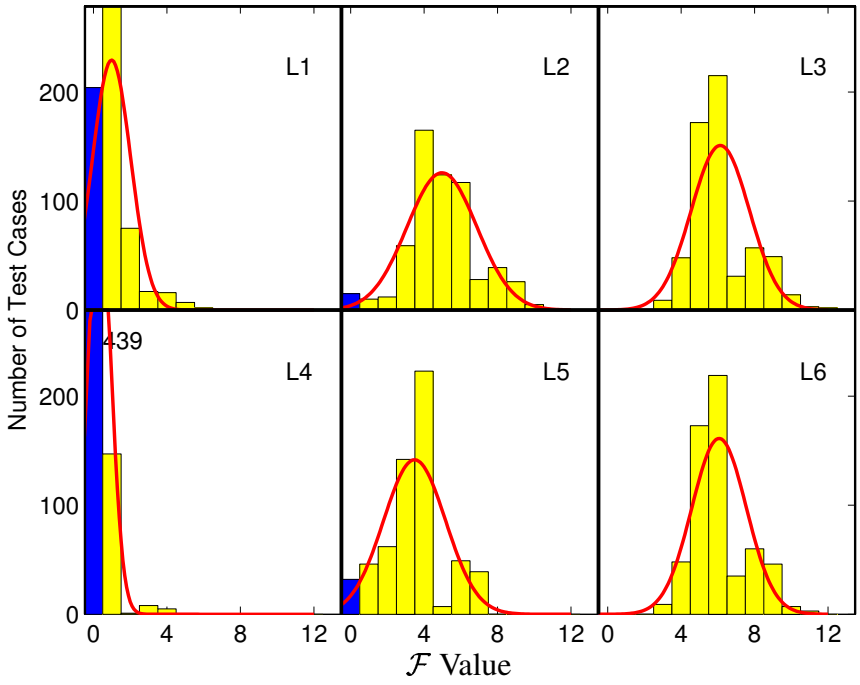


Figure 5.5: Histogram for TerpPresent

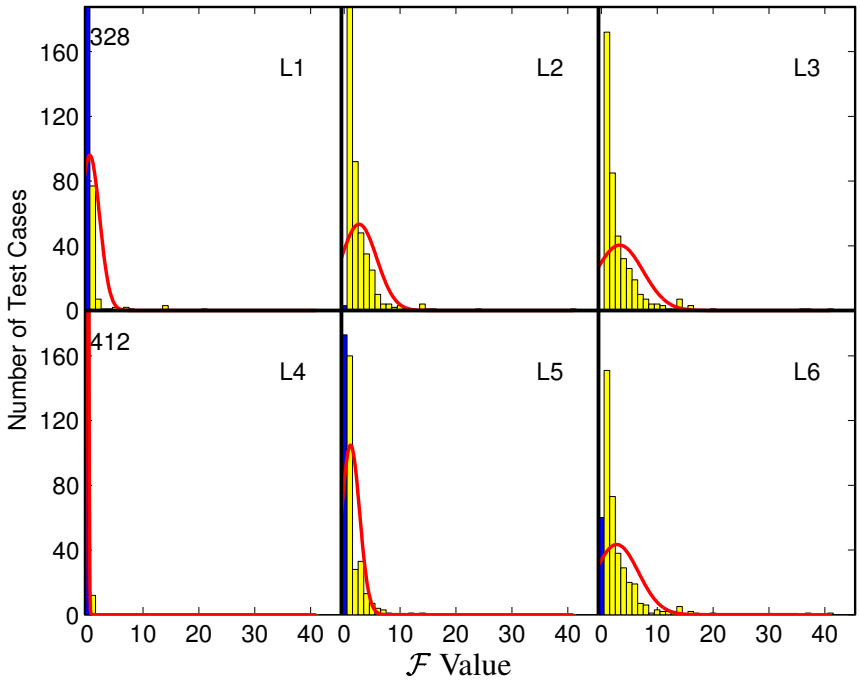


Figure 5.6: Histogram for TerpWord

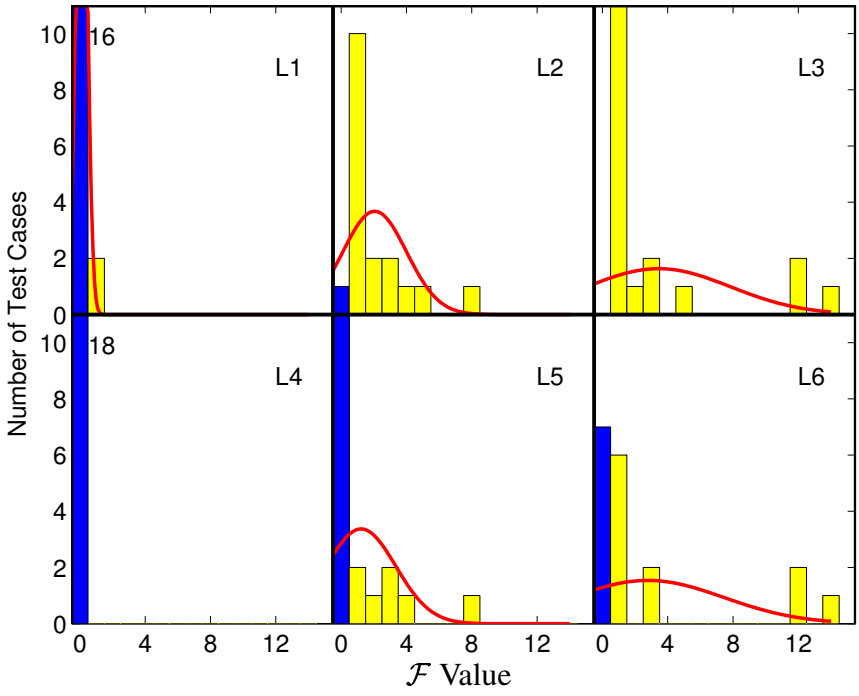


Figure 5.7: Histogram for TerpPaint

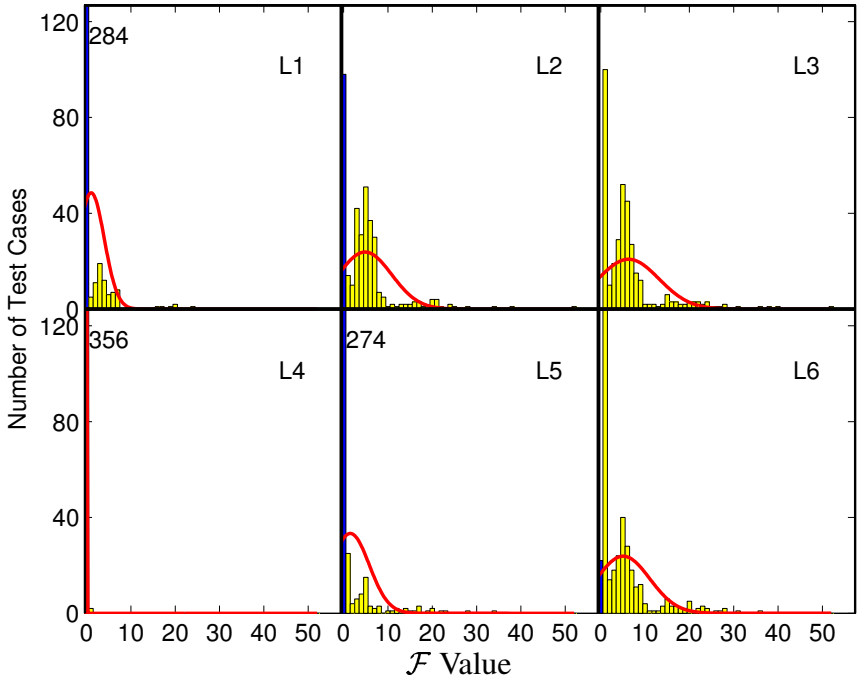


Figure 5.8: Histogram for TerpSpreadSheet

Statistical Analysis

The results discussed thus far have been based on visual examination of the data. While visual examination provides an intuitive overview of the data, valuable information is lost. For example, each test case has six data-points (the six $\mathcal{F}$ values) that are correlated. This correlation is difficult to show and compare visually, especially for large data-sets.

Whether the differences in $\mathcal{F}$ values observed for each test case per test oracle are statistically significant has to be determined. In particular, the differences between the oracles within the sets {L1, L2, L3}, {L4, L5, L6}, {L1, L4}, {L2, L5}, and {L3, L6} need to be studied. Several statistical tests may be used for this study. Choosing the right test is based on the number and the nature of the dependent (in this case the $\mathcal{F}$ values) and the independent variables (*i.e.*, the test oracle). For this experiment, the distribution of the data (Normal vs. non-Normal), the number of groups (2 or 3), size of groups, and whether the groups are matched or not will be considered.

Since the sample sizes are small (*e.g.*, 18 for TerpPaint), normality of the data has to be determined before the statistical tests are chosen. For illustration, the solid line superimposed on the histograms (Figures 5.5 through 5.8) shows the normal distribution approximation; this illustration suggests that the data is not normal. Finally, the data is matched, *i.e.*, each data point (*e.g.*, $\mathcal{F}$ value for oracle L1 with test case T) in one distribution (for oracle L1) has a corresponding matched point in all other distributions (the matched points are the $\mathcal{F}$ values for oracles L2–L6 with test case T). Considering all these factors, the Friedman test was chosen for the three matched groups statistical

comparison ({L1, L2, L3}, {L4, L5, L6}) and Wilcoxon signed ranks test for two matched groups comparison ({L1, L4}, {L2, L5}, {L3, L6}). There is no test to compare {L1, L4} for TerpPaint.

	Sample Size	Friedman Test	Statistic Value	P-Value
TerpPresent	1800	L1/L2/L3	1095.2514	<.0001
		L4/L5/L6	1174.8991	<.0001
TerpWord	1272	L1/L2/L3	710.9736	<.0001
		L4/L5/L6	577.6345	<.0001
TerpPaint	54	L1/L2/L3	31.0000	<.0001
		L4/L5/L6	18.0000	0.0001
TerpSpreadSheet	1074	L1/L2/L3	542.5014	<.0001
		L4/L5/L6	598.8733	<.0001

Table 5.2: Friedman Test Results

Friedman Test: This test compares the mean $\mathcal{F}$ values for the test oracle sets {L1, L2, L3}, and {L4, L5, L6} based on their rank scores. The null hypothesis here is that the mean values do not differ. Table 5.2 summarizes the results of this test. The statistic value shown here is the standard Cochran-Mantel-Haenszel (CMH) statistic used by most popular statistical software packages. The p-values are obtained by a table lookup using the sample size and CMH value. As the result shows, all p-values are less than 0.05. Hence, the null hypothesis is rejected. The alternative hypothesis, *i.e.*, the mean $\mathcal{F}$ values do differ in a statistically significant way, is accepted. An additional Wilcoxon matched pairs test on the oracle pairs {L1, L2}, {L2, L3}, {L1, L3}, {L4, L5}, {L5, L6}, and {L4, L6} showed that the differences between these oracle pairs are also statistically significant.

Wilcoxon Signed Ranks Test: The null hypothesis here is that there is no statistically significant difference between the means among the oracles in the sets {L1, L4}, {L2,

	Sample Size	Wilcoxon Test	Statistic Value	P-Value
TerpPresent	**600**	L1/L4	20808	<.0001
		L2/L5	33764	<.0001
		L3/L6	115.5	<.0001
TerpWord	**424**	L1/L4	1785	<.0001
		L2/L5	17490	<.0001
		L3/L6	6440	<.0001
TerpPaint	**18**	L1/L4	*	*
		L2/L5	27.5	0.0020
		L3/L6	14	0.0156
TerpSpreadSheet	**358**	L1/L4	1387.5	<.0001
		L2/L5	10764	<.0001
		L3/L6	1870.5	<.0001

Table 5.3: Wilcoxon Test Results

L5$\}$, and $\{$L3, L6$\}$. The results of the tests are summarized in Table 5.3. All p-values are less than 0.05, resulting in the rejection of the null hypothesis and acceptance of the alternative hypothesis.

The above two analyses helped to answer the first parts of Q1 and Q2. Based on the results of the Friedman test, and the earlier visual comparison, it is concluded that the oracle information has a significant impact on fault detection effectiveness of a test case; checking more widgets is beneficial. Based on the results of the Wilcoxon signed ranks test, and the earlier visual observations, it is concluded that the frequency of invoking the test oracle does have a significant impact on the fault detection effectiveness of a test case; invoking the test oracle frequently is beneficial.

Faults Detected Per Comparison

To study the cost of the oracles, first, the number of comparisons that each oracle performs per test case should be computed. The average number of comparisons per test case for oracle L is represented as $\Delta(L)$, and is shown in Table 5.4. As expected,

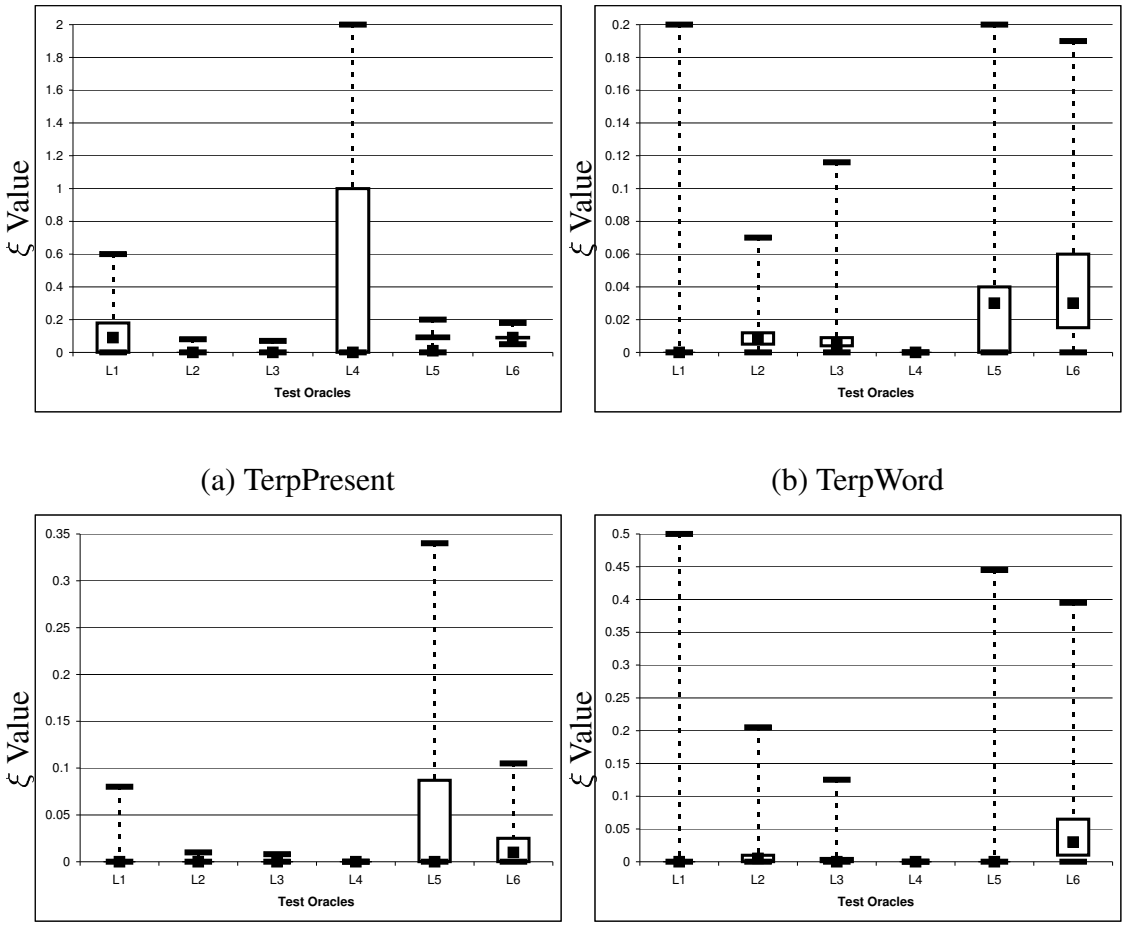
	L1	L2	L3	L4	L5	L6
TerpPresent	11.5	567.69	947.6	1	48.51	82.56
TerpWord	12.93	404.79	671.56	1	30.29	59.56
TerpPaint	14.17	1106.56	1667.17	1	63.33	125.83
TerpSpreadSheet	13.61	600.29	1268.06	1	43.24	104.7

Table 5.4: Average Number of Widget Comparisons Per Test Case

$\Delta(L4) = 1$. The value of $\Delta(L1)$ is larger than $\Delta(L4)$ due to one comparison per event in the test case. The values of $\Delta(L2)$ and $\Delta(L3)$ depend on the number of widgets in the active window and in all the open windows respectively. Similarly, $\Delta(L5)$ and $\Delta(L6)$ depend on the number of widgets in the active window and in all the open windows when the test case ends, respectively.

Recall that ξ has been defined as the faults-detected-per-comparison for each test case. Higher values of ξ are considered better. ξ is computed and the results are presented as box-plots. The results for TerpPresent are summarized in Figure 5.9(a). Since L3 requires the maximum number of comparisons (the entire state of the GUI after each event in the test case), it is penalized the most by the ξ measure; L2 is close behind. Since the number of comparisons is smaller for L5 and L6, their ξ values are better. In the case of TerpPresent, checking the widget alone helped to detect a non-trivial number of faults; combined with a very small number of comparisons required, the ξ value of L4 was better than all other oracles, followed by L1.

The results for TerpWord (Figure 5.9(b)) are different primarily because L1 and L4 did not detect many faults; simply checking the widget was inadequate. L2 and L3 again suffered due to the large number of comparisons they require. L5 did much better due to its reduced frequency of comparison. Although L6 compares the entire state whereas



(a) TerpPresent

(b) TerpWord

(c) TerpPaint

(d) TerpSpreadSheet

Figure 5.9: ξ Values for All Test Cases

L5 compares only the active window, L6 did much better due to its larger $\mathcal{F}$ value. This difference did not help L6 for TerpPaint (Figure 5.9(c)) since the entire state is much larger for this application. Since L5 did not detect many faults for TerpSpreadSheet, its ξ value is very low (Figure 5.9(d)).

Answers to Q1, Q2, and Q3 are now ready. In case of Q1, it was noted that the oracle information does have an impact on the fault-detection effectiveness of a test case. In case of Q2, the invocation frequency of a test oracle has a very significant impact on

the fault-detection effectiveness of a test case. Considering the ξ measure, the additional effectiveness is not worth the cost for L2 and L3 due to the extremely large number of comparisons required for L2 and L3; using L5 and L6 is more practical. However, for L1 vs. L4, the additional cost is very low and helps fault detection.

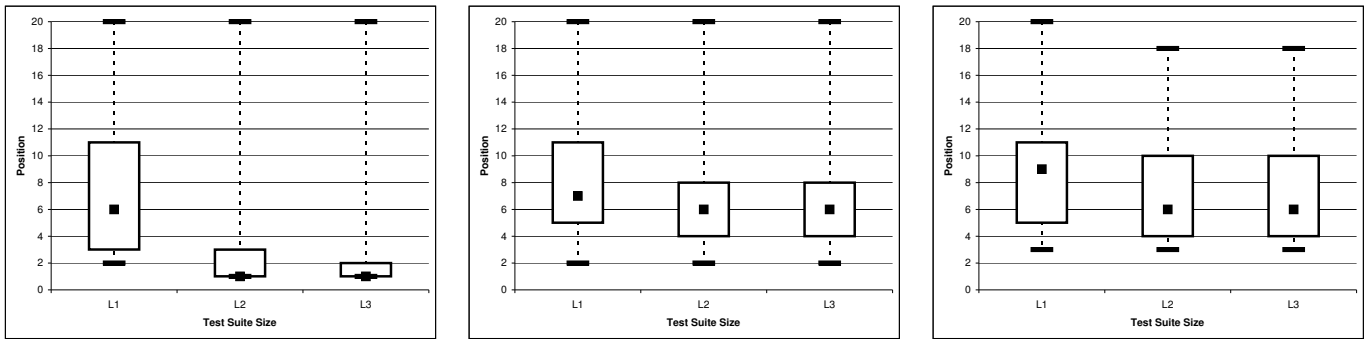
In case of Q3, the combination of oracle information and procedure that provides the best cost-benefit ratio depends largely on the GUI.

Relationship Between Test Oracles and Fault-Detection Position

It was observed that whenever the test oracles L3, L2, and L1 detected a fault at event position a, b, and c respectively, then in many cases (*e.g.*, 33% for TerpWord, 62% for TerpPresent) one of the relationships $a < b$ or $b < c$ held ($a = b = c$ was expected). In other words, when oracles contained more information, they tended to detect faults earlier in the event sequence.

$$\textcircled{e_1} \rightarrow \textcircled{e_2} \rightarrow \ldots \rightarrow \underbrace{\textcircled{e_a}}_{L3} \rightarrow \ldots \rightarrow \underbrace{\textcircled{e_b}}_{L2} \rightarrow \ldots \rightarrow \underbrace{\textcircled{e_c}}_{L1} \rightarrow \ldots \rightarrow \textcircled{e_n}$$

This was an interesting result since it provided a link between test oracles and the length of a test case. Longer test cases are more expensive to generate and execute. Hence, if a test designer has a suite containing short test cases, oracle L3 has better chances of detecting more faults. The box-plots shown in Figure 5.10 illustrate the results. No results are shown for TerpPaint since only two test cases detected a fault using L1. The box-plots show that the position at which the fault is detected using L1 is later than that using L2 or L3. However, for TerpWord and TerpSpreadSheet, the position at which the fault is detected using L2 is almost the same as that using L3.



(a) (b) (c)

Figure 5.10: Position Where the Fault is Detected vs. Oracle for (a) TerpPresent, (b) TerpWord, and (c) TerpSpreadSheet

Hence generating/using a complex (more expensive) oracle is justified if a tester has short test cases. This result partly answers Q4.

5.3 Conclusions

This chapter presented a new GUI testing process called *smoke testing*, which tests a set of changes that have been made by developers. A more thorough feedback is provided to them. This chapter shows the smoke test cases are effective at detecting a large number of GUI faults. Because smoke testing is a form of reference testing, which tests the current version against its previous version, the challenge of performing smoke testing lies in creating test oracles.

Two important parts of a test oracle were defined: *oracle information* that represents expected output and an *oracle procedure* that compares the oracle information with the actual output. A technique to specify different types of test oracles was developed by varying the level of detail of oracle information and changing the oracle procedure; this

technique was used to create six instances of test oracles for an experiment. The results of the experiment showed that test oracles do affect the fault-detection ability of test cases in different and interesting ways: (1) test cases significantly lose their fault-detection ability when using "weak" test oracles, (2) in many cases, invoking a "thorough" oracle at the end of test case execution yields the best cost-benefit ratio, (3) certain test cases detect faults only if the oracle is invoked during a small "window of opportunity" during test execution, and (4) using thorough and frequently-executing test oracles can make up for not having long test cases.

The smoke test cases exhibited the following properties that satisfy the criteria for the intermediate loop (presented in Chapter 3).

1. The test cases can be generated automatically and executed in one night.

2. All system-interaction and termination events are executed; most of the GUI's functionality is covered.

3. As the GUI is modified, many of the smoke test cases remain usable because they do not contain structural events.

4. Test oracle based on reference testing is fully automatic.

Note that in smoke testing, false positive issues may arise, because some of the changes are intentionally made. The testers have to identify those false positives manually.

Chapter 6

Comprehensive GUI Testing

Although smoke and crash testing are useful in that they help to detect major problems in the GUI software, comprehensive GUI testing goes beyond looking for software crashes and reference testing; it should be performed before the software is released to its end-users. As discussed in Chapter 2, several researchers have developed limited and expensive techniques to automate this type of testing; however, in practice, in most organizations, GUI testing continues to be performed manually with limited tool support (JUnit, Capture/Replay). This is due to a number of reasons including testers' expertise and established practices within the organization. Recognizing that these factors are difficult to change, the goal of this chapter is not to develop new techniques for comprehensive GUI testing; rather it is to provide a set of guidelines supported by results of experiments that a test designer may use to improve techniques that are already in use.

In particular, two sets of guidelines are presented. The first set is based on an experiment that studies tradeoffs between test case length, test suite size, and event composition. The second set is based on another experiment that helps to determine strategic points in the test case where a tester may insert an assertion (for the test oracle) to maximize fault-detection effectiveness and reduce cost.

6.1 Experiment - Studying the Characteristics of a "Good" Comprehensive Test Suite

In this experiment, several key characteristics of GUI test suites of interest to testers are varied: size of the suite, event composition, and the length of each test case. For each combination of these characteristics, the impact on fault detection effectiveness and cost is reported. The goal is to compile a set of "lessons learned" that can be used by testers to create effective GUI test cases for comprehensive testing.

6.1.1 Experimentation Procedure

This experiment has been designed to examine two hypotheses: (H_1) large test suites are more effective at detecting faults compared to smaller test suites, (H_2) test suites that contain long GUI test cases are more effective at detecting faults compared to test suites that contain only short GUI test cases. The experiment will prove or disprove, via hypothesis testing, the set (H_0) of null hypotheses: $\{$(H_{01}) increasing the size of a test suite does not correspondingly increase the fault-detection effectiveness and generation/execution cost of the suite, (H_{02}) increasing the length of a test suite's constituent test cases does not correspondingly increase the fault detection effectiveness and generation/execution cost of the suite$\}$. The alternative hypothesis will be the negation of the corresponding null hypothesis.

Four of the TerpOffice applications and their fault-seeded versions (TerpWord, TerpSpreadSheet, TerpPaint, and TerpCalc) are selected. A large number of test suites with various, carefully controlled characteristics are created and executed on these applica-

tions. Keeping in mind the above hypotheses, the primary measured variable is the fault-detection effectiveness of a test suite; the secondary measured variable is the cost of generating and executing the test suite.

6.1.2 Test Pool

This experiment requires the development and execution of a large number of test cases. For example, Part 1 of the experiment requires the execution of 9000 test suites, each with an average size of 2780 test cases for one subject application. GUI test cases are expensive to execute – each test case can take 5-60 seconds to execute. Hence, for the results to be statistically significant, the experiment must generate and execute a prohibitively large number of test suites. Other researchers, who have also encountered similar issues of practicality, have circumvented this problem by creating a *test pool* consisting of a large number of test cases that can be executed in a reasonable amount of time [10]. Each test case in the pool is executed only once and it's execution attributes *e.g.*, time to execute and faults detected are recorded. Multiple test suites are created by carefully selecting test cases from this pool. Their execution is "simulated" by combining the attributes of constituent test cases using appropriate functions (*e.g.*, *summation* for cost of execution). This research will also employ the test pool approach to create a large number of test suites.

Due to its central role in this experiment, it is important to create the test pool carefully. The test pool should allow the creation of test suites with three controllable attributes, namely size, length of the constituent test cases, and the event composition of

the suite. For example, for Part 2 of the experiment, the test pool should allow the creation of test suites containing test cases that vary in length; at the same time, the size and event composition of the suites should remain constant. Hence, the test cases used in earlier experiments cannot be used here. However, it should be noted that the threats to validity stated in Section 4.4 also hold for this experiment.

The following process was employed to create the test pool:

1. Create twenty empty buckets; each $bucket_i$ can hold test cases of length i, for $1 \leq i \leq 20$.
2. Add all GUI events into $bucket_1$. Each event forms a length 1 test case.
3. For each event x in $bucket_1$, create five[1] copies of x and append each copy to a randomly chosen (without replacement) element from `follows(x)`. The "without replacement" choice ensures that the test cases are unique. For all events, except for the `Exit` event, $|\texttt{follows(x)}| > 5$; the `Exit` event is ignored in this experiment. The result is a set of unique length 2 test cases, which forms $bucket_2$.
4. To fill $bucket_i$ ($3 \leq i \leq 20$): for each event x in $bucket_1$, create 5 copies of x and concatenate each copy with a randomly chosen (without replacement) element from `follows(x)`. Increase the length of this test case to i by repeating the concatenation process, selecting a random event each time.
5. The test pool is the Union of $bucket_2$ through $bucket_{20}$. Note that bucket$_1$ is ignored due to its smaller (one-fifth) size.

[1]The choice of five copies is not arbitrary. This experiment was conducted with 2, 3 and 4 copies. There was no significant difference in results between 4 and 5 copies. Hence for these applications, the results of experiments that used 5 copies were reported.

All the buckets are of equal size; they have $5 \times N$ test cases, where N is the number of events (minus 1 for `Exit`) in the GUI. The test pool for each application contained 11875, 15010, 18240, and 7980 test cases for TerpWord, TerpSpreadSheet, TerpPaint, and TerpCalc respectively. Each bucket is guaranteed to contain at least 5 instances of each GUI event (as the first event in the test case). Each test case will be executed in the same initial state of the GUI. Hence, these 5 events will behave identically. As expected, the exact number of times each event was executed was much larger than 5. The event frequency distribution is shown in Figure 6.1 in the form of box-plots. Note that some events, those that open pull-down menus, are executed much more frequently (*e.g.*, as much as 3500 times in TerpSpreadSheet) than others.

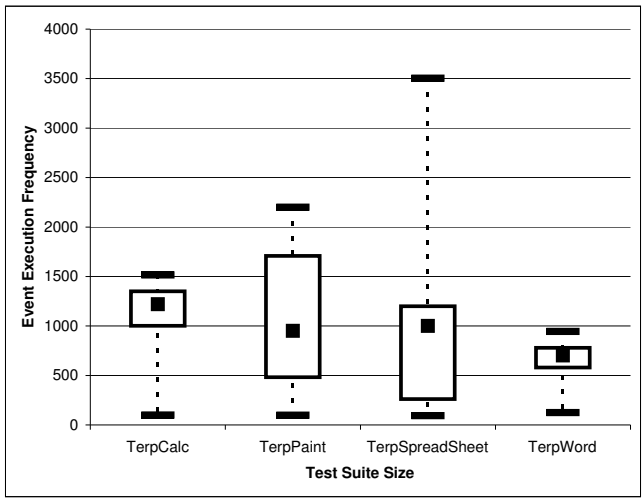


Figure 6.1: Event Distribution for Each Application

Two variables were measured in these experiments for each test suite, *i.e.*, cost in terms of execution time and fault-detection effectiveness. Execution time of the test suite was simply the cost of executing each test case in the suite. The fault-detection effectiveness was measured as the number of unique faults detected by the test cases in

the suite.

6.1.3 Part 1: Effect of Test Suite Size

Since several factors (test suite size, event composition, test-case length) may have an impact on the fault-detection effectiveness of a test suite, one factor will be varied in each part of the experiment, keeping other factors constant. In this part of the experiment, the event composition and length of test cases will be kept constant; only the test suite size will be varied.

To find the minimum test suite size that may be used for this experiment, the following process is executed:

1. For each application, randomly generate 100 test suites. Each test suite should cover all GUI events (*i.e.*, randomly select test cases without replacement from the test pool until all the GUI events have been covered). Measure the size of each suite; add these 100 values (sizes) to an *initial observation set* OS_0.
2. Randomly generate 100 more test suites. Add them to the most recent observation set OS_i. Determine if OS_i is *equivalent* to the old observation set OS_{i-1}; if so, then skip to the next step; else repeat this step. Equivalence is determined by the formula: $(Median(OS_i) == Median(OS_{i-1})$ && $Q_1(OS_i) == Q_1(OS_{i-1})$ && $Q_3(OS_i) == Q_3(OS_{i-1}))$, where $Median$, Q_1, and Q_3 are the median, first quartile, and third quartile of a data set respectively. This step terminates only if the formula returns `TRUE`.
3. The above step executed 11, 14, 11, and 11 times respectively for TerpCalc, Terp-

Paint, TerpSpreadSheet, and TerpWord before terminating.

The median of the last observation set is the smallest test suite size (n) that is considered in this experiment. The median for TerpCalc, TerpSpreadsheet, TerpPaint and TerpWord is 220, and 377, 556, and 170 respectively. The test suite size will be varied from n to $10 \times n$ test cases, in increments of n.

Since the test suites for this experiment need to have the same event composition and lengths of test cases, the following process is used to create them:

1. Create a test suite of size n by randomly choosing (without replacement) n elements from the test pool. If all the events in the GUI are not covered by this test suite, then discard the suite and re-execute this step. Create 19 partitions of this suite by test-case length. If for any length i, $partition_i > (bucket_i/10)$, then discard the suite (since it cannot be used to create the size-$10n$ suite in Step 3 below) and re-execute this step.
2. For each test case t in the size-n suite do: let the length of t be x; randomly select from the test pool, without replacement, 2 test cases of length x. Insert them into the size $2 \times n$ suite. Random choice without replacement throughout this step's execution ensures that there are no duplicate test cases in the suite. If all the events in the GUI are not covered by the $2 \times n$ test suite, then discard the suite and re-execute this step.
3. Repeat the above step for size $3 \times n$ through size $10 \times n$, choosing 3 through 10 test cases respectively from the test pool for each element of the size-n test suite.

The event composition of all the suites is exactly the same. Also, they all have

similar-length test cases. This process of test suite creation is repeated in increments of 100 test suites per unit of size until the data converges, *i.e.*, additional runs do not yield useful information. If the data has not converged yet, the latest 100 data points are added to the observation set; hence the observation set grows in increments of 100. Convergence is determined using the three-value (median, first quartile, third quartile) comparison process described earlier. The only difference is that all 10 same-sized sets are compared to each other.

The number of increments for TerpWord, TerpSpreadSheet, TerpCalc, and TerpPaint was 10, 8, 7, and 9 respectively, representing 1000, 800, 700, and 900 test suites in the final observation set. Figure 6.2 through Figure 6.5 summarizes the results for TerpCalc, TerpWord, TerpSpreadSheet, and TerpPaint. These figures show a trend that the number of faults detected grows as test suite size grows, *i.e.*, larger suites are more effective at detecting faults. The convergence towards a plateau above a size roughly corresponding to 1000 is an artifact of the number of faults seeded and/or the size of the GUI.

The analysis of variance test (ANOVA) with $\alpha = 0.05$ was performed to show that the differences of fault-detection for test suite size are statistically significant. The "factor" in the ANOVA was the test suite size and the "response" was the fault-detection effectiveness. The ANOVA test would indicate, with a certain degree of confidence, that the observed differences were statistically significant. The observed p-value was 3.3×10^{-154}, much less than 0.05, leading to the conclusion that the suite size has a statistically significant impact on the fault-detection effectiveness. Hence the null hypothesis H_{01} is rejected.

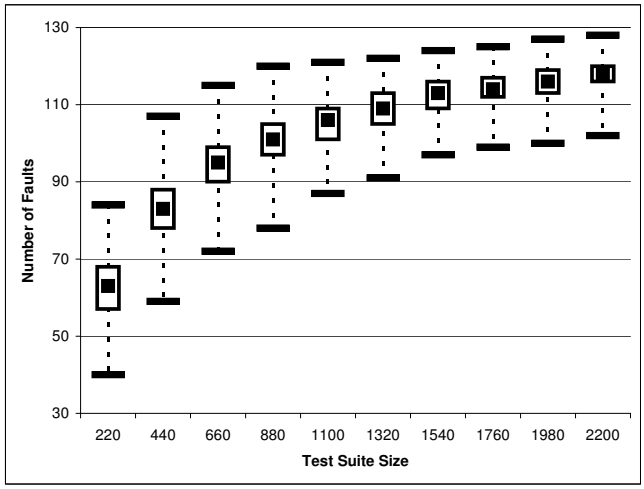


Figure 6.2: Fault Detection Effectiveness vs. Test Suite Size for TerpCalc

This result shows that the size of a test suite improves its fault-detection ability even though the lengths of its constituent test cases and event composition do not change. The only difference in larger test suites is that events are executed multiple number of times in combination with different preceding events (different GUI states), *i.e.*, increased diversity of GUI states. A larger test suite, however, requires more time to generate as well as execute; the time is proportional to the size of the suite.

6.1.4 Part 2: Effect of Test Case Length

This part of the experiment will study the effect of test case length on fault-detection effectiveness of a test suite, keeping event composition and size constant. The following process was used to obtain the test suites.

1. To create a test suite containing test cases of length i: randomly choose (without replacement) test cases from bucket$_i$ until all events have been covered. Execute

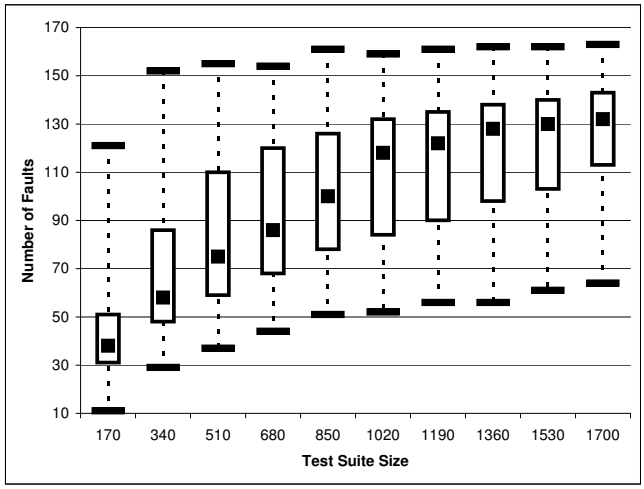


Figure 6.3: Fault Detection Effectiveness vs. Test Suite Size for TerpWord

this step for $2 \leq i \leq 20$.

2. Let N be the size of the largest of the 19 test suites. Add test cases into the remaining test suites from their corresponding buckets until they have N test cases. Ensure that no test cases are repeated.

Evaluate the fault-detection effectiveness of the 19 test suites. Repeat the above process using the three-value comparison technique outlined in Part 1 of this experiment. The data distributions converged after 10, 11, 12, and 12 iterations for TerpWord, TerpSpreadSheet, TerpCalc, and TerpPaint respectively, representing 1000, 1100, 1200, and 1200 data points in the final observation set.

The results are summarized in Figure 6.6 through Figure 6.9. The x-axis shows the test case length (2-20) and the y-axis shows the fault-detection effectiveness. The results show that the fault-detection effectiveness does not increase with test-case length. There is not the slightest evidence against the null hypothesis H_{02}; hence it cannot be rejected.

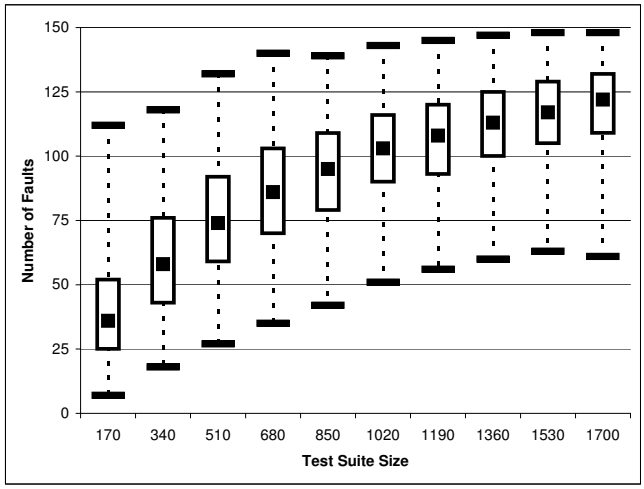


Figure 6.4: Fault Detection Effectiveness vs. Test Suite Size for TerpSpreadsheet

Although the results show that the length of test cases has no significant impact on the *number* of faults detected, additional analysis showed that there were certain faults that could only be detected by long test cases; short test cases did not detect these faults. The analysis results for TerpCalc are summarized in Figure 6.10. The figure shows a column graph; the x-axis shows the test case length; for each column i, the height of the column shows the size of the set $Complement(Faults(i), Union_{j=1}^{i-1} Faults(j))$, where $Faults(x)$ is the set of faults detected by all length-x test cases in the test pool, $Union$ and $Complement$ are set operators. For example, the graph shows that length 10 test cases detected 12 new faults that could not be detected by any of length 1 through length 9 test cases. The number of new faults decreases for very long test cases. For example, length 16, 17, and 18 test cases did not detect any faults that had not been detected by shorter ($<$ 16) test cases. Length 19 and 20 test cases detected only 3 and 2 new faults respectively. The converse of this result was not true, *i.e.*, $Complement(Faults(i), Union_{j>i} Faults(j))$ was almost always the empty set.

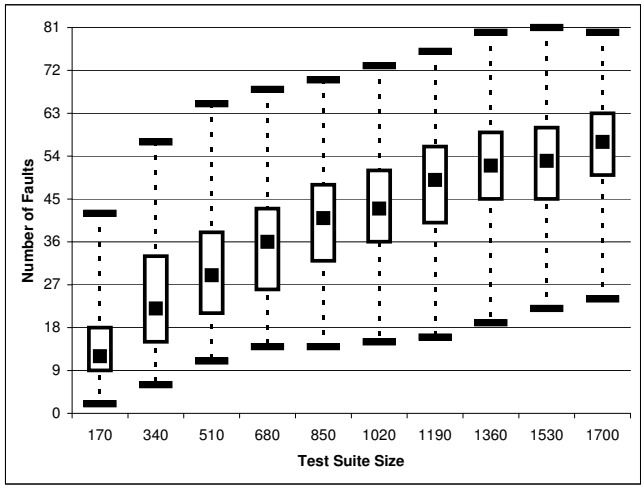


Figure 6.5: Fault Detection Effectiveness vs. Test Suite Size for TerpPaint

This experiment showed that when test suite size is kept constant, the length of the test cases has an impact on the type (not number) of faults detected. This result reinforces the earlier observation that an event, when executed in multiple contexts, detects different faults. A tester has two ways of improving diversity in the way an event is executed: (1) by creating longer test cases and (2) generating more test cases as observed from Section 6.1.3.

6.1.5 Part 3: Effect of Event Composition

In the first two parts of this experiment, the event composition of the test suites was kept constant, *i.e.*, all the events were used. This part of the experiment keeps the test suite size and test-case length constant and varies the event composition. The following process was used to create the test suites.

1. Randomly generate a test suite t that covers all events.

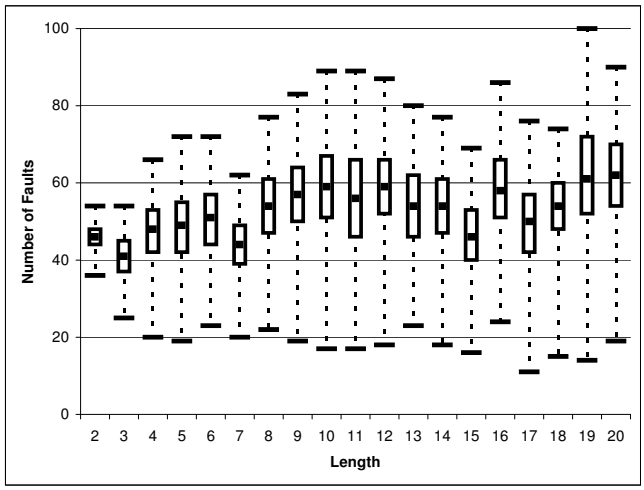


Figure 6.6: Fault Detection Effectiveness vs. Test Case Length for TerpCalc

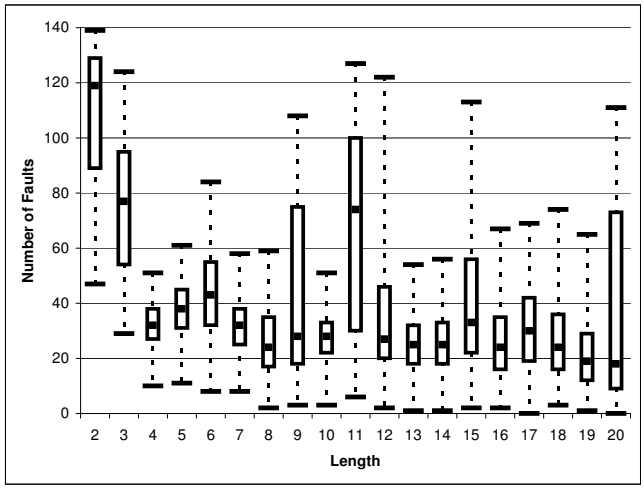


Figure 6.7: Fault Detection Effectiveness vs. Test Case Length for TerpWord

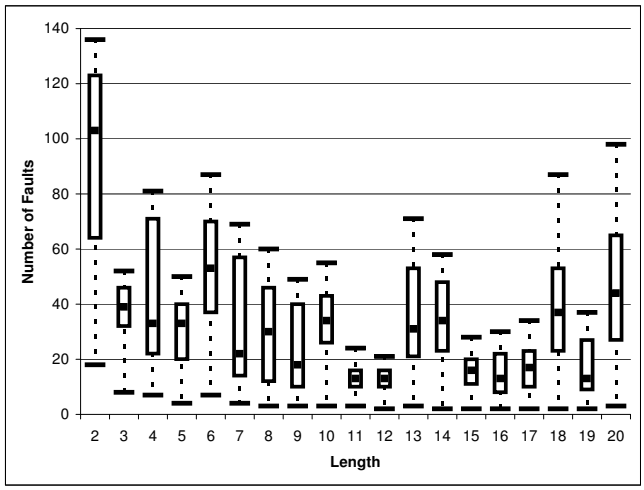


Figure 6.8: Fault Detection Effectiveness vs. Test Case Length for TerpSpreadsheet

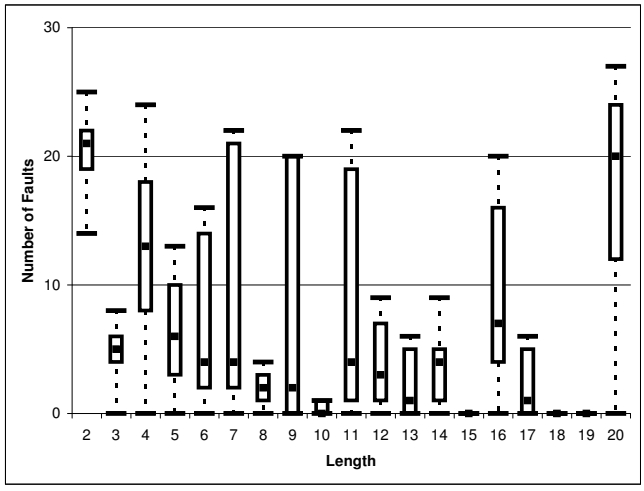


Figure 6.9: Fault Detection Effectiveness vs. Test Case Length for TerpPaint

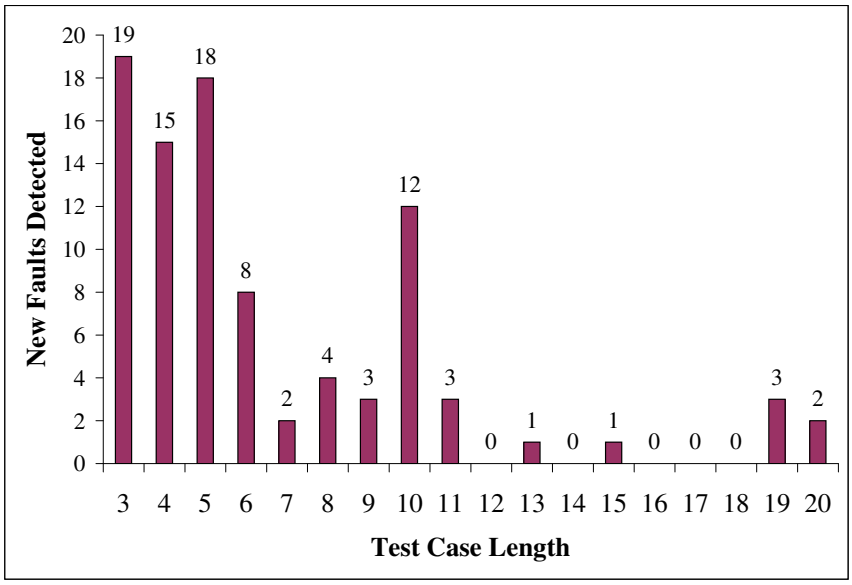


Figure 6.10: New Faults Detected with Length Increase

2. For each event x in the GUI, obtain a test suite called non-x, which is identical to t in that it has test cases of similar lengths and is of the same size. However, non-x does not contain any test case that uses event x. The following process is used to obtain non-x: copy those test cases from t to non-x that do not contain x. For each of the remaining test cases, choose from the test pool a test case of the same length but one that does not contain x and that maximizes the chances of covering other events that are not in non-x. If all same-length test cases are exhausted, then discard t and repeat Step 1. Also, if the final test suite does not cover all events (except x) then discard t and repeat Step 1.
3. Determine the fault-detection effectiveness of the generated test suites. Repeat the above process using the three-value comparison technique outlined in Part 1.

Some events (ones that open pull-down menus, *e.g.*, `File`) are used very frequently (as much as 3500 times) in the test pool. Removing such events caused problems with the

above steps; for example, when `File` was removed, it was impossible to create a suite that covered all other events in the GUI. Fortunately, none of these pull-down menu opening events contributed to the fault-detection of the test cases; it was hence not necessary to remove them.

There was a strong correlation between faults detected by some of the of test suites and the functional unit in which faults were seeded. A classification of events done using the same functional units as the ones used for code, revealed that in all cases, non-i test suites (for $i \in F$ functional unit class), the suite did not detect any faults seeded in functional unit F. Hence, the absence of an event (that interacted with a functional unit F) in a test suite directly effects the detection of a fault that was seeded in F's code.

This experiment showed that a test suite that uses a wide diversity of states in which an event executes has good fault-detection effectiveness. There are two ways to improve state diversity – increasing test case length and creating larger test suite size. A tester should allocate maximum resources to finding the majority of bugs that can be detected by generating a large number of short test cases in multiple combination of events. Additional resources may be used to find the relatively fewer bugs that can be detected by generating long test cases.

6.2 Experiment - Developing Test Oracles for Comprehensive Testing

Chapter 5 discussed six types of test oracles and showed that they have a significant impact on fault detection effectiveness and cost. Two types of invocation frequency of oracle procedure were defined: (1) "after each event" for oracles L1, L2, and L3, and

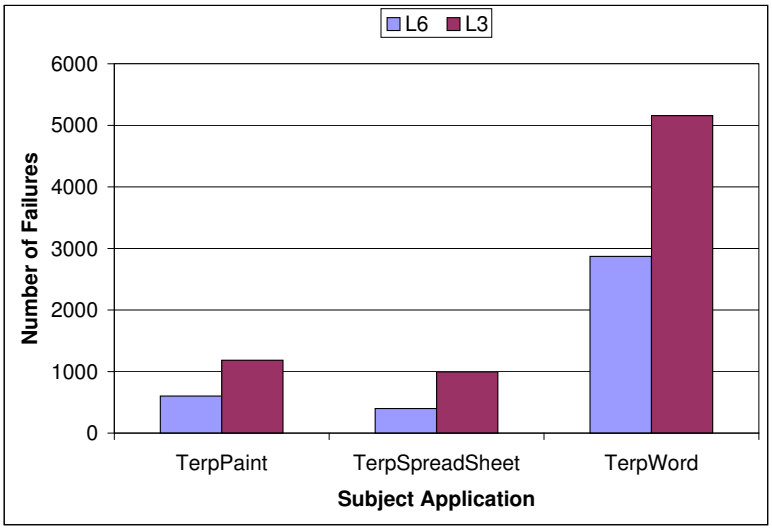


Figure 6.11: Number of Failures

(2) "after the last event" for L4, L5, and L6. Because comprehensive testing may use several manual techniques (*e.g.*, Capture/Replay tools), it is too expensive to specify and check the GUI state information "after each event" in the test case. On the other hand, specifying and checking state information "after the last event" in the test case is relatively cheaper, but as Section 5.2 showed, it may miss faults. The experiment presented here helps to identify strategic points in the test case at which assertions may be inserted and comparisons may be done to maximize fault detection effectiveness and minimize cost. It should be noted that the threats to validity stated in Section 4.4 also hold for this experiment.

In this experiment, the oracle information will be the "entire GUI state"; the oracle procedures will be (1) "check for `equality` of the oracle information and actual output after each event" and (2) "check for `equality` of oracle information and actual output after the last event" of the test case. Note that this corresponds to oracles L3 and L6. Test cases already available for the experiment of Section 6.1 were rerun with these two types

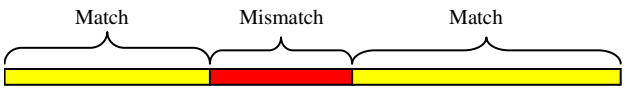
of oracles. The number of failed test cases is shown in Figure 6.11. This graph shows that certain types of GUI errors were missed when using L6. These errors will be called *transient errors* because they "disappear" before the oracle L6 is invoked.

- **Definition:** A *transient GUI error* occurs during execution of a test case, if $A_i \neq S_i$, for an event e_i in the test case, where S_i is the oracle information and A_i is the actual state and $A_j == S_j$, for some $i < j \leq n$. □

On the other hand, many mismatches persist until the last event in the test case. More formally, persistent errors are defined as:

- **Definition:** A *persistent GUI error* occurs during execution of a test case, if $A_i \neq S_i$, for all events e_i in a test case of length n ($j \leq i \leq n$ for some j). □

The parts of test cases that caused the expected and actual states to mismatch are shown in Figure 6.12 through Figure 6.13. Figure 6.12 shows the results for TerpPaint. The x-axis shows the event number (*i.e.*, its position in the sequence) in the test case. The y-axis represents failed test cases. For each test case, there is a line, with 2 levels of shading. The dark band shows the events after which the actual and expected states mismatched. The light band shows the event number after which the actual and expected states matched:



If a test case failed on more than one fault-seeded version, it is counted more than once. The test cases were sorted carefully to show the dark/light bands clearly. Note that

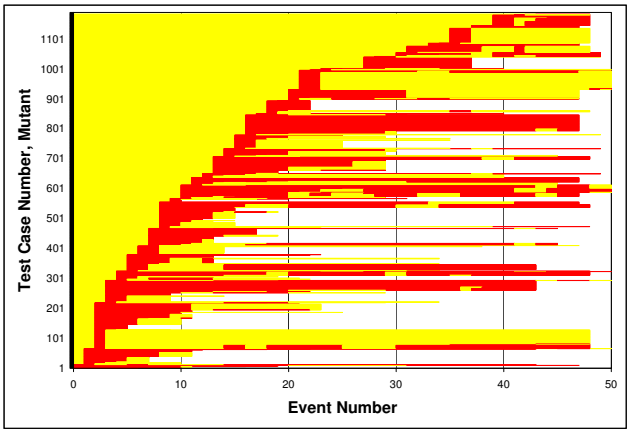


Figure 6.12: Errors for TerpPaint

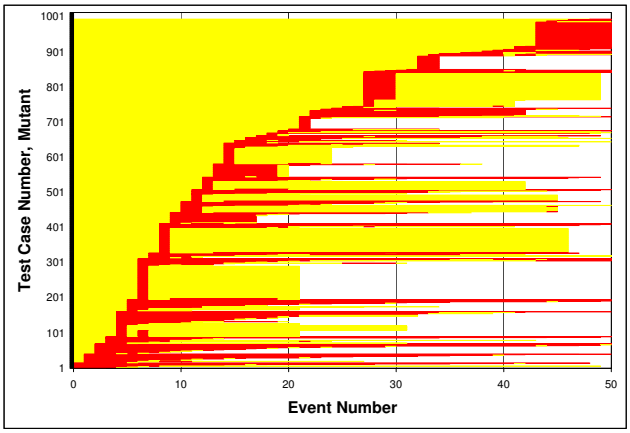


Figure 6.13: Errors for TerpSpreadSheet

many test cases have small areas of mismatch. In all cases where the test case ends in a light band (*i.e.*, a match) , the test oracle L6 would have failed to report an error. Similar results are seen for TerpSpreadSheet and TerpWord in Figures 6.13 and 6.14 respectively.

Having observed that long test cases, during execution, can transit frequently between matching and mismatching, the results were manually examined to identify classes of events that caused the transitions. Some of these classes were defined in Section 4.3; a new class used in this analysis called menu-open events is used to open/close pull-down menus. The execution data was mined to find events that led from a match to a mismatch

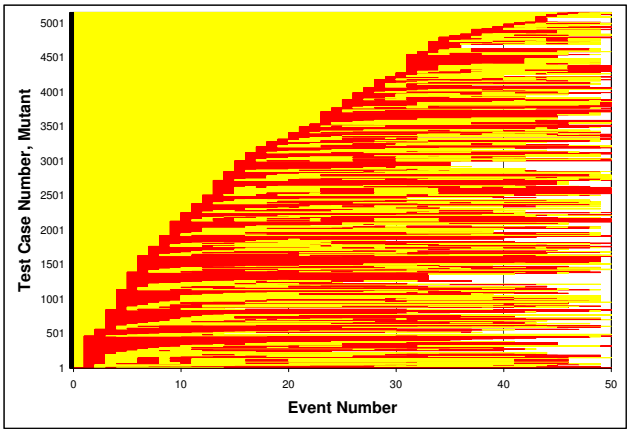


Figure 6.14: Errors for TerpWord

and vise versa. The results are summarized in Figure 6.15. The figure shows a column graph with event types on the x-axis. The y-axis shows the number of times an event type led to a transition. As seen in the graph, termination, window-open, and system-interaction events cause the maximum number of transitions. A test oracle that compares the expected and actual states of the GUI at these events is most likely to report transient errors.

Manual examination of the test case showed that object creation and destruction play an important role in transient errors. Examples of such objects for GUIs include windows, menus, widgets, etc. A window-open event that opens a erroneous window will cause an error to be detected by the test oracle. On the other hand, a termination event that destroys a window will close the erroneous window, resulting in a match between expected and actual states.

The analysis was used to create a new test oracle called O_{new} by modifying L6. Oracle information was generated for termination and window-open events. The oracle procedure was modified to compare the expected and actual states at these points. System-

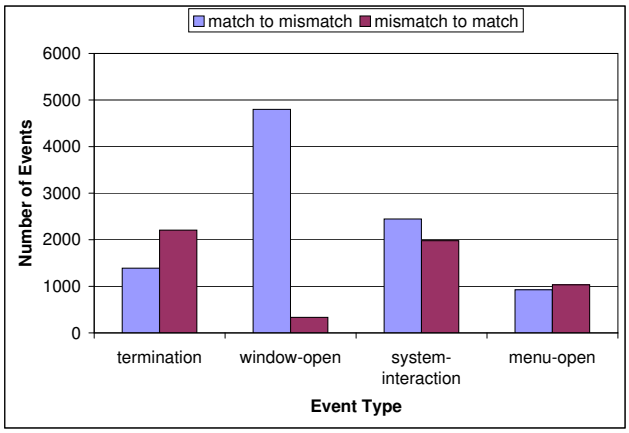


Figure 6.15: Event Classes and Error Types

interaction events were not chosen because the test cases contain a large number of these events, *i.e.*, had the oracle for system-interaction events been compared, O_{new} would have degraded to L3 in terms of cost because it would have required frequent comparisons.

Figure 6.16 shows three columns for O_{new}, L3, and L6 respectively for each application. The y-axis shows the number of errors reported. As seen from the graph, O_{new} is able to report almost as many errors as L3. L6 performed worst because it missed all the transient errors.

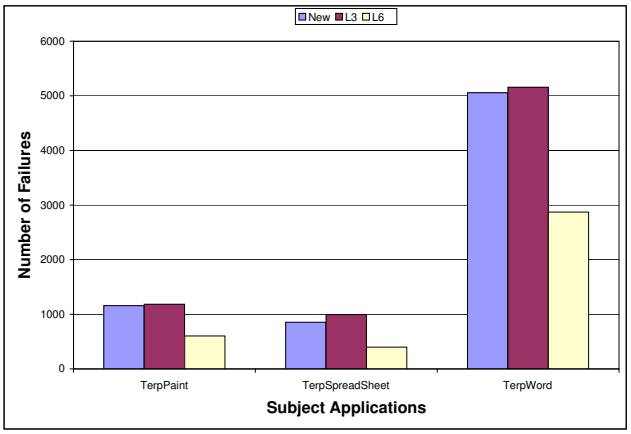


Figure 6.16: Error detection of O_{new}

The time to execute the oracles also varied significantly. Figure 6.17 shows three

columns for O_{new}, L3, and L6 respectively for each application. The y-axis shows the time required in seconds for all the test cases. As seen from the graph, O_{new} requires significantly less time than L3 and more time than L6.

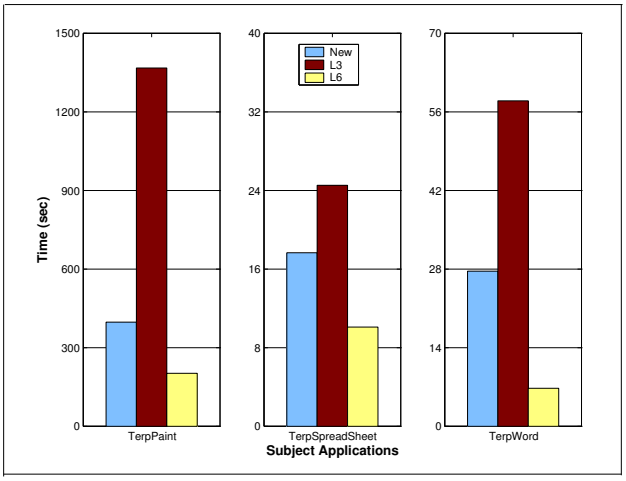


Figure 6.17: Time Required for O_{new}

The results of this study showed that O_{new} was almost as effective as L3 in terms of error detection. However, it was much cheaper to execute.

6.3 Conclusions

This chapter presented two experiments. The first experiment studied the effect of test case length, test suite size and event composition on fault detection and cost. The goal of this experiment was to develop an initial set of lessons learned that GUI testers may use to develop better test cases.

The second experiment led to the observation that GUI errors "appear" and later "disappear" at several points (*e.g.*, after an event is executed) during test case execution.

Two types of GUI errors were defined – *transient*, those that disappear and *persistent*, those that don't disappear. The experiment showed that in practice, a large number of errors in GUIs are transient and that there are specific classes of events that lead to transient errors. Testers need to compare the expected and actual output at these strategic points during test case execution.

Chapter 7

Summary and Future Work

This dissertation developed a continuous GUI testing process that is applicable to today's evolving GUIs. The research contributions of this dissertation that helped to realize the process include an abstract model of the GUI and a set of model-based techniques for test-case generation, test oracle creation, and continuous GUI testing. The models and techniques were obtained by studying GUI faults, interactions between GUI events, and why certain event interactions lead to faults.

7.1 Summary of Contributions

The continuous process consists of three concentric testing loops, each with specific GUI testing goals, resource usage, and targeted feedback. The innermost loop, called *crash testing*, is executed very frequently and is very inexpensive. Software crashes are reported back to the developer who initiated the check-in. The second loop, called *smoke testing*, is executed nightly/daily and completes within 8-10 hours. The third, and outermost loop, called *comprehensive GUI testing*, is executed after a major version of the GUI is available.

Several techniques were developed as part of this research to enable the above process. Each technique is a new research contribution of this dissertation. A new GUI model that represents potentially problematic event interactions was developed. The model was

obtained by using automated techniques that employ reverse engineering, thereby eliminating manual work. It was then used to generate test cases, create descriptions of expected execution behavior, and evaluate the adequacy of the generated test cases. The fault detection effectiveness of all the techniques was empirically evaluated on several open-source GUI subjects developed in-house and downloaded from SourceForge.

The experiments and analyses conducted in this dissertation have also contributed to a better understanding of GUI faults, GUI design, and how GUIs should be tested and developed. Code coverage analysis showed that each user event executed a specific part of the GUI code (called the event handler). In most cases, no other event executed this code. Since the subject applications used in this research were implemented using an object-oriented programming language (Java), event handlers were usually implemented as Java methods. Handlers for functionally related events (*e.g.*, file open, file save) share some methods and are almost always implemented as part of a Java class. Event handlers typically have one of three structures. First, a few event handlers have no conditional statements; they contain only one basic block. Faults in this code are likely to be detected each time the corresponding event is executed, irrespective of the state in which it is executed. However, these types of incidents are very rare since very few event handlers have this structure.

The second and most common type of event handlers contains at least one simple conditional statement, which checks the value of a single variable. This statement is used to enable/disable the event. The variable is set/reset using other events (*e.g.*, `Copy`/`Cut` enable `Paste`). Hence, most GUI faults are detected if events are executed in short test cases with a large number of preceding events. This observation is also supported by the

results shown in Figure 6.10; length 3 through length 10 test cases detected additional faults since they executed events in new states. The above two types of structures, *i.e.*, (1) no conditional statements and (2) one simple conditional statement lead to "shallow" faults that can be detected by executing GUI events in different combinations.

The third type of structure of event handlers is the most complex, although rare. It typically consists of a complex conditional statement or several nested conditional statements. Detecting faults in this type of code requires long sequences of events that can set/reset variables. Event handlers rarely have this structure; hence GUIs have very few faults that require long test cases.

The results of the test oracles experiments presented in Chapter 5 and Chapter 6 also help to understand the characteristics of today's GUIs for "testability." First, GUIs contain several types of widgets. Some of these widgets have a state (*e.g.*, check-boxes, radio-buttons) whereas others are stateless (*e.g.*, buttons, pull-down menus). Events (such as clicking on a check-box) performed on state-based widgets are used to change (usually toggle) their state. A test oracle that checks the correctness of the state of the *current* widget (*i.e.*, on which an event was just executed) is able to detect specific types of faults – ones that may adversely affect the current widget's state only; other faults are missed. TerpPresent has many such faults. L1 is an example of this type of oracle. Second, many events affect the state of multiple widgets of the active window, not just the current widget. L2 is able to detect all faults that are manifested anywhere on the active window. Finally, several events affect the state of the entire GUI. For example, `OK` in "preferences setting" has a global impact on the overall GUI. Oracle L3 is able to detect faults in such events.

The frequency of oracle invocation has a significant impact on fault detection effectiveness since the constantly changing structure (*e.g.*, currently open windows, active window) of the executing GUI provides a small "window of opportunity" for fault detection. A test oracle (such as L1 or L4) that examines only the current widget, if not invoked immediately after a faulty widget state is encountered, will fail to detect the problem. Hence L4, which waits until the last event, to examine the then-current widget detects fewer faults than L1. L4 is successful only if the widget associated with the test case's last event is problematic, as was the case with TerpPresent. Similarly, L5 detects fewer faults than L2 because a faulty active window is either closed or is no longer the active window by the time the last event in the test case executes; L5 misses these faults. On the other hand, L2 is able to detect such faults immediately as they are manifested on the active window. The small difference between L3 and L6 is due to the windows/widgets that are available at any time for examination. Errors that persist anywhere (*i.e.*, in any window or widget) across the entire test case execution are easily detected by L6 since it examines the entire state of the GUI after the last event. L6 misses only those errors that occurred in windows that were later closed or "disappeared" due to other reasons. The small number of such disappearing errors in TerpWord, TerpPaint, and TerpSpreadSheet show the reduced impact of comparing the entire state after each event.

The cost of test oracles is directly related to GUI layout issues that stem from usability concerns. Factors that impact the cost of the test oracles include the number of windows in the GUI that are open at any time (since L3 and L6 compare a larger number of widgets) and the number of widgets per window (since L2 and L5 compare all the widgets in the active window). There are several lessons-learned for GUI developers and test

designers. First, testers who use capture/replay tools typically create assertions for very few widgets after each event (*e.g.*, the one on which the current event is being executed). Seeing that L1 and L4 were the least effective at detecting faults, testers need to capture more information with their test cases, perhaps by using a reverse engineering tool; use of such automated tools will also reduce the overall effort required to create these oracles. Second, since it is difficult and expensive to create many long GUI test cases, testers who conserve their resources and create few short test cases should use test oracles such as L3 and L6 that check a more complete state of the GUI to improve fault-detection effectiveness. Third, testers should realize that the dynamic nature of GUIs provides a small window of opportunity to detect faults. They should place their assertions at strategic places in the test case (*e.g.*, before a window/menu is closed) to maximize fault-detection effectiveness. Finally, GUI designers must realize that their decisions will not only have an impact on usability but also on its "testability."

7.2 Future Work

Several open issues and intriguing research questions were raised while conducting this research and performing the experiments. These issues and questions point to the following future research directions.

1. *Using other reduced models similar to EIG for GUIs:* In this research, model-based techniques helped to generate test cases and create test oracles automatically and systematically. There may be many other ways to create new reduced models for GUIs. Most of today's GUIs have a central component, and the central component

interacts with other events. For example, the central component of Microsoft Paint is the canvas, which interacts with other events in the GUI. It may be possible to further reduce the EIG model and create a "star" model of the GUI, where the center of the star is the central component.

2. *Applying fault-injection techniques:* The experiments conducted in this research revealed that exception handlers are rarely executed by GUI test cases. Fault-injection techniques are commonly used to test exception handling code in fault-tolerant systems. New techniques based on fault-injection may be developed to enhance the EIG model and its associated testing algorithms.

3. *Developing techniques to identify false positives:* A significant issue observed during smoke testing is that testers have to manually identify false positives. New techniques/models may be developed to identify these false positives automatically. The experimentation infrastructure developed in this dissertation may be leveraged to conduct new experiments on multiple versions of software and their fault-seeded versions to evaluate the impact of false positives.

4. *Applying static analysis:* The experiments in this research showed that certain types of events interact with each other. It is important to test such events together. However, identifying sets of interacting events is a complex problem. Static analysis techniques may be used to identify sets of event handlers that interact with each other. New testing techniques may be developed to partition the EIG model into clusters of related events and test the related events together.

5. *Testing web applications:* Web applications are also based on the event-driven model of GUI applications; users interact with these applications that change their state and produce outputs, while the application continues waiting for the next user event. However, a web application may be executed in a large number of different client configurations that may change its execution behavior. This additional demand for portability imposes new requirements on test cases, test oracles, and coverage criteria for web applications testing. The GUI models and algorithms developed in this dissertation may be enhanced to handle multiple configurations.

6. *Testing object-oriented systems:* Modern software development is truly an engineering effort where a software developer composes software by reusing classes, objects, and components. However, these development paradigms create new challenges for testing. Source code from certain classes may not be available to the test designer. In such cases, code-based testing may not be applicable. An interface-based technique similar to the one used for GUI testing may be beneficial.

7. *Testing other event-driven software:* GUI testing techniques may be extended to other event-driven software as well. Common examples include component-based systems, embedded software, etc. Software components form the building-blocks of most of today's large software systems. Messages (events) are sent from one component to another. Components react by changing their internal state, responding with messages, and/or waiting for the next message. Similarly, embedded software controls modern buildings, cars, elevators, etc. Sensors send signals to the software, which changes its state, sends output signals to control devices, and con-

tinues to wait for signals. The test cases for these event-driven software are all sequences of events. Some of the techniques developed in this dissertation may be enhanced to test these classes of software.

8. *Extending subject application pool:* The GUI subject applications used in this dissertation have a fixed number of windows and deterministic behavior. Moreover, they are all implemented in Java. In the future, characteristics of these applications may be used to identify other types of applications that are "different" in that they require the development of new techniques for testing.

9. *Extending the fault classes:* Twelve types of faults were modeled in this dissertation. It may be possible to study the nature of these faults and their impact on GUI failures and to use the results of these studies to develop additional classes of faults that are specific to GUI functions.

BIBLIOGRAPHY

[1] Aegis: Software Configuration Management System. http://aegis.sourceforge.net.

[2] JUnit, Testing Resources for Extreme Programming. http://junit.org/news/extension/gui/index.htm.

[3] Abbot Java GUI Test Framework, 2003. http://abbot.sourceforge.net.

[4] Mercury Interactive WinRunner, 2003. http://www.mercuryinteractive.com/products/winrunner.

[5] Rational Robot, 2003. http://www.rational.com.ar/tools/robot.html.

[6] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, New York, NY, USA, 2005. ACM Press.

[7] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[8] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[9] P. J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology*, 3(3):201–220, July 1994.

[10] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 86–95. IEEE Computer Society, 2004.

[11] H. F. Brophy. Improving programming performance. *Australian Computer Journal*, 2(2):66–70, 1970.

[12] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE trans. on Software Engineering*, SE-4, 3:178–187, 1978.

[13] 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.

[14] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[15] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] S. Elbaum, S. Karre, and G. Rothermel. Improving web application testing with user session data. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[17] S. Esmelioglu and L. Apfelbaum. Automated test generation, execution, and reporting. In *Proceedings of Pacific Northwest Software Quality Conference*. IEEE Press, Oct 1997.

[18] M. Fewster and D. Graham. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[19] 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.

[20] B. S. Green. Software test automation. *SIGSOFT Softw. Eng. Notes*, 25(3):66–66, 2000.

[21] 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*, Orlando, Florida, May 2002.

[22] M. J. Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM Press.

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

[24] B. Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[25] 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.

[26] 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.

[27] D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, pages 244–251, New York, 13–18 Apr. 1996. ACM Press.

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

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

[30] A. Memon, A. Nagarajan, and Q. Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27–64, 2005.

[31] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, and B. Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Proceedings of the 26th IEEE/ACM International Conference on Software Engineering*, Edinburgh, Scotland, May 2004. IEEE/ACM.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, pages 118–127, Sept. 2003.

[40] 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.

[41] A. M. Memon and Q. Xie. Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In *Proceedings of The International Conference on Software Maintenance 2004 (ICSM'04)*, pages 8–17, Chicago, Illinois, USA, Sept. 2004.

[42] A. M. Memon and Q. Xie. Using transient/persistent errors to develop automated test oracles for event-driven software. In *Proceedings of The International Conference on Automated Software Engineering 2004 (ASE'04)*, pages 186–195, Linz, Austria, Sept. 2004.

[43] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, Oct. 2005.

[44] A. Michail and T. Xie. Helping users avoid bugs in GUI applications. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 05)*, pages 107–116, May 2005.

[45] B. A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995.

[46] B. A. Myers. Why are Human-Computer Interfaces Difficult to Design and Implement? Technical Report CMU-CS-93-183, July 93.

[47] 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.

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

[49] W. E. Perry. *How to test software packages: a step-by-step guide to assuring they do what you want*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[50] D. J. Richardson, S. Leif-Aha, and T. O. OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[51] R. D. Riecken, J. Koenemann-Belliveau, and S. P. Robertson. What do expert programmers communicate by means of descriptive commenting? In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 177–195, 1991.

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

[53] S. R. Schach. Testing: Principles and practice. In *Allen B. Tucker, Jr. (Editor-in-Chief), The Computer Science and Engineering Handbook*. CRC Press, in cooperation with ACM, 1997.

[54] C. B. Seaman and V. R. Basili. Communication and organization: An empirical study of discussion in inspection meetings. *IEEE Transactions on Software Engineering*, 24(7):559–572, July 1998.

[55] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington - Brussels - Tokyo, June 1997. IEEE Press.

[56] S. Siegel. *Object oriented software testing: a hierarchical approach*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[57] Software Research, Inc., capture-Replay Tool, 2003. Available at `http://soft.com`.

[58] J. Su and P. R. Ritter. Experience in testing the Motif interface. *IEEE Software*, 8(2):26–33, Mar. 1991.

[59] P. A. Vogel. An integrated general purpose automated test environment. In T. Ostrand and E. Weyuker, editors, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 61–69, New York, NY, USA, June 1993. ACM Press.

[60] E. J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982.

[61] L. White. Regression testing of GUI event interactions. In *Proceedings of the International Conference on Software Maintenance*, pages 350–358, Washington, Nov.4–8 1996.

[62] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 110–121, Oct. 2000.

[63] 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.

[64] L. White, H. Almezen, and S. Sastry. Firewall regression testing of GUI sequences and their interactions. In *Proceedings of the International Conference on Software Maintenance*, pages 398–409, the Netherlands, Sept.22–26 2003.

[65] Q. Xie. Developing cost-effective model-based techniques for GUI testing. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 997–1000, New York, NY, USA, 2006. ACM Press.

[66] Q. Xie and A. M. Memon. Rapid crash testing for continuously evolving GUI-based software applications. In *Proceedings of The International Conference on Software Maintenance 2005 (ICSM'05)*, pages 473–482, Budapest, Hungary, Sept. 2005.

[67] Q. Xie and A. M. Memon. Model-based testing of community-driven open-source GUI applications. In *Proceedings of The International Conference on Software Maintenance 2006 (ICSM'06)*, Philadelphia, PA, USA, Sept. 2006.

[68] Q. Xie and A. M. Memon. Studying the characteristics of a good GUI test suite. In *Proceedings of The International Symposium on Software Reliability Engineering 2006 (ISSRE'06)*, Raleigh, NC, USA, Nov. 2006.

[69] Q. Xie and A. M. Memon. *Agile Quality Assurance Techniques for GUI-Based Applications*. Idea Group Inc., 2007. to appear.

[70] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, to appear.

[71] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.