Studying the Characteristics of a "Good" GUI Test Suite

Qing Xie and Atif M Memon
Department of Computer Science
University of Maryland, College Park, MD 20742
{qing, atif}@cs.umd.edu

Abstract

The widespread deployment of graphical-user interfaces (GUIs) has increased the overall complexity of testing. A GUI test designer needs to perform the daunting task of adequately testing the GUI, which typically has very large input interaction spaces, while considering tradeoffs between GUI test suite characteristics such as the number of test cases (each modeled as a sequence of events), their lengths, and the event composition of each test case. There are no published empirical studies on GUI testing that a GUI test designer may reference to make decisions about these characteristics. Consequently, in practice, very few GUI testers know how to design their test suites. This paper takes the first step towards assisting in GUI test design by presenting an empirical study that evaluates the effect of these characteristics on testing cost and fault detection effectiveness. The results show that two factors significantly effect the fault-detection effectiveness of a test suite: (1) the diversity of states in which an event executes and (2) the event coverage of the suite. Test designers need to improve the diversity of states in which each event executes by developing a large number of short test cases to detect the majority of "shallow" faults, which are artifacts of modern GUI design. Additional resources should be used to develop a small number of long test cases to detect a small number of "deep" faults.

1 Introduction

Designing a test suite is widely recognized as a fundamental activity for effective software testing. It includes complex tasks such as the definition of test requirements (objectives), determination of the type of test cases needed, determination of the type of *test oracles* (mechanisms to determine whether the software is executing correctly during test execution [6]) needed for each test case, definition of test adequacy criteria, and development of schedules for test creation and execution. Ideally, a test designer formulates a test design that is both within budget and maximizes the chances of finding software defects. Formulating a successful test design requires considerable experience on the part of the tester. Different application parameters (*e.g.*, whether it contains a graphical user interface (GUI), application domain, implementation platform) and different test team composition parameters (*e.g.*, training level of personnel, familiarity with tools) must be taken into account when designing the test suite. Testers rely on past experience, either their own or of others packaged into mathematical models. Common examples of packaged models that may be used for certain aspects of test suite design include cost estimation tools (*e.g.*, CosteXpert [1], SLIM-Estimate [3], PRICE TruePlanning [2]) and defect estimation models (*e.g.*, COnstructive QUALity MOdel (COQUALMO) [5]).

The widespread deployment of GUI-based applications has made testing significantly more complex. The functional correctness of an application's GUI is necessary to ensure the correctness of the overall application. To test a GUI, testers create test cases consisting of sequences of GUI input events. The tester needs to answer the questions: (1) How many test cases should be generated? (2) What should be the length of each test case? (3) What events should be put into a test case? These questions are especially important for GUIs for a number of reasons. First of all, the space of possible user interactions with a GUI is enormous, in that each sequence of GUI events can result in a different state, and each GUI event may need to be evaluated in all of these states [10]. The large number of possible states results in a large number of input permutations [10] requiring extensive testing. The tester needs to cover this extensive event-interaction space by considering factors such as the number of test cases, their composition in terms of events, and the length of each test case.

GUI test design can be improved in several ways. First, by developing models and techniques that give the tester a global view of the overall test process and allow the definition of global test requirements, e.g., CIS model by White et. al. [16]. Second, by providing empirical evidence that shows the relationship between types of GUI test cases and GUI faults. We have already started to address the former by developing an *event-flow model* of the GUI and its events

[17, 12]. We briefly describe parts of the event-flow model in Section 3. In this paper, we focus on the latter. We posit that a fundamental reason for poor GUI test design is the lack of empirical studies that demonstrate the tradeoffs between various attributes of test cases (*e.g.*, length, event composition) and test suites (*e.g.*, size), and their impact on fault detection effectiveness. A GUI tester needs access to results of experiments that point to effective combinations of these attributes. We provide the first such empirical study in this paper.

More specifically, we design an empirical study in which we choose subject applications with GUI front-ends and generate test cases for them. We vary several key characteristics of GUI test suites that are of interest to testers, namely size of the suite, event composition, and the length of each test case. For each combination of these characteristics, we report the impact on fault detection effectiveness and cost. Our goal is to compile a set of "lessons learned" that can be used by testers to create effective GUI test cases. We note that this study is the first in a series of much needed studies to develop and evolve a set of lessons learned. While the work presented in this paper should be considered work-in-progress, we feel that it provides a strong starting point with several useful results. We have also made the artifacts of our study available as downloadable "benchmarks" on the Internet,[1] thereby enabling other researchers to use them and extend our work. We mention some exciting new directions in Section 7.

We report several lessons learned from the empirical study. We show that two factors significantly effect the fault-detection effectiveness of a test suite: (1) the *diversity of states* in which an event executes and (2) the *event coverage* of the suite. We also show that testers need to develop a large number of short test cases to detect the majority of "shallow" GUI faults typically found in modern GUIs; additional resources, if available, should be used to generate a small number of long test cases to detect a relatively smaller number of "deep" faults.

The contributions of this work include: (1) A comprehensive empirical study comparing several GUI testing factors, (2) the first attempt to assist in test design for GUIs, (3) the relationship between test suite size and fault-detection effectiveness, (4) the impact of test case length on fault-detection effectiveness, (5) the role of state diversity in GUI testing, and (6) a collection of shared artifacts that other researchers may leverage to replicate the study and conduct new ones.

Structure of the paper: The next section discusses related work; Section 3 introduces basic terminology needed to understand the experiments. Section 4 briefly describes the experimentation infrastructure. The experiments and their results are presented in Section 5. Finally, Section 7 concludes with a discussion of on-going work and future research opportunities.

[1] http://www.cs.umd.edu/users/atif/benchmarks.htm

2 Related Work

In practice careful test design is neglected for GUI testing. Due to the complexity of GUI testing, most testers don't develop test designs; rather, the *de facto* strategy is to "stop testing when they run out of time." In fact, current techniques for GUI test-case generation and test oracle creation promote incomplete and *ad hoc* test designing in that they force testers to test GUIs on a per test case basis. GUI testing tools rely on the tester's memory to remember what events have been executed by the previously generated test cases; a tester who uses these tools loses global view of the testing process. The most popular tools used to test GUIs are capture/replay tools such as WinRunner[2] that require the GUI tester to make decisions on the fly. The tester uses these tools in two phases: a capture and then a replay phase. During the capture phase, a tester manually interacts with the GUI being tested and performs events. The tool records the interactions. The tester also manually "asserts" that certain attributes of specific widgets be stored. The recorded test cases can be replayed automatically on (a modified version of) the software using the replay part of the tool. The assertions can be used to check the GUI's execution for correctness. As can be imagined, these tools require a significant amount of manual effort. Testers who employ these tools typically come up with a small number of test cases [10]. Moreover, since they make decisions (*e.g.*, events to execute, length of test case) on a per test case basis when they interact with the application, they lose global perspective of the overall testing process. They also do not get immediate feedback on the full impact of their decisions on fault detection effectiveness and test coverage.

Commercially available test composition/management tools [2, 3] (none available for GUI testing) typically have more or less identical features. They allow users to compose and manage tests. Tests can be mapped to functional requirements. They may be assigned a type (*e.g.*, regression, functional). These tools also provide test management support. For example, if a tester does not want to execute all test cases, the tools allow testers to pick test cases for execution. They track the pass and fail rate of each test. The testers can view bug, progress, and failure-rate reports.

Defect estimation/prediction models [7] may be used for some aspects of testing. Ostrand and Weyuker [13] use software content and development process measures to predict faulty files in multiple releases of two software systems at AT&T. The COQUALMO project uses COCOMO II data to estimate the total number of defects in a software system [5]. However, none of the existing defect estimation models provide explicit support for GUI defects.

[2] `http://mercuryinteractive.com`

3 Basic Terms

Since we will study the interactions between the most fundamental factors considered by GUI testers, *i.e.*, test suite size, event composition of a test case, and test case length, in this section we briefly describe these factors. Note that due to lack of space, we provide details needed to understand the empirical studies presented in this paper; the interested reader is referred to previously published results for additional details [12, 9].

Since GUIs may be used as front-ends to all types of software, in principle, the space of all possible types of GUIs is enormous. Creating a representation of all possible GUIs and studying them is a Herculean (if not impossible) task. Any empirical study must focus on a reasonable subclass; it is important, however, to choose a subclass that is broad enough to be of interest. Hence, we focus on the broad class of GUIs defined next.

Definition: A *Graphical User Interface (GUI)* is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output. A GUI contains graphical *widgets*; each widget has a fixed set of *properties*. At any time during the execution of the GUI, these properties have discrete *values*, the set of which constitutes the state of the GUI. □

Note that this definition would need to be extended for other GUI classes such as web-user interfaces that have synchronization/timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible to model the state of the software in its entirety and hence the effect of an event cannot be predicted. In the remainder of this paper, we will present models, techniques, and studies that are relevant to the above class of GUIs.

We now define a GUI test case and associated terms.

Definition: A **GUI test case** T is a pair $< S_0, e_1; e_2; \ldots; e_n >$, consisting of a state S_0, called the *initial state for T*, and an allowable event sequence $e_1; e_2; \ldots; e_n$. The *length* of the test case is n, *i.e.*, the number of events in the test case. □

The state S_i of a GUI, at any point during its execution, may be modeled as a set of *widgets* (*e.g.*, buttons, panels, text fields) that constitute the GUI, a set of *properties* (*e.g.*, background color, size, font) of these widgets, and a set of *values* (*e.g.*, red, bold, 16pt) associated with the properties. The initial state S_0 of test case T is the GUI state in which the first event e_1 of T is executed.

The state of a GUI is not static; sequences of *events* $e_1; e_2; \ldots; e_n$ performed on the GUI change its state to S_1, S_2, ..., S_n successively. During testing, a tester generates sequences of events as test cases, executes them on the GUI and checks the GUI for correctness. To generate a test case, a tester may use a capture/replay tool (described in Section 2), starts in a state S_0 of the GUI, and executes an allowable sequence $e_1; e_2; \ldots; e_n$ of events. The tester visually indicates parts of the GUI's state that should be stored during capture so that they can be used as a reference during replay. For example, in MS Word, the tester may want to check that the "File Open" window is the currently active window after the menu-item "Open" (in the "File" pull-down menu) has been selected; the tester uses the capture/replay tool to "assert" that the value of the "is-active" property of the "File Open" window is `TRUE`. During (replay) test case execution, the replayer would raise an error flag if the assertion is violated.

Testers may use a number of heuristics (also applicable to state-based testing) to improve their chances of finding software faults. For example, they may generate a test suite that contains only length 1 test cases that test each event once. Such a test suite has *full event coverage*, *i.e.*, it contains test cases that execute all the events in the GUI at least once. However, intuitively, one expects that test suites with longer test cases (but will take more time to generate and execute) will put the GUI in different states causing different execution behavior, perhaps leading to additional fault detection. GUI testers constantly have to make such tradeoff decisions between resource usage and test effectiveness.

4 GUITAR

This research leverages several years of work on GUI test automation, thereby enabling the creation of large numbers of test suites that may be empirically studied. A GUI test automation framework called GUITAR[3] is used to conduct the experiments. The key features of GUITAR (relevant to this work) include a model of the GUI's event-interaction space, automated tools for test-case generation, test oracle creation, test execution, and code coverage evaluation. GUITAR also generates all the scaffolding code (*e.g.*, test scripts) required to setup and "tear down" test cases, collect GUI state information, invoke the test oracle, and produce test reports. This code can be distributed on several different computers to execute test cases in parallel automatically. The key parts of GUITAR are discussed next.

The space of all possible interactions with the GUI are modeled in GUITAR as an *event-flow graph* (EFG) [11]. An EFG contains nodes (that represent events) and edges. An edge from node n_x to n_y means that the event represented by n_y may be performed *immediately after* the event represented by node n_x. This relationship represented by the edges is called `follows`. A function `follows(x)` takes an event `x` as input and returns a set of events that can be executed immediately after `x`. Note that as the nodes in an EFG are events (not states), and edges are the `follows` relationship (not state transitions), the EFG is not a

[3] http://guitar.cs.umd.edu

state-machine model.

EFGs exhibit certain properties. First, they can be obtained automatically from an executing GUI using reverse engineering techniques followed by manual verification by the tester (details are presented in [8]). Second, EFGs 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, enumerating the nodes during the traversal, on 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.

Several graph-traversal techniques may be used to generate different types of test cases. For example, enumerating all the nodes in the EFG (and hence in the GUI) results in a collection of length 1 test cases. Executing all these test cases will result in full event coverage. Enumerating all edges $< e_x, e_y >$ of an EFG will yield all length 2 test cases. Note that some of these test cases may not be executable since the first event in the test case may be embedded in a menu or window not yet open. For example, the event `Check for Updates` in MS Word 2003 is not available when the software is first launched; it requires the execution of the event `Help`, which opens a pull-down menu that makes `Check for Updates` available. Such *prefix* events are generated by GUITAR on-the-fly during test case execution, *i.e.*, if the first event in a test case is not available, GUITAR's algorithms generate the necessary prefix.

5 Experiments

This section presents the design and results of experiments to study the tradeoffs between GUI test suite characteristics, namely test suite size, event composition, and length of test cases. In particular, the studies are designed to examine the 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 experiments are designed to statistically 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$\}$. In these experiments, the alternative hypothesis will be the negation of the corresponding null hypothesis. In most of the experiments presented in this paper, $\alpha = 0.05$, *i.e.*, the experiments' findings have a five percent (0.05) chance of not being true.

To conduct these experiments, a number of subject applications (TerpWord, TerpSpreadSheet, TerpPaint, and TerpCalc from the TerpOffice suite [12]) will be selected. A large number of test suites with various, carefully controlled characteristics will be created and executed on the subject applications. Keeping in mind the above hypotheses, the primary measured variable will be the fault-detection effectiveness of a test suite; the secondary measured variable will be the cost of generating and executing the test suite.

These experiments will also demonstrate that the test suite's event composition (*i.e.*, the set of events executed by the suites constituent test cases) has a direct impact on the faults that it detects.

5.1 Test Pool

The experiments presented in this paper require the development and execution of a large number of test cases. For example, Experiment 1 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 up to 30 seconds to execute (on average, each requires 10 seconds). Hence, for the results to be statistically significant, the experiments 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 [4]. 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 these experiments, 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 Experiment 2, 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.

A related approach was employed by Rothermel *et al.* [14] to create sequences of commands to test command-based software. In that approach, each command was executed in isolation and test cases were "assembled" by concatenating commands together in different permutations. Since GUI events (commands) enable/disable each other, most permutations result in infeasible sequences. Hence, the approach used here will employ event-flow graphs to

construct test cases that are feasible.

The following process was employed to create the test pool:

1. Create twenty[4] 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. Note that for execution, some of these test cases may require a prefix, discussed in Section 4, which is automatically generated by GUITAR's test executor on-the-fly.
3. For each event x in $bucket_1$, create five[5] 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 these experiments. 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.

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 1 in the form of box-plots. Each box's height spans the central 50% of the data, and its upper and lower ends mark the upper and lower quartiles. The bold dot within the box denotes the median. The dashed vertical lines attached to the box extend to 10% and 90% of the data. All other detached points are "outliers." 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.

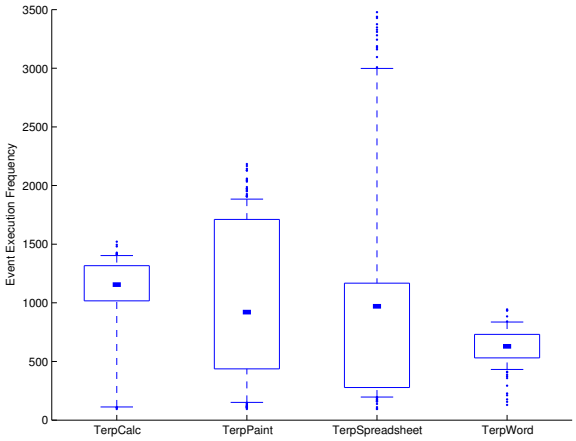

Figure 1. Event Distribution for Each Application.

5.2 Fault Seeding

As discussed in Section 3, a GUI fault is a mismatch, detected by a test oracle, between the "ideal" (or expected) and actual GUI states. Hence, to detect faults, a description of ideal GUI execution state is needed. This description is used by test oracles to detect faults in the subject applications. There are several ways to create this description. First, is to manually create a formal GUI specification and use it to automatically create test oracles [9]. Second is to use a capture/replay tool (discussed in Section 2) to manually develop assertions corresponding to test oracles and use the assertions as oracles to test other versions of the subject applications. Third is to develop the test oracle from a "golden" version of the subject application and use the oracle to test other versions of the application. The first two approaches are extremely labor intensive since they require the development of a formal specification and the use of manual capture/replay tools; the third approach can be performed automatically and has been used in this research. For this research, the faults were similar to those used in [12].

5.3 Test Execution

An automated tool (a part of GUITAR) was used to execute all the test cases in the test pool on the subject applications. The tool automatically executed each event in each test case and captured the GUI state (widgets, properties, and values) automatically by using the Java Swing API. The state information was then converted to a sequence of `assertEquals(X, Y)` statements, where `X` is the extracted value of a widget's property. `Y` is a placeholder that is intantiated with the corresponding value extracted

[4]Our 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.

[5]The choice of five copies is not arbitrary. These experiments were conducted with 2, 3 and 4 copies. There was no significant difference in results between 4 and 5 copies. Hence for these applications, we report the results of experiments that used 5 copies. In the future we will explore the impact of $|\texttt{follows(x)}| >> 5$ for other applications.

from the fault-seeded application. The method `assertEquals()` returns TRUE if its two parameters are equal, otherwise FALSE. Due to the limitations of the Java Swing API, the tool extracted 12 properties for each widget. The test cases were also executed on each fault-seeded version. The results of the `assertEquals()` were recorded. If, after an event e in a test case t executed on fault seeded version F_i, even one `assertEquals()` method returned FALSE, then t is said to have "*detected the fault* F_i." Event e is said to have been "*successful at detecting fault* F_i."

The test cases were executed on four computers, running Windows 2000. Each computer had 512MB of RAM and a 1.8GHz Pentium processor. Had all test cases been executed on each fault-seeded subject application, the total number of test runs would have exceeded 10 Million (53105×200). With each test case requiring 10 seconds to execute, the runs would have taken almost a year on the 4 machines. To conserve resources, first each test case was executed on the original software subjects and its statement coverage was recorded. Only if the test case executed the statement in which a fault was seeded, it was run on the corresponding fault-seeded version. Note that if the test case did not execute this line, the fault could not possibly manifest as an error. This time-saving strategy helped to reduce the execution time to two months, without loss of accuracy.

5.4 Computing Measured Variables

Two variables were measured in these experiments for each test suite, *i.e.*, cost in terms of execution time and fault-detection effectivess. 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.

5.5 Threats to Validity

As is the case with all controlled experiments, these experiments are subject to threats to validity. These threats need to be considered in order to assess their impact on the results. First is the selection of subject applications and their characteristics. The results may vary for applications that have a complex back-end, are not developed using the object-oriented paradigm, or have non-deterministic behavior. Second, the test pool approach was used due to practical limitations. It is expected that the repetition of the same test case across mutiple test suites will have an impact on some of the results. For example, if large test suites (containing more than 10K test cases) are created for TerpWord, which has a test pool of size aproximately 12K test cases, then each generated test suite will be more or less identical. The effect of this threat is minimized by considering reasonably small test suites (the largest TerpWord suite has 1700 test cases). Third, the algorithm used to create the test pool ensures that each event (the first event in the test case) is executed in a known initial state; the choice of this state may have an effect on the results. Fourth, the Java API allow the extraction of only 12 properties of each widget; faults are reported for mismatches between these 12 properties. Fifth, we capped the length at 20 events. The results may vary for longer test cases. We also used one technique to generate test cases – using event-flow graphs. Other techniques, *e.g.*, using capture/replay tools and programming the test cases manually may produce different types of test cases, which may show different execution behavior.

Several threats are related to fault seeding. Threats from issues such as human decision-making are minimized by using an objective technique for uniformly distributing faults based on functional units. The four graduate students were explicitly asked to use the fault classification.

5.6 Experiment 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 experiment, keeping other factors constant. In this 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, TerpPaint, 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 be exactly the same. Also, they all have similar-length test cases. Determine the fault-detection effectiveness of all 10 suites.

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 2 summarizes the results for TerpCalc (it is impossible to visually show the data for each set of 10 related suites separately; the results for other applications are more or less similar; they are not presented here due to lack of space). The figure shows 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 convergance towards a plateau above a size roughly corresponding to 1000 may be an artifact of the number of faults seeded and/or the size of the GUI; a detailed analysis will be conducted in future work.

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,

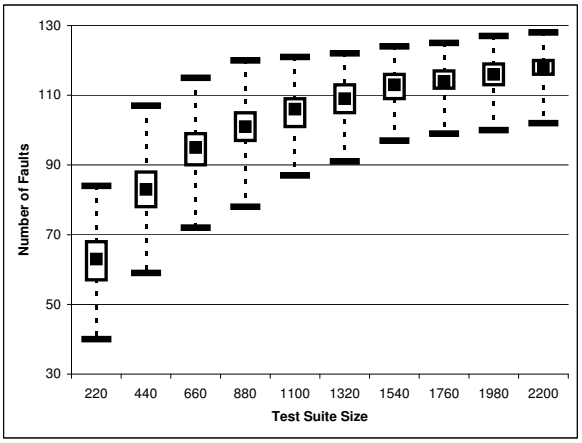

Figure 2. Fault Detection Effectiveness vs. Test Suite Size for TerpCalc

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.

This experiment 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.

5.7 Experiment 2: Effect of Test Case Length

This 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 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 Experiment 1. 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.

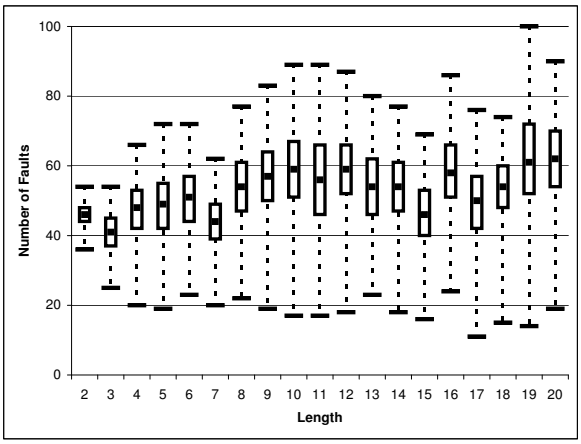

Figure 3. Fault Detection Effectiveness vs. Test Case Length for TerpCalc

The results for TerpCalc are summarized in Figure 3 (results were similar for other applications). 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.

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 4. 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.

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 Experiment 1.

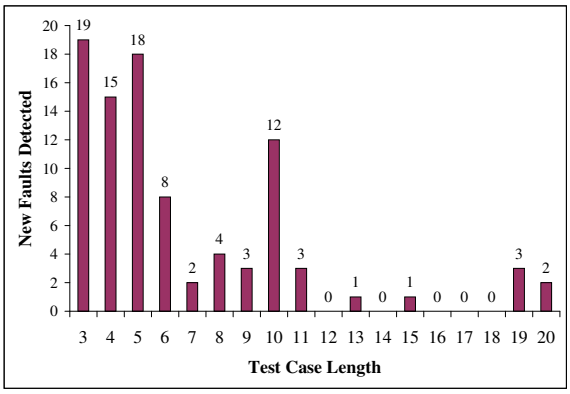

Figure 4. New Faults Detected with Length Increase

Long test cases, however, are more expensive to generate and execute. The generation time is proportional to the length of the test case. However, executing long test cases is expensive. Due to limitations of the event-flow graph models, GUITAR's test-case generator sometimes generates test cases that are in fact infeasible. In practice, long test cases that execute infeasible sequences crash the test executor or the application under test, causing substantial delays.[6]

5.8 Experiment 3: Effect of Event Composition

In the above experiments, the event composition of the test suites was kept constant, *i.e.*, all the events were used. This 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.
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.

[6]Note that for these experiments, infeasible sequences were substituted manually with feasible ones.

3. Determine the fault-detection effectiveness of the generated test suites. Repeat the above process using the three-value comparison technique outlined in Experiment 1.

As observed earlier, 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.

6 Discussion

The above three experiments demonstrated that several factors have an impact on the fault-detection effectiveness of a test suite in different and interesting ways. Post-experiment analysis, based on the code coverage (branch and statement) of the test cases, revealed a better understanding of how modern GUIs are designed and how they should be tested.

The coverage results 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. This observation explains the results of Experiment 3. 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 (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, very few event handlers have no conditional statements; in fact, 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.

As defined in Section 5.3, an event e is successful at detecting a fault F if `assertEquals()` returned FALSE immediately after e executed. The "success ratio of e" is defined as ((the number of times e successfully detected F)/(number of occurrences of e in the test pool)). It is undefined if e did not successfully detect F even once.

The success ratio for all the events of the subject applications is shown in Figure 5. The box-plots show that the success ratio is very small. The median (indicated by a small square inside each box) is close to zero. This result shows that even though a test suite may execute an event multiple times, the event is successful at detecting a fault very few times (in specific states).

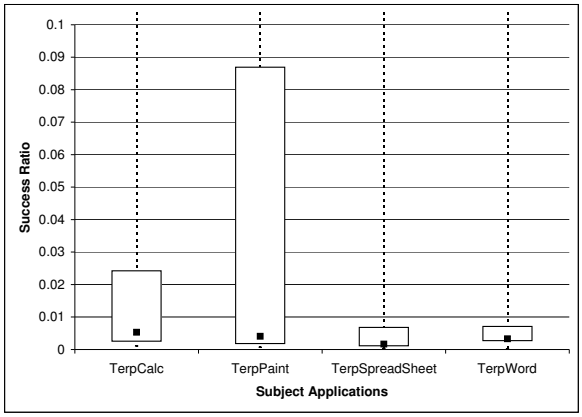

Figure 5. Frequency of Event Execution

The low success ratio is explained by the two other common structures of event handlers. The most common types of event handlers contain 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 Figure 4; 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 to obtain maximum branch coverage. Faults in this type of code are called "deep" faults. Event handlers rarely have this structure; hence GUIs have very few deep faults. Two such faults were detected by length 20 test cases in TerpCalc.

The analysis showed that there was a significant relationship between GUI errors and the way modern GUI code is developed using object-oriented languages. The three ex-

periments and code coverage analysis 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 many "shallow" bugs by generating a large number of short test cases in multiple combination of events. Additional resources may be used to find the relatively fewer "deep" bugs by generating long test cases.

7 Conclusions

This paper presented the first empirical study of the effect of test case length, test suite size and event composition on fault detection and cost. The goal of the study was to develop an initial set of lessons learned that GUI testers may use to develop better test cases. The study showed that a tester should allocate maximum resources to finding the many "shallow" bugs by generating a large number of short test cases. Any additional resources may be used to find the relatively fewer "deep" bugs by generating long test cases.

There are several interesting future directions for this research. An immediate extension will apply multivariate analysis where all three parameters vary, which may help to understand the interaction effects of these three parameters on fault detection. The study will be extended to other GUI test suite characteristics such as test oracle design. Previous research has demonstrated that the type of test oracle used for GUI testing has a significant impact on the fault-detection of GUI test cases. The interplay between test oracles, test case length, suite size, and event composition will be studied. The subject application set will be evolved to include complex back-ends. This evolution will also help to enhance the functional unit classification. This research used a simplistic model of fault-detection effectiveness. In the future, this model will be enhanced to give more "weight" to deep faults. Finally, this research will be extended to other classes of event-driven software applications, in particular web applications [15].

Acknowledgments

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

References

[1] CostXpert. http://www.costxpert.com/.

[2] Price systems. http://www.pricesystems.com/.

[3] Slim-estimate. http://www.qsm.com/.

[4] 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.

[5] S. Chulani. Bayesian analysis of software cost and quality models. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 565–568, 2001.

[6] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 106–117, Oct.16–18 1996.

[7] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, Sept./Oct. 1999.

[8] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, pages 260–269, 2003.

[9] 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.

[10] 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, 2001.

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

[12] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005.

[13] T. Ostrand and E. Weyuker. The distribution of faults in a large industrial software system. In P. G. Frankl, editor, *Proceedings of the ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, pages 55–64. ACM Press, July 22–24 2002.

[14] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13(3):277–331, 2004.

[15] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. Composing a framework to automate testing of operational web-based software. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 104–113. IEEE Computer Society, 2004.

[16] 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, 2003.

[17] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. on Softw. Eng. Methodol.*, to appear.