

Fault-Detection Effectiveness and Fault Detectability in GUI Testing

by

Jaymie Strecker

Dissertation Proposal

Committee:

Atif M. Memon (Chair)

Brian R. Hunt

Dianne P. O'Leary

Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, MD 20742
April 2008

With the abundance of testing techniques being developed, a critical question—yet one that has not been adequately answered by the research so far—is how to evaluate the effectiveness of these techniques. The answer is far from straightforward because different techniques may excel in different situations. For example, test suites generated by the same technique may differ in characteristics that affect their ability to detect faults. Different kinds of faults, in turn, may be more easily detected by some techniques, so evaluating the effectiveness of testing techniques against different samples of faults may give different results. This issue is further complicated by the fact that it is not even clear what characteristics should distinguish different kinds of faults.

This research proposes to explore three inter-related questions: How do characteristics of test suites affect their ability to detect faults? How do characteristics of faults affect their detectability by test suites? And how should faults be characterized? Two insights that are expected to make progress toward answering these questions will be investigated. The first insight is that, in empirical studies of testing techniques, testing situations—in particular, test suites and faults—should be characterized, and results should be presented in terms of those characterizations, to enable researchers or practitioners in different situations to better understand how the study results would translate to their situation. The second insight is that the problem of empirically studying fault detection in testing can be broken down into the sub-problems of studying coverage of faulty program elements, state failures introduced by faults, and failures introduced by state failures. In the proposed research, these two insights will be analyzed empirically. In addition, this work proposes to develop a practical application of the first insight: a framework for adaptive, search-based regression testing, which is expected to enhance existing regression-testing techniques by enabling testers to learn from previous testing iterations how to choose and improve the technique used in the next iteration.

To take advantage of existing resources and to limit the proposed work to a feasible scope, the work will focus on testing in the domain of graphical user interfaces (GUIs), which represent an increasing proportion of modern software yet a small proportion of subjects in software-testing

studies. Although the empirical studies in this work propose to focus on GUI testing, the insights that motivate the studies will potentially apply to a wide range of testing domains.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Existing methods for evaluating fault detection in testing	5
1.3	Existing methods for characterizing faults	6
1.4	GUI testing	7
1.5	Proposed approach	10
1.6	Preliminary work	13
1.7	Intellectual merit and broader impacts	13
2	Background and Related Work	15
2.1	Factors shown to affect fault detection in testing	16
2.2	Fault characterizations	19
2.3	Faults' relation to failures	23
2.4	Adaptive testing techniques	23
2.5	Summary	24
3	Goals and Approaches	26
3.1	Goals	26
3.2	Tasks and approaches	28
4	Preliminary Work	33
4.1	Introduction	33
4.2	Variables of interest	35
4.3	Study design	36
4.3.1	The Sample of \langle Test Suite, Fault \rangle Pairs	37
4.3.2	Measurement of \langle Test Suite, Fault \rangle Pairs	41
4.3.3	Logistic Regression Analysis	42
4.4	Results	44
4.5	Discussion	47
4.6	Conclusions	50
4.7	Lessons learned from preliminary work	50
5	Remaining Work	52

List of Tables

4.1	Study variables	36
4.2	Size of applications, test suites, and test pools.	39
4.3	CWS: Data summary.	45
4.4	FM: Data summary.	45
4.5	CWS: All main effects.	46
4.6	CWS: Reduced main effects.	46
4.7	CWS: Reduced interactions.	46
4.8	FM: All main effects.	47
4.9	FM: Reduced main effects.	47
4.10	FM: Reduced interactions.	47
5.1	Timeline for completion	53

List of Figures

1.1	A GUI and part of its event-flow graph	9
2.1	In lines 3 and 5, the same mutation operator has been applied, changing != to ==. The program never fails at line 3 but always fails at line 5.	20
2.2	The fault in line 4 is semantically small but potentially costly.	20

Chapter 1

Introduction

1.1 Motivation

Software testing is a critical part of the quality assurance process of many organizations. Testing helps developers detect and locate problems with software. When no problems are found, it builds confidence that the software works correctly.

In software testing, *test cases*, which are elements of the input space, are executed on the software. Typically, multiple test cases are run; together these test cases make up a *test suite*. When a test case is run, its output and/or the resulting program state are checked against the *oracle*, which describes the expected output and/or state. If the actual output or state differs from the oracle, then a *failure* is said to occur. A failure is caused by a *fault*, which is an incorrect piece of source code. A test case is said to *detect* a fault if its execution leads to a failure that can be attributed to that fault.

Numerous techniques for performing testing have been put forth in the literature. These include algorithms and approaches to generating test cases, measuring the portion of the software covered by a test suite, selecting test cases from a previously-run suite to run on a new version of the software, and prioritizing the test cases in a suite. The latter two techniques are used in *regression*

testing, which is the process of re-testing a new version of software to make sure that the changes made since the previous version have not introduced new faults into the software.

Evaluating the effectiveness of testing techniques is a critical part of software-testing research. But how does one judge a test suite's effectiveness at detecting faults? By far the most common way is to run the test suite on a sample of faulty program versions and count the faults detected. For example, Graves et al. [11] compare the percentage of detected faults retained by different regression-test-selection techniques. To evaluate several regression-testing methods, Rothermel et al. [30] also use metrics based on numbers of faults detected. As these studies themselves admit, a potential weakness of this approach is that it ties the results to the sample of faults that happens to be used.

A related question is: How does one judge a fault's ease of detection? This question, too, is usually answered with sampling. A sample of test cases is run on the faulty program and the number that either detect the fault or cover the faulty program element are counted [1, 26, 34]. As with the approach above, the results are beholden to the sample chosen.

Thus, while fault-detection effectiveness of test suites and detectability of faults might in theory be absolute measures, any practical method for estimating them makes them relative to some sample of faults or test cases. This is not necessarily bad. If there were some widely agreed-upon population of faults or test cases, and if all samples drawn from this population gave roughly equivalent results for fault detection and detectability, then the reliance on sampling would not be a problem.

But different testers sample from different populations of test cases. They use different testing techniques, or they use the same technique but produce test suites with different characteristics. Consider the following simple program:

```
1  if (x == 10) {  
2      // Fault  
3  } else {
```

```
4 // OK
5 }
```

Suppose, for example, that a tester creates test suites using the criterion that they must be statement-coverage-adequate (i.e., they must execute each statement at least once). For this program, each of the two test cases in a minimal statement-coverage-adequate test suite (e.g., $\{(x = 10), (x = 11)\}$) could have a probability as high as 50% of detecting the fault in line 2. In contrast, methods that use a random choice of test cases from some domain (e.g., $x \in [-100, 100]$) to estimate fault detectability [1, 26, 34] would estimate the fault's probability of detection to be near zero. (Michael and Voas [21] have made a similar observation.) The point here is not that one population of test cases is better than another, but that a measure of fault detectability that is based on a fixed population may be irrelevant to testers who select test cases from a different population.

Likewise, a measurement of the fault-detection effectiveness of a test suite based on a sample of faults from one population may be quite different from a measurement based on a sample from another population. Presumably, different testers deal with different populations of faults. But, currently, there is no way to tell; when considering populations of faults, there is a difficulty that did not occur above with populations of test cases. Previous research gives us some idea of what characteristics make populations of test cases similar to or different from one another (e.g., technique, coverage, size of sample, granularity of test cases [4, 8, 11, 16, 19, 23, 30, 32, 36]). But there does not yet exist an analogous characterization of faults with proven applicability to testing [31]. While Basili and Selby [4] find promisingly that validation techniques performed differently on different classes of faults, the classification of faults they use is subjective and not automatable. Other proposed fault characterizations either depend on a sample of test cases (e.g., [26]) or have not been shown to make a difference in testing results [13]. So the problem is not just that different fault populations may lead to different measurements of fault-detection effectiveness for test suites, but also that there is no widely-accepted way to tell whether two fault populations *are*

meaningfully different.

Thus, three inter-related questions lie open: how test-case samples affect faults' detectability, how fault samples affect test suites' fault-detection effectiveness, and how fault samples should even be characterized. These are big questions that will require many independent studies to answer. Of course, this work only proposes to address a small part of these problems. Thus, it will focus on characterizing and studying test suites and faults in graphical-user-interface (GUI) testing.

This work proposes to explore two insights that are expected to improve upon the state of the art in characterizing and studying test suites and faults. With these insights, the results of empirical evaluations of testing techniques may be more readily generalized to new test subjects and interpreted in new situations. One insight is that, first, characteristics of test suites and faults that can affect testing techniques' effectiveness should be identified and, second, testing instances used to empirically evaluate testing techniques should be characterized with respect to those characteristics. This would enable researchers and practitioners to better predict how the evaluated techniques would perform in testing instances with different characteristics. The other insight is that the empirical investigation of fault detection should be decomposed into three sub-problems: the probability that a program location is executed, the probability that a fault at an executed location leads to an unexpected program state, and the probability that an unexpected program state leads to a failure detected by testing. This decomposition follows Richardson's and Thompson's [29] RELAY model of fault detection. Combined with the first insight described above, this insight is expected to lend unprecedented precision to the empirical understanding of fault detection in testing.

In addition, this work proposes to develop a novel approach to regression testing that is a practical application of the first insight described above. The approach is expected to be a framework for adaptive, search-based regression testing, which is meant to enable a tester to learn from the experience of previous iterations of regression testing how to make the next iteration more cost-

effective. Using the first insight above, the framework will characterize testing situations with respect to test-suite and fault characteristics that can affect testing techniques' effectiveness. As those characteristics change, the framework will potentially help testers adapt their testing strategies accordingly.

1.2 Existing methods for evaluating fault detection in testing

While some analytical methods for evaluating the fault-detecting abilities of testing techniques have been proposed (e.g., relations for coverage criteria like *subsumes*, *more powerful*, *better*, and *probbetter* [35]), most progress in evaluating fault detection has been made empirically. As they relate to this work, empirical studies of testing techniques' fault detection fall into two categories: those that use measures based on the number of faults detected in a sample of faults and those that go on to break down the numbers of faults detected with respect to characteristics of the faults in the sample. (Some studies measure numbers of program elements covered rather than numbers of faults detected; the arguments here regarding studies of fault detection apply as well to studies that use coverage as a proxy for fault detection—i.e., results of these studies can be made more informative by breaking them down with respect to characteristics of the program elements.)

As Section 1.1 stated, testing techniques are most often evaluated by running a sample of test suites generated with the technique on a sample of faulty program versions and counting the faults detected. A well-known weakness of such studies is that this method ties the results to the sample of faults used, which may not be representative of the testing situations of interest to other researchers and practitioners who would like to use the study results. Some studies (e.g., [16]) partially mitigate this weakness by providing information about the detectability of the faults used in the study with respect to the test cases used. Other studies (e.g., [2]) go further by analyzing results separately for faults at different levels of detectability. However, detectability in these studies is not an independent characteristics of faults but rather a characteristic of the interaction

between the faults and test cases used in the study. It does not describe an inherent property of a fault that influences the fault's probability of detection across all kinds of test cases.

A few empirical studies of testing techniques analyze the results with respect to such properties of faults. In a comparison of code reading, structural testing, and functional testing, Basili and Selby [4] break down the results along two fault taxonomies. Although the break-down gives some insight into the results—some techniques were better at detecting some kinds of faults but worse at others—the taxonomies used were not automatable and therefore not practical in many situations. In an evaluation of data-flow and mutation testing, Harrold et al. [13] classify faults with respect to the changes they induce on the program dependence graph. However, since mutation testing detected all 74 faults and data-flow testing detected 72, the taxonomy turned out not to help much in interpreting the study results.

1.3 Existing methods for characterizing faults

When empirical studies rely on samples of faults to evaluate testing techniques, it is not clear whether the study results generalize to other samples of faults. Although researchers often attempt to choose a “realistic” or “representative” sample of faults, too little is currently known about the role of faults in software testing to understand how realistic or representative a sample of faults really is [1, 13, 26]. To help a researcher or practitioner looking at study results understand how the results might translate to her context, the study results should be characterized in a way that satisfies several criteria: it should provide information useful to testing, it should be consistent and objective, it should be automatable, and it should not rely on artifacts such as formal specifications that may not exist [31].

In the literature, faults have been characterized with respect to several viewpoints. One that is frequently used is the syntactic view, which characterizes faults by the syntactic changes they induce on a program [26]. A view that is more closely related to fault detection in software testing

is the semantic view, which looks at the portion of the program's input-output mapping changed by the fault [26]. In a viewpoint that falls somewhere between syntactic and semantic, some researchers have characterized faults in terms of the changes they induce on program models, such as the program dependence graph [13] and UML class diagrams [6]. Finally, several partially or wholly subjective classifications have been used, such as the omissive/commissive and initialization/control/data/computation/interface/cosmetic taxonomies used by Basili and Selby [4], the IEEE Standard Classification for Software Anomalies, and classifications by fault severity. However, as Section 2.2 will explain, all of these characterizations either fail to meet at least one of the criteria listed above or have not been shown to be applicable in empirical studies.

1.4 GUI testing

The domain of interest in this work is model-based GUI testing. GUI testing is a critical step in the quality assurance of GUI-based applications, which make up a large portion of modern software. Tools in the GUI Testing Framework (GUITAR) have been created to rapidly and automatically model these applications and generate test cases for them [20]. One reason that this work will focus on GUI testing is the ready availability of these tools along with supporting infrastructure and expertise, since the tools were developed at this institution. These tools automate the creation and execution of test cases and the comparison of test outputs (i.e., the oracle procedure). GUITAR makes it possible to produce much larger data sets than would be feasible using other, more labor-intensive approaches such as capture-replay tools.

Because test cases can be generated with these tools much more quickly than they can be run, it is important to be able to predict how characteristics of test suites affect the kinds of faults detected during testing; this work will make progress toward that goal. Furthermore, this work will enlarge the body of empirical knowledge about the testing of GUI-based applications, which is small relative to the body of knowledge about traditional, batch-style applications.

Figure 1.1 shows a screenshot of a GUI. In model-based GUI testing, test cases consist of sequences of GUI *events*—interactions with GUI widgets, such as selecting a menu item or typing in a text box. The portion of the application code that executes in response to a GUI event is called the *event handler*. In Figure 1.1, several of the GUI’s widgets are circled; these circles or nodes can also represent events. Any path along the directed edges between events in Figure 1.1 would be a test case (e.g., *click Write Clue button, type in Input text box, click OK button*). GUI test cases are actually system tests, as they exercise the whole software, including non-GUI code.

The set of all valid event sequences that can be performed on a GUI can be described by a graph called the event-flow graph, which GUITAR can reverse engineer automatically from a GUI-based application. Figure 1.1 shows only a small portion of the event-flow graph for the example GUI; the full graph would include all events in all windows, with an edge connecting every pair of events that can be performed consecutively. Test cases for an application can be generated automatically by starting at any node (GUI event) in the event-flow graph and traversing edges leading to other GUI events. The first event in this sequence is called the *starting event*. To make each test case a valid one, a sequence of *reaching events*, which begins with an event that can be executed in the GUI’s initial state and ends with an event that the starting event can be executed immediately after, must be prepended to the test case. It is not strictly necessary to think of test cases as having reaching events—rather, the test case generator could always start traversing the event-flow graph at events available in the GUI’s initial state—but it conveniently allows test-case length to be defined such that any event can take part in any length of test case, regardless of the event’s distance from the GUI’s initial state. Thus, a *length- n test case* is defined to be any sequence of n events that can be traversed consecutively in the event-flow graph.

A GUI user receives output in the form of changes to the visible GUI windows and widgets, whose properties together make up the *GUI state*. The output of a test case is considered to be the sequence of GUI states that the application passes through. After each GUI event in the test case, the GUI state is checked. A test case detects a fault if the actual state is not as expected at any of

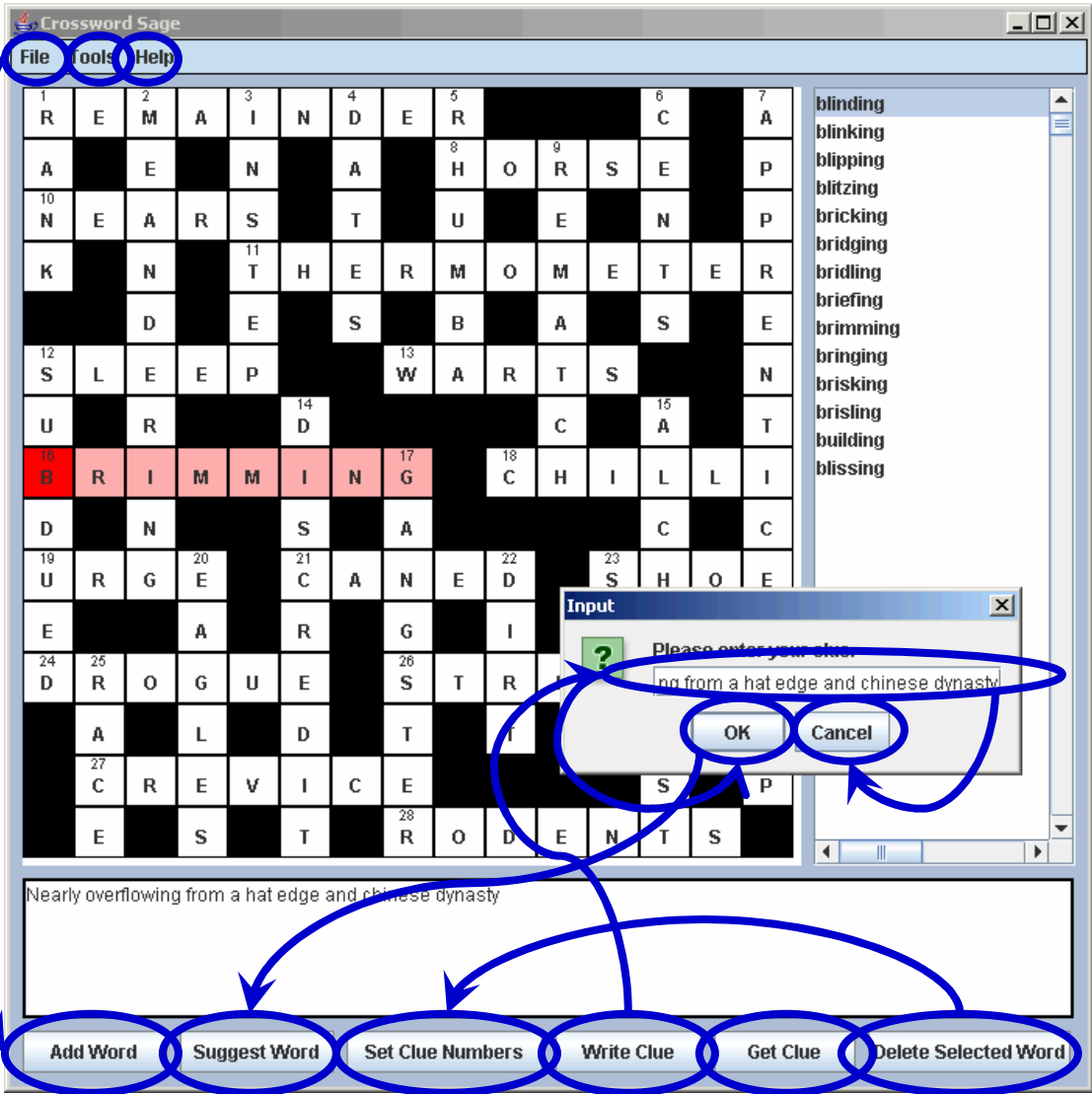


Figure 1.1: A GUI and part of its event-flow graph

those checks.

Although the empirical studies in this work propose to focus on model-based GUI testing, the proposed insights and theories that motivate the studies will have the potential to apply to other domains of testing as well. Even aspects of the empirical studies specific to GUI testing (e.g., characteristics of faults and test suites studied that are based on GUI events) may have analogues in more general testing. For example, the sequences of events in GUI testing may be analogous to sequences of method or function calls in other kinds of testing.

1.5 Proposed approach

As Section 1.1 stated, three important questions have yet to be fully answered by the software-testing literature: How do characteristics of test suites affect their ability to detect faults? How do characteristics of faults affect their detectability by test suites? And how should faults be characterized? Although these are difficult questions that will require the collaboration of numerous researchers to answer, the proposed approach is expected to make progress toward answering each of them in the domain of GUI testing. Two inter-related approaches to answering these questions will be presented and investigated empirically for real, GUI-based applications. In addition, a framework that embodies an application of one of these approaches to the problem of regression testing will be described and its feasibility demonstrated.

The first proposed insight shows how to evaluate testing techniques so that the results can be more readily generalized to new test subjects and interpreted in new situations. The key is, first, to understand the influences on testing techniques' effectiveness that can vary across testing instances and, second, to characterize the testing situation in which an evaluation is performed with respect to those influences. A researcher or practitioner in a different situation—with a different test subject or different assumptions about fault severity, for example—could then better predict, based on the differences between the two situations, how the evaluated technique would perform in

her case. This work will develop a methodology for studying the relationship between situational characteristics and testing techniques' effectiveness, focusing on characteristics of faults and test suites in GUI testing. Unlike in previous studies, faults and test suites will be studied as individuals, rather than as sets or samples, to reveal how characteristics of each fault and test suite affect fault detection.

Preliminary work [32] investigating that insight found that a fault's detectability by statement-coverage-adequate test suites was a strong predictor of its detectability by GUI test suites. Many others, too, have provided empirical evidence that the theoretical relationship between coverage and fault detection is relevant in practice [8, 11, 16, 23, 27]. Yet such studies do not satisfactorily explain why faults are or are not detected. Rather, they inspire further questions: why faulty parts of the program are or are not covered and why coverage of these faulty parts does or does not lead to fault detection (i.e., to a failure recognizable by the test oracle). Others have decomposed the relationship between faults and failures in this way before: Morell [22] with the notion of propagation of faulty values to outputs, Richardson and Thompson [29] with the RELAY model of fault detection, and Voas [34] with the PIE model of testability. In RELAY, for example, a fault leads to a failure in several steps: an expression is incorrectly evaluated, which causes further expressions to be incorrectly evaluated, which results in state failures (differences between the actual and expected state) and failures (differences between the actual and expected output).

The second insight that this work proposes to contribute is that the empirical investigation of fault detection should be decomposed into analogous sub-problems. Combined with the approach described above of studying $\langle test\ suite, fault \rangle$ pairs individually, this has the potential to lend unprecedented precision to the empirical understanding of fault detection in testing. Studying the probability that a program location is executed, given characteristics of the location and the test suite, will shed light on which testing techniques are better at covering which kinds of locations. Studying the probability that a fault at an executed location leads to a state failure will give researchers a more accurate way than previously used [1, 16, 26, 30] to estimate the semantic size

or prevalence of the fault with respect to a set of test cases. Finally, studying the probability that the state failure goes on to be detected will help reveal the role of factors other than test coverage level that affect fault detection—factors whose influence might otherwise be masked by that of test coverage level. One interesting class of situations to study would be those in which a test suite does not detect a certain fault even though the suite covers the piece of the program in which the fault resides.

To gain a thorough understanding, using the two insights above, of how fault and test-suite characteristics affect coverage and fault detection in a wide range of testing situations, numerous empirical studies by independent researchers will be required. To apply these results to a particular regression-testing situation, however, this work proposes a shortcut: a framework for adaptive, search-based regression testing that generalizes and improves upon existing regression-testing techniques.

A weakness of current regression-testing techniques is that they miss the opportunity with each iteration of regression testing to adapt the criterion for choosing a test suite. The criterion, such as coverage maximization, remains the same, despite the fact that other factors may also contribute to testing effectiveness. Thus, these techniques miss a chance to help the tester improve fault detection based on past experience and keep up with changes to the software and its development goals. This work proposes to remedy this weakness by combining and generalizing existing approaches to regression testing and adding a novel adaptive component to create a framework for adaptive, search-based regression testing. Unlike the existing approaches, the framework will allow for a definition of “good” test suites—i.e., a fitness function—that is specific to a software project and changes along with it. Unlike coverage- or fault-based approaches, the framework will permit other kinds of data, not just coverage or fault-detecting ability, to inform the choice of test suite. Furthermore, it will work both when the test cases used to construct the suite have been run before and when they are newly generated.

In the proposed framework for adaptive, search-based regression testing, a search-based tech-

nique will be used to discover a test suite that is predicted to best detect the kinds of faults the tester wants to target. The prediction will come from a statistical model built from data collected during past iterations of regression testing; the model will serve as the basis for the fitness function in the search. As more data is collected over iterations of regression testing, the fitness function will be updated. Thus, the search for a test suite at each iteration of regression testing is expected to improve and adapt over time. The proposed framework will not restrict the choice of search algorithm; any appropriate algorithm may be plugged in.

1.6 Preliminary work

The preliminary work, published in the proceedings of the International Conference on Software Testing, Verification, and Validation, 2008, addresses the first insight described above. It shows how understanding the context in which testing occurs, in terms of factors likely to influence fault detection, can make evaluations of testing techniques more readily applicable to new situations. It also presents a methodology for discovering which factors do statistically affect fault detection. In the methodology, characteristics of individual faults and test suites are related to the probability of fault detection for individual $\langle test\ suite, fault \rangle$ pairs. Using the methodology, experiments are performed for a set of test-suite- and fault-related factors in the GUI testing of two fielded, open-source applications. Statement coverage and GUI-event coverage are found to be statistically related to the likelihood of detecting certain kinds of faults.

1.7 Intellectual merit and broader impacts

The proposed work has the potential to improve the state of the art in software testing. By developing new ways to understand and study the effectiveness of software-testing techniques, this work is expected to help the research community discover which techniques are most effective

in different testing situations and why some techniques perform better than others. Using the insights to be investigated in this work—that empirical studies on software testing may be improved by characterizing results with respect to contextual factors of the study and by decomposing the study of fault detection into sub-problems based on models of faults and failures—and the methods proposed to investigate those insights, researchers will potentially be able to evaluate testing techniques with greater ease and generalizability than before.

The proposed work is expected to help advance not only research in software testing but also the quality of software in practice. The methodology presented in the preliminary work may be used by practitioners, not just researchers, to evaluate the merits of different testing techniques within the practitioners' context. To further help practitioners select the best testing technique for their unique situation—and to continually adapt the technique to the situation as it changes over time—the proposed research includes a framework for adaptive, search-based regression testing. The proposed framework, methodology, and research insights all will potentially lead to a better understanding by software testers of how to use testing resources most effectively in their situation, which in turn will lead to safer, cheaper, more correct software.

Chapter 2

Background and Related Work

This review of related work begins in Section 2.1 by surveying characteristics of testing situations that can affect the effectiveness of testing, with emphasis on characteristics of test suites and faults, as they are the focus of the proposed efforts to characterize testing situations. Next, in Section 2.2, an overview of existing fault characterizations relevant to software testing is presented. Because few fault characteristics have been studied in relation to fault detection in testing, the proposed work will improve upon current characterizations, which, as Section 2.2 will show, are inadequate. In Section 2.3, the process by which a fault in a program's source code leads to a failure in the program's execution is described. This process is the basis for a major part of the proposed work, which will decompose the problem of understanding factors that affect fault detection into the sub-problems of understanding factors that affect coverage, program state failures, and program output failures. Finally, to motivate the need for the proposed framework for adaptive, search-based regression testing, existing adaptive testing techniques will be reviewed in Section 2.4.

2.1 Factors shown to affect fault detection in testing

Test suites. Probably the most studied way that test suites can differ is in the technique used to make or vet them. In many studies, a sample of test suites yielded by a technique is used to evaluate the technique empirically against other testing or validation techniques. Techniques that have been compared in this way include code reading, functional testing, and structural testing [4]; data-flow- and control-flow-based techniques [16]; regression test selection techniques [11]; variations of mutation testing [27]; and strong and weak test oracles for GUIs [20].

Even when produced by the same testing technique, test suites can differ in important ways. Rothermel et al. [30] investigate how two test-suite characteristics, in addition to testing technique, affect the number of faults detected. One characteristic is *granularity*, a measure of the amount of input given by each test case. The other, *grouping*, describes the content of each test case and its meaning to testers (e.g., functional grouping, random grouping). For the applications studied, granularity significantly affects the number of faults detected. Grouping (functional vs. random) may also have an effect, though weaker.

Xie and Memon [36] investigate granularity and other test-suite traits in the arena of GUI testing. The variables of interest are the number of test cases and granularity of test cases, which they call *test-suite size* and *test-case length*, respectively. Test-suite size is found to affect the number of faults detected, while test-case length affects the kind of faults detected: some faults can only be reached by longer test cases. The authors conjecture that faults that can be detected by shorter test cases lie in event handlers with less complex branching than those that can only be detected by longer test cases.

In another study of GUI testing, McMaster and Memon [19] show that the coverage criterion used in test-suite reduction affects the size of the reduced suites and the average probability of detecting each fault. Certain faults were found to be more consistently detected by some coverage criteria than by others. Across all faults, suites reduced with call-stack and event-pair coverage had

the highest average probability of detecting each fault. Event, line, and method coverage resulted in smaller reduced suites but also lower average probabilities of detecting each fault.

A study by Elbaum et al. [8] applies principal component analysis and regression analysis to a large set of test-suite characteristics related to regression testing. Of the characteristics studied, two related to coverage—the *mean percentage of functions executed per test case* and the *percentage of test cases that reach a changed function*—best explain the variance in fault detection. The other characteristics studied (which include the number of test cases in the suite, the number of functions and statements executed per test case, and the number of changed functions and statements executed per test case) turn out to be less influential.

Applications. In the study by Elbaum et al. [8] described above, characteristics of the application under test are also investigated. The characteristics describe the size and complexity of applications and changes made to them. Of the characteristics studied, the *mean function fan-out* and the *number of functions changed* together explain the most variance in fault detection.

Morgan et al. [23] also investigate how application characteristics and test-suite characteristics jointly influence fault detection. Here, the variables of interest are *test-suite size*, *proportion of application units* (blocks, decisions, and variable uses) *covered*, and *application size* measured in lines and application units. In quadratic models fit to the data set, each characteristic by itself (i.e., linear term in the model) and some squares or products of characteristics (i.e., quadratic terms) are found to contribute to the variance in fault detection, although the influence attributed to test-suite size alone is slight.

A study by Ostrand et al. [28] of several sizable software systems suggests another way in which application and test-suite characteristics can interact. In this study, a statistical model whose parameters are properties of individual files in the system (e.g., file age, number of lines of code, number of faults found in earlier versions of the file) predicts which files contain the greatest number or highest density of faults. If a test suite targets those files, rather than spreading coverage evenly across the system, then it is likely to detect more faults.

Research into software testability offers yet another perspective on the influence of application characteristics on fault detection. Voas [33] describes characteristics of applications that affect the flow of internal program information to output—specifically, the domain-range ratio of a specification and the amount of state information checked during testing. While several design heuristics to reduce information loss from that flow are presented, the effect of the characteristics or heuristics on fault detection in real programs is not evaluated. In an empirical study of several applications and the JUnit test suites developed for them, Bruntink and Deursen [5] find that several object-oriented metrics—specifically, measures of the size, external dependencies, and quality of classes—are related to the size of these test suites.

Faults. Chapter 1 argued that empirical evaluations of testing techniques are often more generalizable if the reported results for fault detection are broken down by fault type. A few studies do this, including Basili’s and Selby’s [4] comparison of code reading, structural testing, and functional testing. Two orthogonal fault taxonomies are used to characterize the faults studied. One classifies faults as either omissive or commissive; the other, as initialization, control, data, computation, interface, or cosmetic. In some cases, the validation technique and the fault type appear to interact in their influence on fault detection.

In a study comparing data-flow and mutation testing, Harrold et al. [13] classify detected faults using a different taxonomy. Classes of faults are defined by their effect on the program dependence graph, a representation of the data and control dependencies in the application under test. At a coarse level, the taxonomy classifies faults as either *structural*—altering the structure of the program dependence graph—or *statement-level*—altering a statement but leaving the graph structure unchanged. Unfortunately, to our knowledge, no tools that implement this analysis for Java applications are currently available. Furthermore, because mutation testing detected all 74 faults and data-flow testing detected all but two, the taxonomy hardly helped differentiate between the techniques.

Offutt and Hayes [26] recommend that faults be characterized by their *semantic size*, which can

be thought of as the probability that a random test case detects the fault. Similar measures have been used in empirical studies by Andrews et al. [1], Hutchins et al. [16], and Rothermel et al. [30] to characterize faults' ease of detection with respect to the test pools used in the studies. However, as Section 1.1 noted, such characterizations are weakened by their relativity to the sample of test cases used to calculate them.

2.2 Fault characterizations

This section reviews fault characterizations that have been used in software-testing studies with respect to four criteria: (1) they should provide information useful to testing, (2) they should be consistent and objective, (3) they should be automatable, and (4) they should not rely on artifacts such as formal specifications that may not be available. As Section 3.1 will explain, these criteria are necessary for a fault characterization to lead to more generalizable results in empirical studies of testing techniques.

Viewing faults syntactically and semantically. Offutt and Hayes [26] distinguish between *syntactic* and *semantic* characterizations of faults. Syntactically, a fault is a difference in source code between a correct and an incorrect version of a program. The fault's *syntactic size* is, roughly speaking, the size of the textual difference in source code. The categories of mutation faults are a prime example of a fault characterization based on a syntactic perspective. Semantically, a fault is a difference in the input-output mappings that the correct and incorrect program versions induce, and its *semantic size* is, again, the size of the difference.

As Offutt and Hayes [26] point out, of the two views, the semantic view is more closely related to program execution and, hence, to testing. Yet the syntactic view persists in the testing literature, perhaps because it aligns with a programmer's perspective during debugging. The trouble with syntactic characterizations of faults is that their relation to fault detection does not translate across contexts, or even across different parts of the same program—failing to meet Criterion 1.

```
1 x = null;
2 if (false) {
3     if (x == null) // if (x != null)
4         x.foo();
5 }
6 if (x == null) // if (x != null)
7     x.foo();
```

Figure 2.1: In lines 3 and 5, the same mutation operator has been applied, changing `!=` to `==`. The program never fails at line 3 but always fails at line 5.

```
1 print("Pick a number between 1 and 100.");
2 read(number);
3 if (number == 4058295011)
4     launchMissiles(); // Oops
```

Figure 2.2: The fault in line 4 is semantically small but potentially costly.

Figure 2.1 demonstrates this.

As a schema for characterizing faults, semantic size also presents problems. In an empirical study, Offutt and Hayes [26] measure the semantic size of faults by executing the correct and faulty programs with a random sample of inputs. The main problem with this measure is that it is poorly defined, in that it does not specify how many inputs should be sampled. A secondary problem is that a fault's semantic size is not necessarily related to its elusiveness, assuming one is using a technique more sophisticated than random or weighted random testing. Nor is semantic size necessarily related to cost, as Figure 2.2 shows. This is true even if the random sampling of inputs is based on operational profiles. To summarize, semantic size fails to meet Criterion 2 and, arguably, Criterion 1.

Viewing faults through program models. Along the spectrum from syntactic to semantic views of faults lie interpretations of faults based on program models. Like the syntactic perspective, these interpretations are based on static properties of a faulty program, but, like the semantic perspective, the properties of interest are closely related to the program's dynamic behavior. Harold et al. [13] offer a prime example of such an interpretation. Refining a fault model by How-

den [14], which classifies faults by their effect on the paths followed and the function computed by the program, they further classify faults as either *structural* or *statement-level* for the purpose of fault seeding. The latter classification is based on the way a fault affects the program dependence graph, a representation of the control and data dependencies in the program: structural faults alter the graph, while statement-level faults leave it unchanged.

It should be clear that the approach of Harrold et al. meets Criteria 2, 3, and 4. What about Criterion 1? This approach may indeed provide useful information about the cost of detecting and repairing a fault. For example, an abundance of faults of class *Incorrect Expression in Predicate* in a program's latest release may persuade its testers to use a more rigorous predicate-coverage criterion for the next version. As for the cost of repairing a fault, the program dependence graph provides the information needed to perform a change-impact analysis of the faulty node(s). However, in the empirical study presented along with the program-dependence-graph taxonomy, the taxonomy turned out not to distinguish between the two testing techniques studied, data-flow and mutation testing. Thus, the taxonomy's potential impact on software-testing studies remains unproven.

Faults may be characterized in terms of other graph models as well. For example, Dinh-Trong et al. [6] propose a taxonomy of mutation faults for UML class diagrams. One can imagine a rudimentary fault taxonomy that would apply to any kind of graph model, with categories like *Extra Node (and Edge)*, *Missing Edge*, and *Intra-Node Fault*. Whether such a taxonomy would meet the four criteria listed earlier in this section depends in part on the taxonomy's relationship to development or maintenance tasks (Criterion 1) and the graph model's ability to be generated automatically from the program (Criterion 3).

Other ways to characterize faults. In an empirical comparison of structural testing, functional testing, and code reading, Basili and Selby [4] classify the faults studied using two orthogonal schemes. One scheme has the categories *Omissive* and *Commissive*; the other, categories *Initialization*, *Control*, *Data*, *Computation*, *Interface*, and *Cosmetic*.

Both schemes miss some of the criteria listed at the beginning of this section. To decide whether to label a fault *Omissive* or *Commissive*, one has two options: analyze the fault’s relation to the program’s specification, either by obtaining it or by inferring it (violating Criteria 4 or 2, respectively), or look at syntactic characteristics of the fault, i.e., whether code was added or modified to fix it (violating Criterion 1). (The second option is reminiscent of Munson’s and Nikora’s [24] method for counting faults.) In the second fault-classification scheme, the boundaries between some of the six categories are fuzzy, leaving them open to interpretation (violating Criterion 2). For example, passing an array of uninitialized values to a function that assumes the array is initialized with zeros could be either an *Initialization* fault or an *Interface* fault. Basili and Selby [4] themselves note that their classification schemes are somewhat subjective.

Parts of IEEE Standard 1044-1993 (“IEEE Standard Classification for Software Anomalies”) suffer from similar problems. Of the high-level categories in this classification, those that pertain to faults are *Logic*, *Computation*, *Interface/Timing*, *Data Handling*, and *Data*. Each category is further divided into sub-categories. The illustration above of the ambiguity between the *Initialization* and *Interface* categories applies here as well (violating Criterion 2), since *Data Handling* and *Interface/Timing* have equivalent sub-categories. Some other sub-categories could not always be identified objectively and automatically (violating Criteria 2 and 3), at least not without detailed change logs (violating Criterion 4). These sub-categories (and their parent categories) include *Misinterpretation (Logic)*, *I/O Timing Incorrect (Interface/Timing)*, *Data Referenced Out of Bounds (Data Handling)*, and *Output Data Incorrect or Missing (Data)*.

In practice, faults are often classified by their severity. Unfortunately, the notion of severity translates poorly from one context to another, as it depends on factors such as who is using the system and how. Operational profile testing [25] may detect the faults that users are most likely to encounter, but even these faults are not necessarily the most severe (cf. Figure 2.2). Thus, fault severity fails to satisfy Criterion 4.

2.3 Faults' relation to failures

Morell [22] described the conditions under which a fault would become a failure in terms of the propagation of symbolic values to output expressions in symbolic executions of the program under test. Richardson and Thompson [29] refined Morell's theory into the RELAY model of fault detection. In RELAY, a fault leads to a *potential failure* when it causes a test case to incorrectly evaluate an expression; when the expression is the smallest one containing the fault, the potential failure is said to *originate*. The potential failure may then *transfer* to other parts of the program and lead to *state failures* (differences between the actual and expected state) and failures (differences between the actual and expected output). Voas [34] used the key steps in the RELAY model to create the propagation, infection, and execution (PIE) analysis of program testability. In PIE, the probability that a hypothetical, arbitrary fault at a program location would result in a failure is decomposed into three sub-problems: the probability that the location is executed, the probability that a fault at the location leads to a state failure when executed (infection), and the probability that a state failure following execution of the location leads to a failure (propagation).

2.4 Adaptive testing techniques

In the proposed framework for adaptive, search-based regression testing, information about past regression-testing iterations will be used to inform and improve future iterations. However, the idea that feedback from iterative processes can be used to improve them is not new. This principle underlies existing regression-testing techniques that carry over coverage or fault-detection information from one iteration to the next [8, 18], which the proposed framework generalizes. At a smaller scale, iterative learning has been used to generate test cases. At a larger scale, it is the basis of Basili's [3] Improvement Paradigm.

In the Improvement Paradigm, the cycle of observing and improving processes occurs at the

level of software projects. When planning a new project, an organization draws from its experience with past projects, taking into account the context of the new project (e.g., available resources, product size). During and after the project's lifetime, the knowledge gained about the methods and techniques used by the project is structured for reuse and added to an experience base [3]. In structure, the proposed framework is patterned after the Improvement Paradigm, but instead of projects, it considers regression-testing iterations.

Several test-case-generation techniques similarly operate in cycles of observation and improvement. For these, the objects of interest are not software projects or regression-test suites, but individual test cases. Interleaving test execution with test-case generation, these techniques progressively find test inputs to cover application states that have not yet been tested. The selection of test inputs may be guided by various kinds of information gained from test execution, including data dependencies [9], numerical constraints [12], and GUI-state changes [37]. Cai et al. [17] also use feedback from test execution to select additional test cases, but their goal is to accurately estimate defect-detection rates using as few test cases as possible.

2.5 Summary

The previous research surveyed in this chapter motivates the need for the work proposed in the next chapter. Section 2.1 described the current body of knowledge on how characteristics of testing situations can affect the ability of testing techniques to detect faults. This body of knowledge is far from comprehensive, particularly with regard to characteristics of faults. In fact, as Section 2.2 showed, it is not even clear how faults should be characterized in software-testing studies. This work proposes to address these problems (within a restricted domain to make the work feasible) by contributing both concrete data to the body of knowledge and more general insights into how such knowledge can be generated and understood more effectively. In order to accomplish this, a system of characterizing faults will need to be developed in this work. Section 2.3 explained the

process by which a fault leads to a failure. In the proposed work, an understanding of this process will be incorporated into a method for studying the effects of characteristics of testing situations on testing outcomes in order to yield more precise results. Finally, Section 2.4 described existing testing techniques that adapt over time to improve fault detection. As that section demonstrated, the framework for adaptive, search-based regression testing proposed in this work will fill a niche left open by existing techniques.

Chapter 3

Goals and Approaches

3.1 Goals

This work aims to shed light on the role faults play in software testing. Many characteristics of testing situations—including characteristics of test suites and applications—have been shown to affect the effectiveness of testing at detecting faults, yet characteristics of faults themselves have not been extensively explored. The proposed work will produce a clearer empirical understanding of the relationship between characteristics of faults and their detection during testing. To take advantage of existing tools and resources and to limit the proposed work to a feasible scope, the work will focus on testing in the domain of graphical user interfaces (GUIs).

The proposed work will explore two insights that are expected to lead to a better understanding of the relationship between characteristics of testing situations and the effectiveness of testing. The first insight is that testing situations should be characterized, and results of empirical studies of those situations should be presented in terms of those characterizations, so that researchers or practitioners in different testing situations can better understand how the study results would translate to their situations. In this work, the characteristics of testing situations studied will be those of faults and test suites in GUI testing.

In order to characterize faults for this work, a system of characterization must first be developed; as Section 2.2 asserted, current fault characterizations are inadequate for this research. To answer a tester’s most important question about empirical evaluations of testing techniques—*How will the technique perform in my context?*—the faults that the testing technique detects must be characterized in a way that meets the following criteria (discussed earlier in Section 2.2):

1. provides useful information about the prevention, detection, repair, or cost of the faults,
2. is consistent and objective,
3. can be automated, and
4. does not rely on formal specifications or other information that may not be available.

Criterion 1 is clearly vital; otherwise, characterizing faults would add no value to software-testing studies. Without Criteria 2 and 3, which are closely linked, researchers and practitioners would likely balk at the cost of integrating fault characterizations into their work. Criterion 4 ensures that a fault characterization can be applied to the large population of programs, including open-source software from sites such as <http://sourceforge.net>, that lack formal specifications and other artifacts from the development process.

The second insight that this work proposes to explore is that the problem of studying faults’ detection empirically can be broken down into sub-problems following models of fault detection like RELAY [29]. Factors that affect the probability that a faulty program element is executed during testing can be studied separately from those that affect the probability that execution of a faulty program element leads to an unexpected program state or output. This will potentially lead to a more thorough understanding of the relationship between coverage and fault detection in software testing, and it is expected to help expose the role of factors whose influence on fault detection may be obscured by the overwhelming influence of coverage.

In addition to these two insights, this work proposes to develop a practical application of the first insight. The application is a framework for adaptive, search-based regression testing, which is expected to generalize and improve upon existing regression-testing techniques by enabling

testers to learn from previous testing iterations how to build or select test suites that detect faults more effectively. Unlike existing regression-testing techniques, the proposed framework will be designed to allow the kinds of test suites used during testing to adapt to the unique context of each testing situation—including the kinds of faults that are most important to detect—and evolve along with the application under test and the needs of testers.

The goals of this study—the two insights and the practical application just described—will be investigated in a series of empirical studies on real, GUI-based applications. The procedure for designing and performing these empirical studies is outlined in the tasks below. Because the empirical studies are expected to be similar in design, each of the tasks may be revisited for each goal.

3.2 Tasks and approaches

Task 1: Select the set of fault characteristics to study.

Approach: A primary goal of this research is to study the relationship between characteristics of faults and the likelihood of faults' detection during testing. A critical step, then, is to choose a set of characteristics to study that satisfies the criteria outlined above (applicable to testing, objective, automatable, and computable from source code). Although no set of characteristics satisfying these criteria has been provided in the literature so far, previous research will inform both the choice of characteristics and the methodology used to study them.

Previous research on software testing points to several types of fault characteristics that may be interesting to study. For example, Xie and Memon [36] conjecture that faults that can be detected by shorter test cases lie in event handlers with less complex branching than those that can only be detected by longer test cases. Thus, characteristics describing the complexity or reachability of the code in which the fault is embedded should be investigated. Extensive empirical confirmation of the close relationship between coverage and fault detection [8, 11, 16, 19, 23, 27] hints not only

that characteristics describing the coverability of faulty program elements—and their interactions with the coverage provided by test suites—should be studied, but also that characteristics of faulty program elements affecting coverage should be studied separately from characteristics affecting fault detection to factor out the strong effect of coverage on the latter.

Because there are so many possible fault characteristics to study, and because their relationship to coverage and fault detection is at this point hypothetical, the process of selecting a set of characteristics is expected to be iterative. As empirical results are obtained for these characteristics, the set will be refined to include new characteristics and exclude characteristics shown to be irrelevant.

Task 2: List test-suite characteristics that may interact with fault characteristics to affect testing outcomes.

Approach: To fully understand the relationship between fault characteristics and testing outcomes, it is necessary to see whether this relationship is consistent across different kinds of test suites. Like the previous task, this task will involve a survey of previous research. Numerous empirical studies have found that the level of coverage [8, 11, 16, 19, 23, 27] and other characteristics of test suites [8, 11, 20, 30, 36] can affect fault detection. For this research, a manageable set of characteristics applicable to GUI testing will be selected. Since most of these characteristics have not been rigorously studied for GUI-based applications, an added contribution of this research will be to provide statistical results for the relationship between these characteristics and testing outcomes in the domain of GUI testing.

Task 3: Investigate statistical methods for studying the effects of characteristics on fault detection.

Approach: To discover whether apparent differences among faults with different characteristics are meaningful, appropriate statistical tests will need to be performed. Since statistical methods typically assume that the data satisfies certain assumptions, the choice of method will depend on the expected properties of the data and will influence the design of the experiments to be performed. To control the levels of statistical significance and power in the experiments, a sample size calculation will need to be performed for the sets of test-suite and fault characteristics to be

studied.

Task 4: Design empirical studies based on the chosen statistical methods.

Approach: Experiments using applications, faults, and test suites will be designed to answer the questions posed in this research—namely, whether each fault and test-suite characteristics studied has a statistically and practically significant effect on coverage or fault detection. To calculate the appropriate sample size for statistical testing, a pilot experiment will need to be performed. Its design will be similar to the full experiments, except that the sample size for the pilot will be chosen, by necessity, somewhat arbitrarily. A further empirical study will be designed to demonstrate the feasibility of a proposed application of this work to practical software development—the framework for adaptive, search-based regression testing.

Task 5: Choose subjects on which to carry out the empirical studies.

Approach: Software applications will be selected to study empirically. The selected applications will be GUI-intensive, realistic in size and functionality (to be distinguished from “toy applications”), and freely available to other researchers. An ideal source for such applications is <http://sourceforge.net>. An additional requirement of the selected applications is that they work with the GUITAR tools, since these are research tools with certain practical limitations (e.g., they can recognize only a limited set of GUI widget types).

Task 6: Build infrastructure and create artifacts for experimentation.

Approach: Preparing to perform the experiments on the subject applications will be a complex task. To construct test suites for each application, the GUI will first need to be reverse-engineered using GUITAR to create an event-flow graph. Then, using a custom-built test-case generator if necessary to conform to the experimental design, test cases will be generated. These will be combined into test suites following the experimental design. To obtain a sufficiently large sample of known faults for each application, faults will be seeded using a mutation tool such as MuJava¹. (Artificial mutation faults have been shown to behave virtually the same as real faults in software-

¹<http://www.ise.gmu.edu/~ofut/mujava/>

testing experiments [1, 2, 7].) To collect code coverage information, the subject applications will be instrumented using a tool such as Instr².

A major challenge in performing the proposed empirical studies will be to run potentially hundreds of test cases on the “clean” version of each application (to be used as the test oracle) and the dozens of versions in which faults have been seeded. Although the tasks of running test cases and comparing results to the oracle are automated by GUITAR, each test case still takes a non-trivial amount of time to run, in part because of the time required for GUI windows to load. To reduce the overall time required for test execution to a feasible amount, test cases will be executed in parallel on the distributed system Condor³. An infrastructure of scripts will need to be written to package application versions and test cases for execution on Condor and to process and combine the results from the Condor nodes. Since this will be the first research to use GUITAR and Condor together, the process of test execution itself will need to be tested to verify that correct results are being produced.

Task 7: Perform experiments and analyze data.

Approach: Since the infrastructure described in the previous task will automate most of the experimental procedure, the main tasks here will be to clean the resulting data and analyze it according to the chosen statistical methods. To clean the data—a task that will be necessary if some of the results for fault detection are found to be incorrect, as happened in the preliminary work—results for fault detection will be checked against coverage information to eliminate reports of fault detection where the test case did not actually cover the faulty part of the application. For the data analysis, the R software environment⁴ will be used to perform statistical calculations.

Task 8: Manually inspect individual faults to validate and understand the results.

Approach: To better understand why certain test cases covered or detected certain faults in the empirical studies and to look for potentially distinguishing fault or test-suite characteristics that

²<http://www.glenmcc1.com/instr/index.htm>

³<http://www.cs.wisc.edu/condor/>

⁴<http://www.r-project.org>

were left out of the sets of characteristics studied, some of the results will be manually inspected. Because the fault matrices (records of which test case detected which fault) in these empirical studies are expected to be large (at least on the order of ten thousand entries), a system will need to be built that will allow the user to easily view the characteristics of each fault, the change the fault induces on the application's source code, and the test cases that cover or detect it.

Task 9: Explore the proposed approaches as applied to other kinds of applications and test suites.

Approach: To gain some understanding of the extent to which the results of the above empirical studies apply to domains other than GUI testing, the applicability of the fault and test-suite characteristics studied to other kinds of applications and test suites will be investigated. This may involve replicating some of the above experiments for these new applications and test suites and comparing the results to see whether the domain affects the relationship between fault and test-suite characteristics and testing outcomes. One promising source of alternative applications with fault sets and test cases is the Software-Artifact Infrastructure Repository (SIR)⁵.

⁵<http://sir.unl.edu>

Chapter 4

Preliminary Work

This chapter presents preliminary work, which was published in the proceedings of the International Conference on Software Testing, Verification, and Validation, 2008, under the title “Relationships Between Test Suites, Faults, and Fault Detection in GUI Testing”. The preliminary work pursued the first insight described in Section 3.1—that characterizing the context in which software-testing studies are performed and presenting results in terms of that characterization can lead to results that are more precise and more generalizable. The research contributions of the work included a methodology for using such characterizations in software-testing studies and an experiment that applied the methodology to study the relationship between test-suite and fault characteristics and fault detection in two real, GUI-based applications.

4.1 Introduction

For conclusions about fault detection drawn from a sample of faults and a sample of test suites to be generalizable to other sets of faults and other test suites, (1) measurable characteristics of faults and test suites that may be related to fault detection must be identified, (2) the influence of these characteristics, independently or jointly, on fault detection must be validated in a small set

of contexts, and (3) the influence on fault detection must be established empirically or analytically (or preferably both) on a large set of contexts.

This work makes progress on (1) and (2), focusing on GUI testing, in which test cases consist of sequences of events that a user might perform on a GUI. First, a set of characteristics of faults and test suites expected to affect fault detection is identified: for faults, branch points nearby, probability of detection by other techniques, and type of mutant (method- or class-level); and for test suites, length of test cases, size, event-pair coverage, and event-triple coverage. Let it be emphasized that this study is a first step in a direction requiring much future research, so the set of characteristics studied is not meant to be comprehensive, only sufficient to demonstrate a methodology for studying such characteristics. Second, the relationship between these characteristics and fault detection is statistically analyzed for two fielded, open-source, Java applications, using logistic regression analysis to isolate the effect of each. The results show that a fault's detectability by statement-coverage-adequate test suites and, for one application, a fault's mutant type, a test suite's event-triple coverage, and the interaction between mutant type and detectability by statement-coverage-adequate suites, are significantly related to the probability of fault detection by GUI test suites.

Although this work focuses on GUI testing so that the experimental infrastructure can sit atop the existing GUI Testing Framework (GUITAR) [36], the approach used here is not limited to this domain. In fact, this work makes several contributions to research on software testing and software defects, including:

- a methodology for studying statistical relationships between test-suite and fault characteristics and fault detection and
- an experiment that uses the methodology to show which of a set of test-suite and fault characteristics are statistically related to fault detection for two fielded applications.

4.2 Variables of interest

As research into factors affecting fault detection matures, it is hoped that a paradigm for selecting characteristics to study will emerge. But for now, while the selection for this work is rooted in the literature, it must remain somewhat ad hoc. To make this work more replicable, the characteristics chosen were required to be measured objectively, automatically, and without special artifacts such as specifications [31].

Chapter 2 named several characteristics of GUI test suites that can affect the number and kinds of faults a suite detects. This study examines four such characteristics: length of test cases (**Len**), size (**Size**), event-pair coverage divided by test-suite size (**E2Cov**), and event-triple coverage divided by event-pair coverage (**E3Cov**). These are summarized in Table 4.1. The length of a test case is the number of events it contains; in our study, all of the test cases in a suite have the same length. (This assumption would not necessarily hold for real test suites, but it makes it possible to isolate length as a variable for the purposes of the experiment.) The size of a test suite is the total number of events the test suite executes, counting duplicates; the suite size of ten length-two test cases would be twenty. Event-pair coverage and event-triple coverage are the numbers of unique length-two and length-three event sequences, respectively, in the test cases that comprise a suite. These two variables are normalized by test-suite size and event-pair coverage, respectively, to avoid confounding their influence on fault detection with that of test-suite size and of each other. This work could have looked at coverage of length-four and longer event sequences but chose not to until it had been verified that coverage of shorter sequences was related to fault detection. Since all test suites in the study contain each event at least once, no information about the events, such as their complexity or content, is included among the variables.

The influence of fault variables on a fault's chances of detection is also investigated. Table 4.1 summarizes the fault variables studied. As Chapter 2 mentioned, it has been speculated that the number of branch points in an event handler (**Branch**) together with test-case length is related

Table 4.1: Study variables

	Abbrev.	Description
Test suite	Len	Length of test cases
	Size	Size (number of events)
	E2Cov	Event-pair coverage / size
	E3Cov	Event-triple coverage / event-pair coverage
Fault	Mut	Mutant type (method- or class-level)
	Branch	Branch points in faulty method’s bytecode
	StmtDet	Estimated probability of detection by statement-coverage-adequate test suite

to the probability that a fault in the event handler is detected. In prior work [31], it has been asserted that an effective way to characterize the faults detected by a testing technique is in terms of their detection by other techniques. Hence, this work considers the ability of statement-coverage-adequate test suites (**StmtDet**) to detect a fault. The faults used in this study are generated by class-level and method-level mutation operators, which change the structure of the program differently in ways that may affect fault detection, so each fault is classified accordingly (**Mut**).

4.3 Study design

The goal here is to learn which of the variables listed in Table 4.1 affect the likelihood that fault detection occurs in a $\langle test\ suite, fault \rangle$ pair. In doing this analysis, there are several tricky points. First, to isolate the effect of each individual variable, the values of the remaining variables must be accounted or controlled for in some way. This presents a challenge because several of the variables (e.g., **E2Cov**) can take on hundreds of values (theoretically), and any categorization of these values (e.g., into “low”, “medium”, and “high” event-pair coverage) would be artificial. Second, *interaction effects* among variables may occur. For example, increasing the values of two variables at once may either amplify or suppress the variables’ individual effects on the dependent.

These points have led this study to be conducted by collecting a random sample of $\langle test\ suite, fault \rangle$ pairs; measuring the values of the variables in Table 4.1, as well as fault detection (a Boolean

value) for each pair; and analyzing the data set with logistic regression to test hypotheses about the independent variables. Each of these steps is now explained in turn.

4.3.1 The Sample of $\langle \textit{Test Suite}, \textit{Fault} \rangle$ Pairs

Subject applications. Two open-source applications from SourceForge¹ serve as subjects for this study: CrosswordSage (CWS), a crossword-design tool; and FreeMind (FM), a tool for creating documents called “mind maps”. Both are implemented in Java and have a GUI. These applications’ availability to the public and their use in previous research [37] make them attractive candidates for this study. Table 4.2 gives each application’s size in non-commented, non-blank lines of code (LOC) and in classes (Cls.).

Sample size. The sample of $\langle \textit{test suite}, \textit{fault} \rangle$ pairs must be large enough to provide the desired levels of significance ($\alpha = 0.05$) and power ($1 - \beta = 0.80$) when the independent variables’ influence on the dependent variable is not too faint. The faintest detectable influence is a function of the *effect size*, the minimum coefficient magnitude of interest in the logistic regression model (Section 4.3.3). Typically, a researcher fixes the effect size at the smallest value of practical significance. Having no precedent in the software-testing literature, the effect size selected is 0.3, which seems reasonable and results in a feasible sample size. The levels of significance and power limit the probabilities of Type I and Type II errors to 0.05 and 0.20, respectively. Both errors have to do with “unlucky” samples. A Type I error occurs when a relationship in the sample data can be found, but no such relationship exists in the population (i.e., the null hypothesis is spuriously rejected). A Type II error occurs when a relationship does exist in the population, but it cannot be found in the sample (i.e., the researcher erroneously fails to reject the null hypothesis) [10].

The necessary sample size is estimated by applying the procedure outlined by Hsieh et al. [15] to data from a pilot study. In the pilot study, a sample of 100 $\langle \textit{test suite}, \textit{fault} \rangle$ pairs is analyzed for

¹<http://sourceforge.net>

one subject application, CWS, following procedures for test-pool generation, test-suite construction, and fault seeding similar to those described in the rest of this section. From this data, for the significance level, power, and sample size noted above, the sample size turns out to be 146. Details of the sample size calculation are given below.

The following formulas are due to Hsieh et al. [15] and were implemented for this work in the R software environment. The sample size n required to test one independent X in the presence of the other independents is

$$n = \frac{n_1}{1 - R^2}$$

where n_1 is the sample size that would be required to test X if it were the only independent.

For continuous independent variables,

$$n_1 = \frac{(Z_{1-\alpha/2} + Z_{1-\beta})^2}{r_{\bar{X}}(1 - r_{\bar{X}})\beta^{*2}}$$

where Z_u is the u th percentile of the standard normal distribution and β^* is the effect size. The value of $r_{\bar{X}}$, the probability of fault detection at the mean value of X , is estimated by fitting a logistic regression model to the pilot-study data for X and the dependent variable, then plugging the sample mean of X in for X .

For dichotomous (boolean) independent variables,

$$n_1 = \frac{(Z_{1-\alpha/2}V^{1/2} + Z_{1-\beta}W^{1/2})^2}{(r_0 - r_1)^2(1 - s_1)}, \text{ where}$$

$$V = \frac{r(1 - r)}{s_1}$$

$$W = r_0(1 - r_0) + \frac{r_1(1 - r_1)(1 - s_1)}{s_1}$$

In these formulas, s_1 is the proportion of the pilot-study data with $X = 1$; r_0 and r_1 are the

Table 4.2: Size of applications, test suites, and test pools.

App.	LOC	Cls.	TS(↓)	TS(↑)	TP(2)	TP(20)
CWS	3220	36	8	46	402	226
FM	24665	858	117	424	1093	455

empirical probabilities of fault detection when $X = 0$ and $X = 1$, respectively; and $r = (1 - s_1)r_0 + s_1r_1$ is the overall empirical probability of fault detection.

This takes care of n_1 , leaving R^2 , the squared multiple correlation coefficient relating X to the rest of the independents. In the R environment, R^2 is calculated as a side effect of fitting a linear model, in which X is predicted by the rest of the independents, to the pilot-study data.

Finally, to obtain a sample size for this study, n is found for each independent variable and the maximum of these is taken.

Test pool. As just explained, over 100 test suites must be run for each subject. A test suite can consist of hundreds of test cases, and each length-twenty test case takes a few minutes to run. If each additional test suite in the study required hundreds of additional test cases to be selected from the test-case domain and run, evaluation of the 100-plus test suites would be infeasible. The task is made feasible by restricting the test-case domain to a relatively small set called the *test pool*. The test pool must be small enough that all of its test cases can be executed in a reasonable amount of time, yet large enough that test suites picked from the pool are sufficiently different from one another. The latter requirement depends on the number of test cases per test suite. Table 4.2 lists the minimum and maximum number of test cases per test suite (TS(↓) and TS(↑)) for each application.

Figure 1 shows the algorithm used to construct the test pool, which results in nineteen “buckets” of test cases, one for each length. Test cases do not exceed twenty events because GUITAR often fails spuriously from timing problems with longer test cases. The minimum test-case length is set at two, rather than one, because it was anticipated that the sufficient test-pool size for some applications would exceed the number of possible length-one test cases—yet it was desired that

Algorithm 1 Algorithm for constructing the test pool. $succs(event)$ is the set of successors of the event in the EFG. For a test case in $bucket_i$, $last(testCase)$ is the i th (i.e., last) event in the test case. and $covSuccs(testCase)$ is the set of events such that $testCase \circ event \in bucket_{i+1}$.

```

1:  $bucket_1 \leftarrow \{\text{all events in application}\}$ 
2:  $i \leftarrow 0$ 
3: while  $i < \text{iters}$  do
4:   for  $tc \in bucket_1$  do
5:      $tc' \leftarrow tc$ 
6:     for  $i$  from 2 to 20 do
7:        $uncov \leftarrow succs(last(tc')) - covSuccs(tc')$ 
8:       if  $uncov = \emptyset$  then
9:          $covSuccs(tc') \leftarrow \emptyset$ 
10:         $uncov \leftarrow succs(last(tc'))$ 
11:       end if
12:        $e \leftarrow \text{random event in } uncov$ 
13:        $covSuccs(tc') \leftarrow covSuccs(tc') \cup e$ 
14:        $tc' \leftarrow tc' \circ e$ 
15:        $bucket_i \leftarrow bucket_i \cup tc'$ 
16:     end for
17:   end for
18:    $i \leftarrow i + 1$ 
19: end while

```

each bucket in the test pool to initially contain an equal number of test cases with no duplicates. The test cases are built such that each length-two to length-nineteen test case is a prefix of a length-twenty test case, which permits a time-saving shortcut in test-case execution: the study needs only to run the length-twenty test cases, checking the GUI state after each intermediate event, to obtain results for the length-two to length-nineteen test cases as well. The number of iterations, *iters*, was chosen to limit the probability that two test suites picked from the pool would share more than a small percentage of test cases. After the test cases were executed, however, the pool size was reduced: test cases that failed spuriously on the i th event (as determined by examining statement coverage of lines with seeded faults) were removed from $bucket_i$ to $bucket_{20}$. Table 4.2 shows the resulting number of length-two and length-twenty test cases in the pool (TP(2) and TP(20)).

Test suites. Each test suite is constructed by randomly selecting test cases from some fixed, randomly chosen bucket in the test pool until the test suite covers all GUI events that the test pool

covers. A test case is only added to the suite if it contains some event that the suite does not yet cover.

Faults. So far, it has been explained how the test suites in the $\langle \textit{test suite}, \textit{fault} \rangle$ pairs are generated. To obtain a set of faults, experimenters typically use one of three approaches: identifying actual faults inserted by the developers of the subject application, seeding faults by hand, or seeding faults programmatically. Each approach has its pros and cons, discussed more thoroughly elsewhere [1].

Because of the large number of faults needed for this study, the third approach, automatic seeding, is chosen. Fault seeding is accomplished using the tool MuJava². The oracle problem—classifying a test-case execution as “passed” or “failed”—is made tractable by creating multiple faulty versions of the application, each seeded with just one fault. If the output of a test case differs when it is run on a faulty version and on the “clean” version (in which no faults are seeded), then it is said that the test case detects the fault. For each subject application, all possible mutants are created (9458 for CWS and 53860 for FM); from these, mutants are randomly selected for the $\langle \textit{test suite}, \textit{fault} \rangle$ pairs. Thus, the numbers of mutants of different types found in the pairs are proportional to the numbers of opportunities for seeding those mutants, making the fault set biased in the sense that some mutant types are better represented than others. MuJava creates two kinds of mutants: “traditional” or method-level (e.g., inserting a decrement operator at a variable use) and class-level (e.g., changing the type of a data member).

4.3.2 Measurement of $\langle \textit{Test Suite}, \textit{Fault} \rangle$ Pairs

Several of the characteristics of $\langle \textit{test suite}, \textit{fault} \rangle$ pairs listed in Table 4.1 can be observed before any test cases are executed. This is true of all of the test-suite variables, which are straightforward to measure. Mut is easily obtained from the output of MuJava.

²<http://www.ise.gmu.edu/~ofut/mujava/>

Once the test pool has been executed on the clean version and on each faulty version of the subject application—a task requiring hundreds of hours of computation time, made feasible by running test cases on the distributed system Condor³—GUITAR’s output (an XML representation of the GUI state after each event is executed) from the clean and faulty application versions is compared for each test case to determine if it detected the fault. Since GUITAR itself contains some faults, the test results are manually examined to weed out many of GUITAR’s false reports of test-case failure. When it is known which faults each test case detects, *Det*, the fault-detection value for each $\langle \textit{test suite}, \textit{fault} \rangle$ pair (1 if the test suite detects the fault, 0 otherwise), is straightforward to compute.

This leaves two of the fault variables, *Branch* and *StmtDet*. *Branch* is found by noting in which method (if any) a fault occurs and analyzing the control-flow graph created by Sofya⁴ for that method. *StmtDet* is calculated using statement-coverage traces recorded by Instr⁵ as each test case is run on the clean version of the application. One hundred length-twenty test suites are constructed using a procedure similar to the one described in Section 4.3.1, except that the coverage criterion used here is “100%” statement coverage. The percentage is in quotes because ensuring that every executable statement is covered is a hard problem; each test suite is merely required to cover all statements that are exercised at least once by the test pool.

4.3.3 Logistic Regression Analysis

For statistical analysis in a study such as this, with a mixture of continuous and categorical independent variables and a boolean dependent variable, logistic regression is an obvious choice. Logistic regression models are a variation of linear regression models (e.g., best-fit lines) in which the *logit* of the dependent’s probability, rather than the probability itself, is expressed as a linear

³<http://www.cs.wisc.edu/condor/>

⁴<http://sofya.unl.edu>

⁵<http://www.glenmcc1.com/instr/index.htm>

function of the independents.

The logit function is

$$\text{logit}(x) = \log\left(\frac{x}{1-x}\right)$$

When x ranges from 0 to 1, as both **Det** and $Pr(\text{Det} = 1)$ do, $\text{logit}(x)$ ranges from $-\infty$ to ∞ . Let \vec{I} be the vector of independents:

$$\vec{I} = [\text{Len}, \text{Size}, \text{E3Cov}, \text{E2Cov}, \text{Mut}, \text{Branch}, \text{StmtDet}]$$

The logistic regression model of interest, then, is

$$\text{logit}(Pr(\text{Det} = 1)) = \alpha + \vec{\beta} \cdot \vec{I}$$

where α is a constant and $\vec{\beta}$ is the vector of coefficients for \vec{I} . Given a data set of \vec{I} and **Det** values, a maximum likelihood estimation algorithm finds values for α and $\vec{\beta}$. This is called *fitting* the model to the data. In this study, the R software environment⁶ is used to do model-fitting and other statistical analysis.

In a model that fits the data well, the coefficients $\vec{\beta}$ indicate the magnitude and direction of their respective independent variables' influence on the logit of the dependent variable. Usually, logistic regression coefficients are interpreted by transforming them into *odds ratios*. If an event occurs with probability p , the *odds* of its occurrence are

$$\text{odds}(p) = \frac{p}{1-p}$$

This is the ratio of the probability that the event occurs to the probability that it does not occur. It

⁶<http://www.r-project.org>

is also $\exp(\text{logit}(p))$. The odds ratio for \vec{I}_i ,

$$OR_i(\Delta) = \exp(\vec{\beta}_i \Delta)$$

is the factor by which $\text{odds}(Pr(\text{Det} = 1))$ increases when \vec{I}_i is increased by Δ and all other independents are held constant. (For dichotomous independents such as Branch, Δ must be 1.)

The significance of individual variables in the model can be assessed by using backwards stepwise regression and statistical tests. In this process, two models are fit to the data, one that includes the variable in question (the *full model*) and one that does not (the *reduced model*). Except for the variable in question, the two models must include the same set of variables, so that the reduced model is nested within the full model. A *likelihood ratio test* is a chi-square test of the difference between the two models' likelihood ratios, the error metric used in logistic regression. If the outcome of this test is *not* statistically significant, indicating that the two models are approximately equivalent, then the variable in question is superfluous in the full model and can be dropped from it.

4.4 Results

Table 4.3 gives an overview of the data collected for CWS, while Table 4.4 shows the same for FM. Although the two applications' data sets have many similarities, one notable difference is that FM's Size values are much larger.

CWS. Table 4.5 shows the fitted model when all independent variables but no interactions between variables are included. In this and other figures, variables significant at the 0.05 level are italicized. Recall from Section 4.3.3 that each coefficient estimates the log of the odds ratio of the independent variable's effects on the dependent. Because Mut has just two possible values, *class-level* or *method-level*—encoded as 0 and 1, respectively—its coefficient indicates how much

Table 4.3: CWS: Data summary.

Variable	Min.	Q1	Med.	Q3	Max.
StmtDet	0.0000	0.0000	0.0000	1.0000	1.0000
Branch	1	3	12	16	21
E3Cov	0.7619	1.1206	1.1922	1.2421	1.3480
E2Cov	0.6937	0.7956	0.8520	0.9491	2.6875
Size	16	253	330	391	551
Len	2	7	11.5	17	20
Variable	Values				
Mut	Class-level (0)		Method-level (1)		
	86		60		
Det	Undetected (0)		Detected (1)		
	88		58		

Table 4.4: FM: Data summary.

Variable	Min.	Q1	Med.	Q3	Max.
StmtDet	0.0000	0.0000	0.9800	1.0000	1.0000
Branch	1	4	11	19	26
E3Cov	0.3970	0.8643	0.9418	0.9916	1.0430
E2Cov	0.7736	0.8906	0.9065	0.9134	0.9260
Size	824	1468	1772	2130	2580
Len	2	6	10	15	20
Variable	Values				
Mut	Class-level (0)		Method-level (1)		
	63		83		
Det	Undetected (0)		Detected (1)		
	70		76		

more likely it is that $\text{Det} = 1$ for method-level, as opposed to class-level, mutants.

When variables are iteratively removed from the model (using likelihood ratio tests) and the reduced models are re-evaluated until only significant variables remain, the reduced model in Table 4.6 results. This model estimates that the odds of fault detection increase by $\exp(0.75095) = 2.1190$ when StmtDet increases by 10% and by $\exp(0.76295) = 2.1446$ when the E3Cov increases by 10%. Further, the odds of detecting a method-level mutant are $\exp(0.6475) = 1.9108$ times the odds of detecting a class-level mutant.

Table 4.7 shows the model that results when a model including all two-way interactions is fit

Table 4.5: CWS: All main effects.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-19.320275	11.197035	
Mut	0.845414	1.006531	0.043
StmtDet	8.605552	1.802348	9.276e-32
Branch	-0.005935	0.057449	0.688
E3Cov	10.124242	11.051126	0.006
E2Cov	1.575090	3.083643	0.451
Size	-0.009948	0.017290	0.696
Len	0.302461	0.250067	0.209

Table 4.6: CWS: Reduced main effects.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-13.8436	3.7367	
Mut	0.6475	0.9406	0.043
StmtDet	7.5095	1.2689	9.276e-32
E3Cov	7.6295	2.8486	0.005

to the data and non-significant terms are iteratively dropped, as just described. Again, a chi-square test of goodness of fit indicates that the model fits adequately.

FM. Table 4.8 shows the model built from all main effects, while Table 4.9 shows the model obtained by iteratively reducing the full model until only significant terms remain, as was done for CWS. The reduced model estimates that when **StmtDet** increases by 10% the odds of fault detection increase by $\exp(0.60941) = 1.8393$. Table 4.10 shows the reduced model created similarly from a model that includes all two-way interactions between variables. For all of these models, a chi-square test of goodness of fit shows that they fit the data adequately.

Table 4.7: CWS: Reduced interactions.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-13.482	3.944	
Mut	-4.457	4.651	0.043
StmtDet	6.202	1.198	9.276e-32
E3Cov	8.276	3.122	0.005
Mut:StmtDet	7.496	6.425	0.046

Table 4.8: FM: All main effects.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-51.632032	33.409756	
Mut	-1.871670	1.060126	0.283
StmtDet	8.286633	1.560292	$7.662e-34$
Branch	-0.087791	0.064158	0.069
E3Cov	-11.276352	15.762959	0.628
E2Cov	76.814515	48.580206	0.125
Size	-0.008781	0.007747	0.862
Len	0.613092	0.464744	0.170

Table 4.9: FM: Reduced main effects.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-3.1243	0.5858	
StmtDet	6.0941	0.8067	$3.46e-33$

4.5 Discussion

For both CWS and FM, one dominant factor affecting a fault’s probability of detection in this experiment is, perhaps not surprisingly, its probability of detection by statement-coverage-adequate test suites (StmtDet). This validates the common use of statement coverage in practice, but with a caveat: statement coverage in this study may be correlated with false reports of fault detection by GUITAR, as noted in the threats to validity below. In this study, all faults were mutations of single source lines and most occurred inside method bodies. Future work will investigate the hypothesis that statement coverage is less tightly coupled with fault detection for other kinds of faults.

For CWS, several additional factors are found to influence fault detection: mutant type (Mut), the ratio of event-triple coverage to event-pair coverage (E3Cov), and the interaction between

Table 4.10: FM: Reduced interactions.

Term	Coef.	Std. err.	<i>p</i>
Intercept	-3.1494	0.5939	
StmtDet	11.8639	3.5034	$3.46e-33$
StmtDet:Branch	-0.3023	0.1508	0.003

Mut and StmtDet. The results show that, while class-level mutants are more likely to be detected overall, the combination of being method-level and having a high StmtDet value increases a mutant’s odds of detection. According to the model in Table 4.7, increasing a test suite’s event-triple coverage by 10% while holding other variables constant increases the suite’s odds of detecting a given fault by a factor about 2.3, and a fault that is 10% more likely to be detected by a statement-coverage-adequate test suite has about 1.9 times greater odds of detection by GUI testing.

There are at least two alternate explanations for class-level mutants’ overall propensity for detection in CWS. One is that class-level mutants located outside of method bodies (e.g., *member variable initialization deletion*) may be exposed by executing any of several statements, unlike method-level mutants, which affect just one statement. Another is that, since coverage of mutants outside methods is not directly observable from statement-coverage traces, such coverage may be correlated with false reports of fault detection. In either case, the suspected influence of extra-method mutants accounts for the fact that Mut is significant for CWS, which has thirteen extra-method mutants, but not for FM, which only has three. The positive coefficient of $\text{Mut} \times \text{StmtDet}$ for CWS may compensate, in a sense, for the negative coefficient of Mut.

That E3Cov does not bear a relationship to fault detection in FM’s data suggests that the state spaces of FM and CWS differ in some important way. More of CWS’s events may interact semantically with each other (as defined by Yuan and Memon [37]), perhaps making it more likely that a particular sequence of events, rather than some single event, must be executed to expose a randomly placed fault.

The lack of relationship between test-case length (Len) and fault detection partly confirms earlier results by Xie and Memon [36] and Rothermel et al. [30], which show that test-case length/granularity affects which, but not how many, faults are detected. In the results of this work, test-case length is not found to affect fault detection, but neither are any qualities of faults in tandem with test-case length. In particular, the hypothesis that the level of branching in the method surrounding the fault is one of these qualities (inspired by the conjecture of Xie and Memon [36])

is not confirmed by the data, although threats to validity may have interfered. Apparently, if some variable or variables affect which faults are detected with different-length test cases, they remain to be found.

Threats to validity. Threats to internal validity concern factors other than the independent variables that may account for study results. Because the GUITAR framework used to generate and run test cases is an evolving research tool, it sometimes generates invalid test cases or fails to run valid ones. An attempt was made to minimize this threat by checking GUITAR’s results against statement coverage and discounting spurious failures (Section 4.3.1). This correction itself may have biased the results to some degree, since test cases with certain characteristics (e.g., covering a certain event) may be more likely to spuriously fail. Some GUITAR-reported failures that did not contradict statement-coverage traces may still have been spurious, so easily-covered faults may be correlated with false reports of fault detection.

Threats to external validity limit the generalizability of study results. Like any study whose sample of test suites and faulty applications is limited, this study does not enable one to predict with any confidence what would happen with other, different test suites and faulty applications. But, as Section 1 explained, this experiment is a starting point for studies of a much broader sample of testing techniques and software.

Threats to construct validity are discrepancies between the variables conceptually of interest and the variables actually measured in the study. The study variable `Branch` is the number of branch points in the *bytecode* of the faulty *method*, but it serves as a proxy for the number of branch points in the *source code* of the faulty *event handler* (Section 4.2). The latter value is hard to measure, since it is not clear how to map source-code lines to event handlers.

Threats to conclusion validity are problems with the way the study employs statistics. The two main threats here arise from assumptions of logistic regression analysis that may have been violated. First, all relevant variables are assumed to be included in the model. Of course, one goal of this research is to identify the variables that are relevant. Second, the data set of $\langle test\ suite,$

fault pairs must be chosen by independent sampling. Building the test suites from a test pool, though necessary to make the experiment feasible, may violate this assumption, though negligibly if the test pool is large enough [10].

4.6 Conclusions

This work explored the relationship between properties of a $\langle \textit{test suite}, \textit{fault} \rangle$ pair and the likelihood that the test suite detects the fault. It is anticipated that research on this topic will not only supplement the body of empirical software-testing studies, but also prove valuable to practitioners. Of the test-suite and fault properties studied in this work—mutant type, detectability with statement-coverage-adequate test suites, branch points, GUI-event-pair and event-triple coverage, test-suite size, and test-case length—a software tester could reasonably measure most with existing tools; in principle, measurement of all properties could be automated. Our experiments with two GUI-based applications support the hypothesis that certain of these properties can influence fault detection—specifically, detectability with statement-coverage-adequate test suites, event-triple coverage, mutant type, and the interaction between the first and third of these. When these results have been replicated in a broader set of testing situations, a software tester will be able to predict with some confidence how changes to a test suite would affect its ability to detect the kinds of faults most likely to be present. Further, understanding the similarities and differences between her context and the context in which some empirical study is performed, the software tester would better understand how to interpret the results of the study in her own context.

4.7 Lessons learned from preliminary work

The preliminary work presented a methodology for studying characteristics of faults and test suites in software testing and a set of fitted statistical models that resulted from applying the

methodology in an empirical study of two GUI-based applications. A preliminary set of test-suite and fault characteristics to use as parameters in the statistical models was developed. In the study, several of these model parameters turned out to be statistically significantly related to fault detection, indicating that the set of parameters chosen was a reasonable starting point for future studies and demonstrating the usefulness of the methodology. The preliminary work carried out the first iteration of Tasks 1–7 from Section 3.2 in order to investigate the first insight described in Section 3.1. The methodology developed in the work will serve as a model for the studies proposed to carry out the remaining goals.

While a promising start, the preliminary work had some weaknesses that will be addressed in the remaining work. Specifically, the threats to validity in Section 4.5 will be addressed. Threats to construct and conclusion validity arose from the fact that the choice of test-suite and fault characteristics to study was not optimal or complete. Although coming up with an ideal set of characteristics will require contributions from numerous researchers, if it is possible at all, revisiting Tasks 1 and 2 (Section 3.2) to investigate the relationship between test-suite and fault characteristics and coverage will improve upon the original set of characteristics. Another weakness of the preliminary work was that problems with the tools used to perform the study may have introduced noise into the data. Although the resulting threats to internal validity in the preliminary work were reduced by checking the results against statement-coverage data, Task 8 in the proposed work will further validate the results by manual inspection. Finally, the preliminary work suffered from threats to external validity because only a limited set of applications and a single kind of testing were studied. In the remaining work, Task 9 will mitigate these threats by exploring the applicability of the ideas and approaches proposed in this work to other kinds of applications and testing situations.

Chapter 5

Remaining Work

Table 5.1 gives the timeline for the work that remains, which was outlined in Section 3.2. Key parts of the timeline are scheduled around notification dates for submitted papers. At the end of May 2008, notification will be received for a paper submitted to the International Symposium on Foundations of Software Engineering (FSE), which addresses the proposed goal of studying coverage and fault detection separately. During the summer of 2008, notification is expected for a paper submitted to the Journal of Software Maintenance and Evolution (JSME) special issue on Search-Based Software Engineering, which presents the proposed framework for adaptive, search-based regression testing. These two submitted papers, when improved based on reviewers' comments, will complete Tasks 1–7. In particular, the fact that these papers revisit Tasks 1 and 2 will address the weaknesses of the preliminary work that arose from the choice of fault and test-suite characteristics to study, as described in Section 4.7. Since there is a one-month gap between the preliminary oral examination and the FSE notification, May 2008 will be spent on the first half of Task 8. This task will be completed later on, in January 2009, followed by Task 9 in February and dissertation writing in March and April. Tasks 8 and 9 will finish addressing the weaknesses of the preliminary work described in Section 4.7.

Table 5.1: Timeline for completion

Dates	Tasks
May 2008	Begin on Task 8. Specifically, create system for viewing results for individual faults, and use this to investigate some results from preliminary work.
Jun. 2008–Sep. 2008	Receive notification for FSE paper. Using reviewers’ comments, further develop ideas and experiments in the paper, either for camera-ready submission to FSE or for submission to International Conference on Software Engineering. This will partially address Tasks 1–7.
Sep. 2008–Dec. 2008	Receive notification for JSME paper. Using reviewers’ comments, further develop ideas and experiments in the paper, either for a revised submission to JSME or for submission to some other journal or conference. This will finish addressing Tasks 1–7.
Jan. 2009	Complete Task 8 by continuing to view results for individual faults and searching for new fault characteristics that may be important to coverage or fault detection.
Feb. 2009	Explore the applicability of the proposed work to domains other than GUI testing, addressing Task 9.
Mar. 2009–Apr. 2009	Write dissertation.

Bibliography

- [1] ANDREWS, J. H., BRIAND, L. C., AND LABICHE, Y. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering* (2005), pp. 402–411.
- [2] ANDREWS, J. H., BRIAND, L. C., LABICHE, Y., AND NAMIN, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.* 32, 8 (2006), 608–624.
- [3] BASILI, V. R. Software development: a paradigm for the future. In *Proceedings of the 13th international Computer Software and Applications Conference* (1989), pp. 471–485.
- [4] BASILI, V. R., AND SELBY, R. W. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng.* 13, 12 (1987), 1278–1296.
- [5] BRUNTINK, M., AND VAN DEURSEN, A. An empirical study into class testability. *J. Syst. Softw.* 79, 9 (2006), 1219–1232.
- [6] DINH-TRONG, T., GHOSH, S., FRANCE, R., BAUDRY, B., AND FLEURY, F. A taxonomy of faults for UML designs. In *Proceedings of the 2nd MoDeVa workshop - Model design and validation* (2005).
- [7] DO, H., AND ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Trans. Softw. Eng.* 32, 9 (2006), 733–752.

- [8] ELBAUM, S., GABLE, D., AND ROTHERMEL, G. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the 7th IEEE International Software Metrics Symposium* (Apr. 2001), pp. 169–179.
- [9] FERGUSON, R., AND KOREL, B. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* 5, 1 (1996), 63–86.
- [10] GARSON, G. D. Statnotes: Topics in multivariate analysis, 2006. <http://www2.chass.ncsu.edu/garson/PA765/statnote.htm>.
- [11] GRAVES, T. L., HARROLD, M. J., KIM, J.-M., PORTER, A., AND ROTHERMEL, G. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.* 10, 2 (2001), 184–208.
- [12] GUPTA, N., MATHUR, A. P., AND SOFFA, M. L. Automated test data generation using an iterative relaxation method. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (1998), pp. 231–244.
- [13] HARROLD, M. J., OFFUTT, A. J., AND TEWARY, K. An approach to fault modeling and fault seeding using the program dependence graph. *J. Syst. Softw.* 36, 3 (1997), 273–295.
- [14] HOWDEN, W. E. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.* SE-2, 3 (1976), 208–215.
- [15] HSIEH, F. Y., BLOCH, D. A., AND LARSEN, M. D. A simple method of sample size calculation for linear and logistic regression. *Statistics in Medicine* 17, 14 (July 1998), 1623–1634.
- [16] HUTCHINS, M., FOSTER, H., GORADIA, T., AND OSTRAND, T. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering* (1994), pp. 191–200.

- [17] KAI-YUAN CAI, YONG-CHAO LI, K. L. Optimal and adaptive testing for software reliability assessment. *Information and Software Technology* 46, 15 (2004), 989–1000.
- [18] LI, Z., HARMAN, M., AND HIERONS, R. M. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.* 33, 4 (2007), 225–237.
- [19] MCMMASTER, S., AND MEMON, A. M. Fault detection probability analysis for coverage-based test suite reduction. In *Proceedings of the 21st IEEE International Conference on Software Maintenance* (Paris, France, 2007), IEEE Computer Society.
- [20] MEMON, A. M., AND XIE, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.* 31, 10 (2005), 884–896.
- [21] MICHAEL, C. C., AND VOAS, J. Problems of accuracy in the prediction of software quality from directed tests. In *Proceedings of the International Conference on Testing Computer Software* (1997).
- [22] MORELL, L. J. A theory of fault-based testing. *IEEE Trans. Softw. Eng.* 16, 8 (1990), 844–857.
- [23] MORGAN, J. A., KNAFL, G. J., AND WONG, W. E. Predicting fault detection effectiveness. In *Proceedings of the 4th IEEE International Software Metrics Symposium* (1997), p. 82.
- [24] MUNSON, J. C., AND NIKORA, A. P. Toward a quantifiable definition of software faults. In *ISSRE '02: Proceedings of the 13th International Symposium on Software Reliability Engineering* (Washington, DC, USA, 2002), IEEE Computer Society, pp. 388–395.
- [25] MUSA, J. D. Operational profiles in software-reliability engineering. *IEEE Softw.* 10, 2 (1993), 14–32.
- [26] OFFUTT, A. J., AND HAYES, J. H. A semantic model of program faults. In *Proceedings of the International Symposium on Software Testing and Analysis* (1996), pp. 195–200.

- [27] OFFUTT, A. J., LEE, A., ROTHERMEL, G., UNTCH, R. H., AND ZAPF, C. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118.
- [28] OSTRAND, T. J., WEYUKER, E. J., AND BELL, R. M. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.* 31, 4 (2005), 340–355.
- [29] RICHARDSON, D. J., AND THOMPSON, M. C. An analysis of test data selection criteria using the RELAY model of fault detection. *IEEE Trans. Softw. Eng.* 19, 6 (1993), 533–553.
- [30] ROTHERMEL, G., ELBAUM, S., MALISHEVSKY, A. G., KALLAKURI, P., AND QIU, X. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.* 13, 3 (2004), 277–331.
- [31] STRECKER, J., AND MEMON, A. M. Faults’ context matters. In *Proceedings of 4th International Workshop on Software Quality Assurance* (Sept. 2007).
- [32] STRECKER, J., AND MEMON, A. M. Relationships between test suites, faults, and fault detection in GUI testing. In *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation* (Apr. 2008).
- [33] VOAS, J. M. Factors that affect software testability. Tech. Rep. NASA-91-9pnsqc-jmv, NASA Langley, 1991.
- [34] VOAS, J. M. Pie: A dynamic failure-based technique. *IEEE Trans. Softw. Eng.* 18, 8 (1992), 717–727.
- [35] WEYUKER, E. J. Can we measure software testing effectiveness? In *Proceedings of the 1st International Software Metrics Symposium* (1993), pp. 100–107.

- [36] XIE, Q., AND MEMON, A. Studying the characteristics of a "good" GUI test suite. In *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering* (Nov. 2006).
- [37] YUAN, X., AND MEMON, A. M. Using GUI run-time state as feedback to generate test cases. In *Proceedings of the 29th International Conference on Software Engineering* (2007).