Relationships Between Test Suites, Faults, and Fault Detection in GUI Testing

Jaymie Strecker
University of Maryland
College Park, MD, USA
strecker@cs.umd.edu

Atif M Memon
University of Maryland
College Park, MD, USA
atif@cs.umd.edu

Abstract

Software-testing researchers have long sought recipes for test suites that detect faults well. In the literature, empirical studies of testing techniques abound, yet the ideal technique for detecting the desired kinds of faults in a given situation often remains unclear. This work 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. We present a methodology for discovering which factors do statistically affect fault detection, and we perform an experiment with 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. Introduction

A fundamental question in software testing is how to build a good test suite. It is a complicated question, since the term "good" unpacks into so many competing qualities: quick to create, quick to execute, easy to maintain, effective at detecting faults, and many more. From the results of myriad empirical studies, we have accrued a wealth of data points (e.g., in V seconds, technique W detects on average X faults in version Y of application Z). However, the data points come from many different test suites, faults, and applications, and differences among the study subjects may have affected the results in ways we do not yet understand. This makes us hesitate to predict from the existing data how well a technique will perform in a new situation—for example, on a new application.

To make the problem of building good test suites even more complex, a data point may mean different things qualitatively to people in different contexts. Some research has addressed this by employing metrics that account for such qualitative factors as fault severity [4]. But because fault severity is a subjective measure, results in these terms cannot reliably be generalized to a wider range of contexts. On the other hand, the common research practice of assuming that all faults are equally severe and finding how many of a "representative" sample of faults a testing technique detects (e.g., [6, 14, 16, 18]) also ties the results to a specific context, since it is not clear what a truly representative fault set looks like or what class of contexts it would represent [1, 7, 13, 17].

Thus, it is unclear how data points from software-testing studies relate quantitatively to each other, and it is unpredictable how they relate qualitatively to real situations. This work addresses both issues. We show 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 environment in which an evaluation is performed with respect to those influences. A researcher or practitioner in a different environment—with a different test subject or different assumptions about fault severity, for example—could then better predict, based on the differences between the two environments, how the evaluated technique would perform in her case.

Some past work has started in this direction by studying how certain characteristics of the application under test and its coverage during testing influence fault detection [3, 12]. Overall, however, the relationship between application and test-suite characteristics has been sparsely explored. Particularly obscure is the way characteristics of the faults within an application and characteristics of a test suite interact to affect fault detection. With few exceptions (e.g., [2, 7]), empirical studies of testing techniques typically report numbers of faults detected and faults missed but do not characterize those faults. Yet testers could benefit from such an analysis.

The following example, based on a study by Basili and Selby [2], shows one way that testers could benefit. An empirical study finds that techniques T_1 and T_2 each detect, on average, 54%-55% of a sample of faults. A break-down of

the results by fault class shows that T_1 detects fewer faults of class C_1 than T_2 does (42.8% vs. 66.7%), but more faults of class C_2 (46.7% vs. 30.7%). Suppose that a tester wants to know which of T_1 and T_2 would likely reveal more faults in a certain application. In earlier versions, faults of class C_1 caused twice as many user-reported failures as faults of class C_2. Considering these facts, the tester opts for T_2. If the tester had not understood how her environment differed from the study's, she could easily have missed out on the advantages of T_2.

Of course, this example makes some assumptions. For the tester to truly benefit from the break-down of results by fault class, the fault classification must be automatable (unlike the classification on which the example is based [2]), and the relationship between fault classes C_1 and C_2 and test suites produced by techniques T_1 and T_2 must be known to persist from the study's context to the tester's context. More generally, 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, we identify a set of characteristics of faults and test suites that we expect may affect fault detection: 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. We emphasize 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, we statistically analyze the relationship between these characteristics and fault detection 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 our experimental infrastructure can sit atop the existing GUI Testing Framework (GUITAR) [18], our approach 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.

The next section catalogues prior work relevant to this study. In Section 3, we build upon prior work in identifying the set of test-suite and fault characteristics to investigate experimentally. Section 4 describes the experiment design. Section 5 presents the results, while Section 6 discusses their implications. In Section 7, we state our conclusions and point to opportunities for future research.

2. Related work

When applying a testing technique to a piece of software, a tester may be surprised to find that the test suite produced performs better or worse than she had expected, based on experience with the technique in other contexts. This section discusses variables that have been shown or conjectured to influence a test suite's effectiveness and efficiency at detecting faults.

2.1. Test-suite variables

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 [2]; data-flow- and control-flow-based techniques [9]; regression test selection techniques [6]; variations of mutation testing [14]; and strong and weak test oracles for GUIs [11].

Even when produced by the same testing technique, test suites can differ in important ways. Rothermel et al. [16] 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 [18] 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 *shallow faults*, those that can be detected by shorter test cases, lie in event handlers with less complex branching than *deep faults*, those that can only be detected by longer test cases.

In another study of GUI testing, McMaster and Memon [10] show that the coverage criterion used in test-suite reduction affects the size of the reduced test suites and the number of faults they detect. The effect on fault-detection effectiveness may be indirect, however, because the criteria that produce more effective test suites also produce larger test suites. The study shows that test suites reduced using call-stack coverage detect many more faults than randomly reduced suites of the same size. Whether the same is true for the other criteria studied (statement, method, event, and event-pair coverage) for GUI-based applications remains to be studied.

A study by Elbaum et al. [3] 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.

2.2. Application variables

In the study by Elbaum et al. [3] 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. [12] 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. [15] 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.

2.3. Fault variables

Section 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 [2] 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 faults. In some cases, like the example given in Section 1, 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. [7] classify detected faults using a different taxonomy. In it, classes of faults are distinguished 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.

Offutt and Hayes [13] 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] and Rothermel et al. [16] to characterize faults' ease of detection with respect to the test pools used in the studies. This study leaves out semantic size because it is confounded with test-suite size: the greater a fault's semantic size, the more likely that smaller suites detect it.

3. Variables of interest

As Section 1 noted, this work focuses on GUI testing so that we can take advantage of existing infrastructure for performing empirical studies. In GUI-based applications, a user sends input by performing GUI *events* (e.g., clicking on a button, typing in a text box) and receives output in the form of changes to the visible GUI windows and widgets, whose properties together make up the GUI state. The portion of the application code that executes in response to a GUI event is called the *event handler*. As in prior work [18], we define a test case to be a sequence of GUI events and its output to be the sequence of GUI states that the application passes through. The GUI state is checked after each GUI

event. A test case detects a fault if the actual state is not as expected at any of those checks. GUI test cases are actually system tests, as they exercise the whole software, including non-GUI code.

As research into factors affecting fault detection matures, we hope that a paradigm for selecting characteristics to study will emerge. But for now, while our selection is rooted in the literature, it must remain somewhat ad hoc. To make our work more replicatable, we chose characteristics that could be measured objectively, automatically, and without special artifacts such as specifications [17].

Section 2.1 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 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 allows us 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. We could have looked at coverage of length-four and longer event sequences but chose not to until we had 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.

We also investigate the influence of fault variables on a fault's chances of detection. Table 1 summarizes the fault variables studied. As Section 2.1 mentioned, it has been speculated that the number of branch points in an event handler (Branch) together with test-case length is related to the probability that a fault in the event handler is detected. In prior work [17], we have asserted that an effective way to characterize the faults detected by a testing technique is in terms of their detection by other techniques. Hence, we consider 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 we classify each fault accordingly (Mut).

Table 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

4. Study design

We want to know which of the variables listed in Table 1 affect the likelihood that fault detection occurs in a ⟨*test suite, fault*⟩ 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 us to conduct our study by collecting a random sample of ⟨*test suite, fault*⟩ pairs; measuring the values of the variables in Table 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. We now explain each of these steps in turn.

4.1. The Sample of ⟨*Test Suite, Fault*⟩ Pairs

Subject applications. Two open-source applications from SourceForge[1] 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 [19] make them attractive candidates for this study. Table 2 gives each application's size in non-commented, non-blank lines of code (LOC) and in classes (Cls.).

GUITAR. The ⟨*test suite, fault*⟩ pairs in this study are built from test cases that execute the subject applications and faults that are embedded inside the subject applications. Tools in the GUI Testing Framework (GUITAR) [18] automate the creation and execution of test cases and the comparison of test outputs (i.e., the oracle procedure). GUITAR

[1] `http://sourceforge.net`

enables us to produce much larger data sets than would be feasible using other, more labor-intensive approaches such as capture-replay tools. Since GUITAR itself contains some faults, we manually examine the test results to weed out many of GUITAR's false reports of test-case failure.

Sample size. The sample of ⟨*test suite, fault*⟩ 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). Typically, a researcher fixes the effect size at the smallest value of practical significance. Having no precedent in the software-testing literature, we select an effect size of 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) [5].

The necessary sample size is estimated by applying the procedure outlined by Hsieh et al. [8] to data from a pilot study. In the pilot study, a sample of 100 ⟨*test suite, fault*⟩ pairs is analyzed for 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 in Appendix A.

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 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 we anticipated that the sufficient test-pool size for some applications would exceed the number of possible length-one test cases—yet we wanted 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: we need only 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 ith event (as determined by examining statement coverage of lines with seeded faults) were removed from $bucket_i$ to $bucket_{20}$. Table 2 shows the resulting number of length-two and length-twenty test cases in the pool (TP(2) and TP(20)).

Table 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

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, we have explained how the test suites in the ⟨*test suite, fault*⟩ 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, we opt for the third approach, automatic seeding, using MuJava[2]. 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"

[2] `http://www.ise.gmu.edu/~ofut/mujava/`

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 ith (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 ← {all events in application}
2: i ← 0
3: while i <iters do
4:    for tc ∈ bucket_1 do
5:       tc' ← tc
6:       for i from 2 to 20 do
7:          uncov ← succs(last(tc')) − covSuccs(tc')
8:          if uncov = ∅ then
9:             covSuccs(tc') ← ∅
10:            uncov ← succs(last(tc'))
11:         end if
12:         e ← random event in uncov
13:         covSuccs(tc') ← covSuccs(tc') ∪ e
14:         tc' ← tc' ∘ e
15:         bucket_i ← bucket_i ∪ tc'
16:      end for
17:   end for
18:   i ← i + 1
19: end while
```

version (in which no faults are seeded), then we say 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 ⟨*test suite, fault*⟩ 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).

4.2. Measurement of ⟨*Test Suite, Fault*⟩ Pairs

Several of the characteristics of ⟨*test suite, fault*⟩ pairs listed in Table 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.

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 Condor[3]—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. When it is known which faults each test case detects, Det, the fault-detection value for each ⟨*test suite, fault*⟩ 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 Sofya[4] for that method. StmtDet is calculated using statement-coverage traces recorded by Instr[5] 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 4.1, 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; we merely require that each test suite cover all statements that are exercised at least once by the test pool.

4.3. 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 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 environment[6] 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

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

[3] http://www.cs.wisc.edu/condor/

[4] http://sofya.unl.edu

[5] http://www.glenmccl.com/instr/index.htm

[6] http://www.r-project.org

This is the ratio of the probability that the event occurs to the probability that it does not occur. It 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 $odds(Pr(\text{Det} = 1))$ 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.)

To evaluate how well the computed logistic regression models fit our data, we perform a *chi-square test of goodness of fit*. This statistical test determines whether the magnitude of error is acceptably low. By *error*, we mean the discrepancy between the predicted values of $Pr(\text{Det} = 1)$, as computed by plugging values of $\vec{I}$ from the data into the model, and the actual values of Det in the data. For logistic regression models, the error metric used is called the *likelihood ratio*. The likelihood ratio "reflects the significance of the unexplained variance in the dependent" [5] and is computed by the R application during model-fitting. If the model fits well, then the distribution of the likelihood ratio approximates a chi-square distribution. The chi-square test of goodness of fit assesses how much the likelihood ratio deviates from the expected chi-square distribution. If the deviation is small enough, implying that the model fits well, then the chi-square test tells us as much: it is *not* statistically significant.

The significance of individual variables in the model can be assessed in a similar way. In this case, 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. 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.

5. Results

Table 3 gives an overview of the data collected for CWS, while Table 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 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 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 more likely it is that Det $= 1$ for method-level, as opposed to class-level, mutants.

Table 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. 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

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 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 statement-level mutant. For both the full and the reduced model, the chi-square test of goodness of fit gives a non-significant result, indicating that the models fit the data well.

Table 7 shows the model that results when a model including all two-way interactions is fit 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 8 shows the model built from all main effects, while Table 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

Table 5. CWS: All main effects.

Term	Coef.	Std. err.	p
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 6. CWS: Reduced main effects.

Term	Coef.	Std. err.	p
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*

increase by $\exp(0.60941) = 1.8393$. Table 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.

6. 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 our 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 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 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.

Table 7. CWS: Reduced interactions.

Term	Coef.	Std. err.	p
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 8. FM: All main effects.

Term	Coef.	Std. err.	p
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 9. FM: Reduced main effects.

Term	Coef.	Std. err.	p
Intercept	-3.1243	0.5858	
StmtDet	6.0941	0.8067	*3.46e-33*

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 Mut $\times$ 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 [19]) 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.

Table 10. FM: Reduced interactions.

Term	Coef.	Std. err.	p
Intercept	-3.1494	0.5939	
StmtDet	11.8639	3.5034	*3.46e-33*
StmtDet:Branch	-0.3023	0.1508	*0.003*

The lack of relationship between test-case length (Len) and fault detection partly confirms earlier results by Xie and Memon [18] and Rothermel et al. [16], which show that test-case length/granularity affects which, but not how many, faults are detected. In our results, 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, our 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 [18]) 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. We tried to minimize this threat by checking GUITAR's results against statement coverage and discounting spurious failures (Section 4.1). 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 us 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 3). 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 ⟨*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 [5].

7. Conclusions and future work

This work explored the relationship between properties of a ⟨*test suite, fault*⟩ pair and the likelihood that the test suite detects the fault. We think 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.

Much future work remains. This includes addressing threats to validity encountered in this work, improving the statistical model relating characteristics of testing situations to fault detection, and providing techniques and tools to practitioners.

Many of the threats to validity in this work can be mitigated by performing similar experiments with different applications, different kinds of faults, and different testing techniques. If statistical models constructed from different data sets vary widely, then additional model parameters may need to be sought to explain the variation. In fact, we do not expect the set of model parameters used here to be complete; they were just a starting point that allowed us to demonstrate a general methodology for evaluating model parameters. In the future, model parameters for classes of software other than GUI-based, including the closely related class of event-driven software, will need to be selected and evaluated.

8. Acknowledgments

This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421. The authors thank Jeff Offutt for his help with using MuJava.

A. Calculation of Sample Size

The following formulas are due to Hsieh et al. [8] and were implemented by this paper's authors 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 uth 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,

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

$$\begin{aligned} V &= \frac{r(1-r)}{s_1} \\ W &= r_0(1-r_0) + \frac{r_1(1-r_1)(1-s_1)}{s_1} \end{aligned}$$

In these formulas, s_1 is the proportion of the pilot-study data with $X = 1$; r_0 and r_1 are the empirical probabilities of fault detection when $X = 0$ and $X = 1$, respectively; and $r = (1 - s_1)r_0 + s_1 r_1$ 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 our study, we find n for each independent variable and take the maximum of these.

References

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of ICSE '05*, pages 402–411, 2005.

[2] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng.*, 13(12):1278–1296, 1987.

[3] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of METRICS '01*, pages 169–179, Apr. 2001.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of ICSE '01*, pages 329–338, 2001.

[5] G. D. Garson. Statnotes: Topics in multivariate analysis, 2006. `http://www2.chass.ncsu.edu/garson/PA765/statnote.htm`.

[6] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[7] M. J. Harrold, A. J. Offutt, and K. Tewary. An approach to fault modeling and fault seeding using the program dependence graph. *J. Syst. Softw.*, 36(3):273–295, 1997.

[8] F. Y. Hsieh, D. A. Bloch, and M. D. Larsen. A simple method of sample size calculation for linear and logistic regression. *Statistics in Medicine*, 17(14):1623–1634, July 1998.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of ICSE '94*, pages 191–200, 1994.

[10] S. McMaster and A. Memon. Call stack coverage for GUI test-suite reduction. In *Proceedings of ISSRE '06*, pages 33–44, 2006.

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

[12] J. A. Morgan, G. J. Knafl, and W. E. Wong. Predicting fault detection effectiveness. In *Proceedings of METRICS '97*, page 82, 1997.

[13] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *Proceedings of ISSTA '96*, pages 195–200, 1996.

[14] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[15] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005.

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

[17] J. Strecker and A. M. Memon. Faults' context matters. In *Proceedings of SOQUA '07; co-located with ESEC/FSE '07*, Sept. 2007.

[18] Q. Xie and A. Memon. Studying the characteristics of a "good" GUI test suite. In *Proceedings of ISSRE 2006*, Nov. 2006.

[19] X. Yuan and A. M. Memon. Using GUI run-time state as feedback to generate test cases. In *Proceedings of ICSE '07*, pages 396–405, May 2007.