# Fault Detection Probability Analysis for Coverage-Based Test Suite Reduction

Scott McMaster and Atif Memon
*University of Maryland*
*College Park, MD 20742, USA*
*{scottmcm,atif}@cs.umd.edu*

## Abstract

*Test suite reduction seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness. Most approaches to this problem are based on eliminating test cases that are redundant relative to some coverage criterion. The effectiveness of applying various coverage criteria in test suite reduction is traditionally based on empirical comparison of two metrics derived from the full and reduced test suites and information about a set of known faults: (1) percentage size reduction and (2) percentage fault detection reduction, neither of which quantitatively takes test coverage data into account. Consequently, no existing measure expresses the likelihood of various coverage criteria to force coverage-based reduction to retain test cases that expose specific faults. In this paper, we develop and empirically evaluate, using a number of different coverage criteria, a new metric based on the "average expected probability of finding a fault" in a reduced test suite. Our results indicate that the average probability of detecting each fault shows promise for identifying coverage criteria that work well for test suite reduction.*

*Keywords* – Test coverage, Test suite management, reduction, minimization

*Index Terms* – Testing strategies, Test coverage of code, Test Management, Testing Tools

## 1. Introduction

Software developers are increasingly using development approaches in which new code is incorporated into the baseline at a rapid pace and built via a nightly build or continuous integration process [1]. In such an environment, it is critical to perform *regression testing* after each build to ensure that newly incorporated changes have not introduced defects in previously functional parts of the system. However, the time to re-execute the test suite must not be allowed to excessively limit the speed of the integration cycle. Thus, in practice, it may be necessary to limit the size of the regression test suite.

*Test suite reduction* [4][16][11][19] (also referred to as *test suite minimization* in the literature) seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness. Most approaches to this problem are based on eliminating test cases that are redundant relative to some coverage criterion, such as program-flow graph edges [16], dataflow [19], dynamic program invariants [3], or call stacks [10][11]. The effectiveness of applying various coverage criteria in test suite reduction is traditionally based on empirical comparison of two metrics derived from the full and reduced test suites and information about a set of known faults. The two metrics are percentage size reduction:

$$(1) \quad 100 * (1 - \text{Size}_{\text{Reduced}} / \text{Size}_{\text{Full}})$$

And percentage fault detection reduction:

$$(2) \quad 100 * (1 - \text{FaultsDetected}_{\text{Reduced}} / \text{FaultsDetected}_{\text{Full}})$$

In typical test suite reduction experiments such as those found in [16], large numbers of full test suites are reduced using an algorithm such as [4]. Two test suites are generally considered to be equally effective at detecting a specific fault if they each contain at least one test case that exposes the fault.

Neither the percentage size reduction metric nor the percentage fault detection reduction metric quantitatively takes test coverage data into account. Consequently, no existing measure expresses the likelihood of various coverage criteria to force coverage-based reduction to retain test cases that expose specific faults. This leads us to propose and evaluate a new metric based on the *average expected probability of finding a fault in a reduced test suite*. In this paper, we present a formal definition of this metric

and use it to evaluate test suite reduction based on a number of different coverage criteria.

The remainder of this paper is structured as follows. In the next section, we present definitions and propose the new metric, along with an algorithm for calculating it in test suite reduction experiments. Section 3 briefly describes an earlier set of test suite reduction experiments that serve as the basis for this work. In Section 5, we apply the new metric to a set of empirical studies. We present related work in Section 6 and make concluding remarks in Section 7.

## 2. Fault Detection Probability Metric

In this section, we define a number of functions on the coverage and fault data collected from an application, its test pool, a set of known faults, and a coverage criterion. These definitions will lead us to a new metric for coverage-based test suite reduction utilizing the *average probability of detecting each fault*. Intuitively, this metric captures the likelihood that coverage-adequate reduced test suites will detect the same faults as their original counterparts, taking into account the number of coverage requirements which only appear in fault-detecting test cases. As we will show, this quantity varies greatly depending on the selected coverage criterion, thus making it useful in selecting the best criterion to use in a test suite reduction technique.

### 2.1. Data Structures

Given a subject application, a set of test cases *TC(1..J)*, a set of known faults *KF(1..K)*, and a set of coverage requirements *CR(1..I)*, it is possible to obtain two artifacts important to the study of test suite reduction, as well as the closely related topics of test case prioritization [17] and regression test selection. The first is the *coverage matrix, C*, [2] for a test suite. In a coverage matrix, each row represents a coverage requirement, such as a line, edge or call stack, and each column represents a test case. A cell value *C(i, j)* is 1 if coverage requirement *i* is satisfied by test case *j* and 0 otherwise. Based on the coverage matrix, we define a function *covReqTCs(C, i)*, which, given a coverage matrix *C* and a coverage requirement *i*, returns the set of test cases which satisfy the given requirement.

$$(3)\ \text{covReqTCs}(C,i) = \{j \in TC \mid C(i,j) = 1\}$$

Second, we consider the fault matrix, F, where each row represents a known fault and each column is a test case. A cell value *F(k, j)* is 1 if fault *k* is detected by test case *j* and 0 otherwise. This leads to another function, *detectsFaultTCs(F, k)*, which accepts a fault matrix *F* and fault number *k* and returns the set of test cases that detect *k*.

$$(4)\ \text{faultDetectingTCs}(F,k) = \{j \in TC \mid F(k,j) = 1\}$$

For a given test suite, the matrices *C* and *F* have the same column rank which is the number of test cases.

### 2.2. Metric Definition

Making use of the coverage matrix *C* and fault matrix *F*, we seek to define a metric that captures the average expected probably of finding a fault after coverage-based test suite reduction. This metric will be independent of the selection of a specific coverage-preserving reduction algorithm. From *C* and *F*, we define the *fault correlation* for a coverage requirement *i* to a fault *k* as the ratio of test cases in the test suite that satisfy the coverage requirement *and* detect the fault to the number of test cases that merely satisfy the coverage requirement:

$$(5)\ \text{faultCorr}(C,F,i,k) = \text{Card}[\ \text{covReqTCs}(C,i) \cap \text{faultDetectingTCs}(F,k)\ ] / \text{Card}[\ \text{covReqTCs}(C,i)\ ]$$

where Card[] returns the cardinality of a set. If a coverage requirement *i* is only satisfied only by test cases that detect a given fault *f*, then faultCorr(C,F,i,j) = 1, the maximum possible fault correlation. Intuitively, any coverage-preserving test suite reduction technique *must* select a fault-detecting test case for that fault.

The expected probability of finding a given fault after test suite reduction is defined as the maximum fault correlation of all coverage requirements with that fault:

$$(6)\ \text{expProbFindFault}(C,F,k) = \text{Max}(\ \text{faultCorr}(C,F,i,k)\ \forall\ i \in CR\ )$$

For example, if a coverage requirement satisfied by two test cases, one of which detects a given fault and one of which does not, the fault correlation is 0.5. If no coverage requirement leads to a higher fault correlation, then a coverage-preserving test suite reduction technique (considered independently of other coverage requirements) would select a fault-detecting test case with probability of 0.5. Thus, the expected probability of finding all faults after test suite reduction is the product of the expected probability of detecting each fault:

(7) expProbFindAll($C$,$F$) =

$$\Pi \text{ (expProbFindFault}(C,F,k) \ \forall \ k \in KF)$$

Because our goal is to compare how various coverage criteria perform in test suite reduction, we desire a metric which is normalized across subject applications and test suites with differing numbers of coverage requirements and detectable faults. Thus, we consider the average expected probably of detecting each fault:

(8) avgExpProbFindEach($C$,$F$) =

$$\text{Avg}( \text{ expProbFindFault}(C,F,k) \ \forall \ k \in KF \ )$$

Figure 1 presents an algorithm for calculating (8) for a given subject application, fault matrix, and coverage matrix.

```
ALGORITHM:  CalcFaultDetectionProbability (
1  C(1..I, 1..J), /* coverage matrix, I=number of
       coverage requirements,  J=number of test cases*/
2  F(1..K, 1..J) /* fault matrix, K=number of known
                    faults, J=number of test cases */
3    Declare P(1..K) /* expected probabilities of
                        finding faults 1..K */
4    for k = 1..K { /* for each fault */
5      P(k) = 0
6      for i = 1..I { /* for each coverage
                         requirement */
7        countCoveringCases <- 0
8        countCoveringDetectingCases <- 0
9        for j = 1..J { /* for each test case */
10         if C(i, j) = 1 then {
11           countCoveringCases <-
                          countCoveringCases + 1
12         if F(k, j) = 1 then {
13           countCoveringDetectingCases <-
                 countCoveringDetectingCases + 1
14         }
15       } /* j */
16       if countCoveringCases = 0 then next i
17       faultCorrelation =
                 countCoveringDetectingCases /
                          countCoveringCases
18       P(k) = Max(faultCorrelation, P(k))
19     } /* i */
20   } /* k */
21   Return Sum(P(1..K)) / K
```

**Figure 1: CalcFaultDetectionProbability Algorithm**

The `CalcFaultDetectionProbability` algorithm assumes the coverage matrix and fault matrix as inputs (Lines 1 and 2). It then declares an array with length equal to the number of known faults to hold the calculated probabilities (Line 3). Then, for each coverage requirement for each fault, counters are initialized to hold the number of test cases that cover the requirement, and both cover the requirement and detect the fault (Lines 4..8). The coverage matrix and fault matrix are referenced for each test case to increment the counters (Lines 9..15). We allow for the case where no test cases hit the coverage requirement, in which case we move forward to the next one (Line 16). The counters are then used to calculate the fault correlation number (Line 17), and the maximum probability of detecting the fault is potentially updated (Line 18). After all coverage requirements and faults are evaluated, the average probability of detecting each fault is calculated (Line 21).

## 3. Experiments

To assess the effectiveness of the average expected probability of detecting each fault as a metric for test suite reduction, we applied it to a set of test suite reduction experiments [10]. A brief summary of these experiments appears next. For additional details, see [10]. The experimental and analytical process appears in Figure 2, where lines indicate inputs into process steps and calculations.

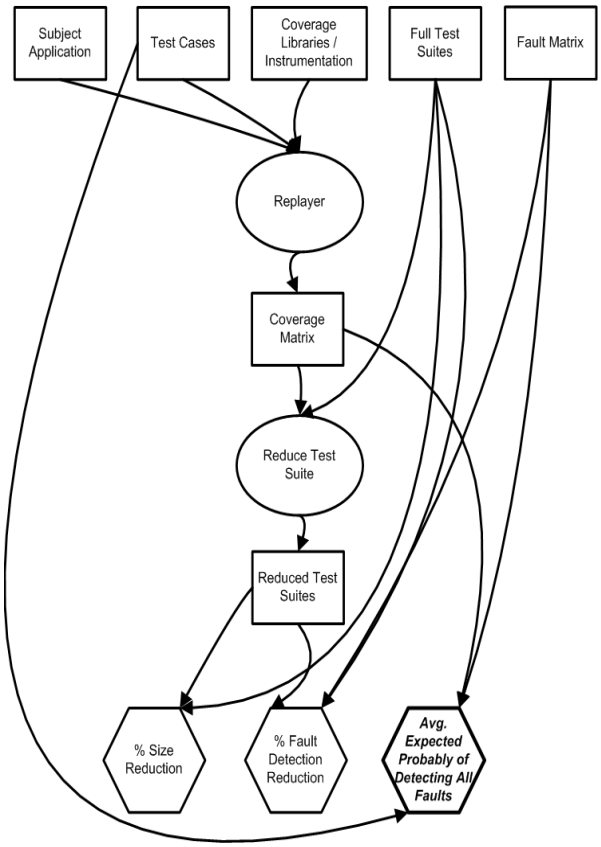


**Figure 2: Experimental and Analytical Process**

### 3.1. Subject Applications

In our study of test suite reduction for GUI applications, we used three subject applications from the TerpOffice Suite [15], TerpPaint (TP), TerpSpreadsheet (TS), and TerpWord (TW). Written

in Java and making heavy use of the Swing GUI framework, these applications were developed by advanced undergraduate software engineering students over a period of years. Each application comes with a large test universe generated using the event flow criterion [13] and a set of versions each containing a single mutation fault. Characteristics of these applications appear in Table 1.

| Application | TerpPaint (TP) | TerpWord (TW) | TerpSpreadsheet (TS) |
|---|---|---|---|
| **Test Universe Size** | 1500 | 1000 | 1000 |
| **# Detectable Faults** | 43 | 18 | 101 |
| **# Call Stacks Observed** | 413166 | 569933 | 333882 |
| **# Methods Observed** | 12277 | 12665 | 11103 |
| **# Events** | 181 | 219 | 110 |
| **# Lines** | 11803 | 9917 | 5381 |

**Table 1: Subject Applications**

## 3.2. Coverage Criteria

We collected five different types of coverage data for the entire test universe for each subject application; event (E1), event-interaction (E2), line (L), method (M), and call-stack (CS).

*Event-based coverage* [14] has been developed specifically for applications where test cases can be defined as sequences of events, and as such it is particularly suited to GUI applications. Examples of events in GUI applications include button clicks, menu selections, and keystrokes. Using a tool called GUI Ripper [12] a model of a GUI can be automatically derived, and from this model, test cases with varying event sequence lengths can be automatically generated. Another tool, the JavaGUIReplayer [15] can subsequently be used to execute the test cases. In this work, we consider two different event sequence lengths represented in the reduction techniques E1 and E2. In E1, each event in isolation is a coverage requirement to be covered by any reduced test suite, and in E2, coverage requirements are made up of pairs of events.

In *line coverage*, the coverage of each source code line induced by test execution against a given subject application is measured. From this, we define reduction technique L, in which reduced test suites must obtain the same line coverage as their full counterparts. In these experiments, line coverage data was obtained using the jcoverage tool [7]. For feasibility, this technique does not include coverage of the supporting Java libraries, but rather only includes coverage of the TerpOffice application source.

*Call-stack coverage* measures the coverage of each runtime call stack observed during execution of a test case. A call stack is an ordered sequence of active method calls in a running application. The call stack criterion has been shows to be particularly effective as a test suite reduction criterion against GUI subject applications [10]. Call stack coverage data was collected using the JavaCCTAgent tool [6] and used in the CS test suite reduction technique. The CS technique does include coverage of methods in the underlying Java libraries.

*Method coverage* is used to reduce test suites in the M technique. In M, each method appearing in the full test suite must also appear in the corresponding reduced suite. This information is derivable from call-stack coverage data and does incorporate coverage of Java libraries.

## 3.3. Reduction Technique

A large number of test suites of varying sizes were randomly generated for each subject application out of their respective test universes. Using the coverage data, each test suite was reduced using the ReduceTestSuite algorithm of Harrold *et al*. [4]. In these experiments, each randomly generated test suite was reduced based on each of the five coverage criteria E1, E2, M, L, and CS.

## 3.4. Prior Metrics and Results

In [10], the reduced test suites were evaluated based on their percentage size reduction and percentage fault detection reduction as defined formulas (1) and (2) from Section 1. We summarize those results in the following sections.

**3.4.1. Size Reduction.** Size reduction behavior was found to be similar across all three subject applications. For the largest original test suite sizes, coverage-preserving reduction using E1, L, and M resulted in size reduction in the 75-91% range. At the other end, reduction with E2 as the criterion yielded very little size reduction at all – at most 11%. Call-stack-based reduction struck a middle ground, providing size reduction in the 38-50% range.

**3.4.2. Fault Detection Reduction.** With fault detection reduction, very similar behavior was observed for the L and M coverage criteria, with fault detection loss of effectiveness in the 19-28% range for larger suites. E1 fared somewhat worse, losing from 37-41% of the faults for the largest suite size. CS-based reduction only lost 0-5% of the detectable faults across all suite sizes, which was comparable to the

performance of E2-based reduction even though the E2 technique yielded very little size reduction. Further investigation of the reasons behind these results is one of the primary motivations of the present work, discussed in the next section.

## 4. Results

### 4.1. Average Probability of Detecting Each Fault by Coverage Criterion

We now extend the analysis from our prior work [10] to include the newly proposed average probability of detecting each fault. To do this, we applied the CalcFaultDetectionProbability algorithm to the previously obtained fault and coverage matrices. Table 2 shows the average expected probability of detecting each fault after test suite reduction for each application and coverage technique. The box plots in Figure 3 show the other key statistics from individual fault probabilities.

**Table 2: Average Expected Probability of Detecting Each Fault After Test Suite Reduction**

| | *TP* | *TS* | *TW* |
|---|---|---|---|
| E1 | 0.51 | 0.52 | 0.47 |
| E2 | 0.92 | 0.88 | 0.96 |
| L | 0.84 | 0.69 | 0.77 |
| M | 0.80 | 0.69 | 0.72 |
| CS | 1.00 | 0.97 | 0.97 |

All of the five coverage techniques perform relatively consistently across applications. Event coverage, E1, fares the worst, while line and method coverage are comparable between 69-84% average probabilities. Event interaction coverage, E2, results in a very high average probability, but E2's usefulness in test suite reduction for these applications is limited for these subject applications and test universe as it results in very large reduced suite sizes [10]. The highest average probability is achieved with the call-stack coverage criterion, CS, with a 97-100% average probability of detecting each fault. This result shows quantitatively that many call stacks are highly correlated with fault-revealing test cases and therefore explains the extremely low percentage fault detection reduction observed when using the CS technique on test suites generated randomly from this pool [10].

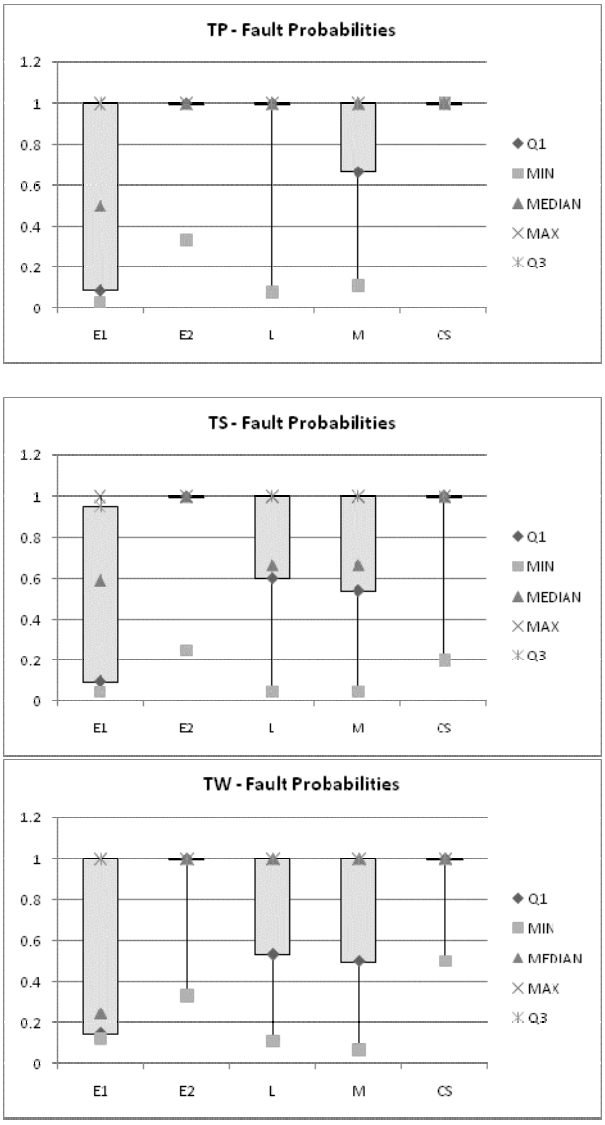


**Figure 3: Fault Probability Statistics**

### 4.2. Faults Always Detected After Reduction

In the case of TP, each known fault has at least one call stack coverage requirement that only appears in fault-detecting test cases. This situation is ideal for test suite reduction: Any reduced test suite derived from a full test suite while preserving coverage must then be able to detect the same set of faults. This observation led us to examine how many such faults with unique coverage requirements exist for the various coverage techniques. For each application, Figure 4 shows the number of faults that, if detected by an original test suite, must be detectable by any subsequent suite reduced using the given coverage technique. The relative performance of the different coverage techniques mirrors the results of for the average

probability of detecting each fault as well as the percentage fault detection reduction.

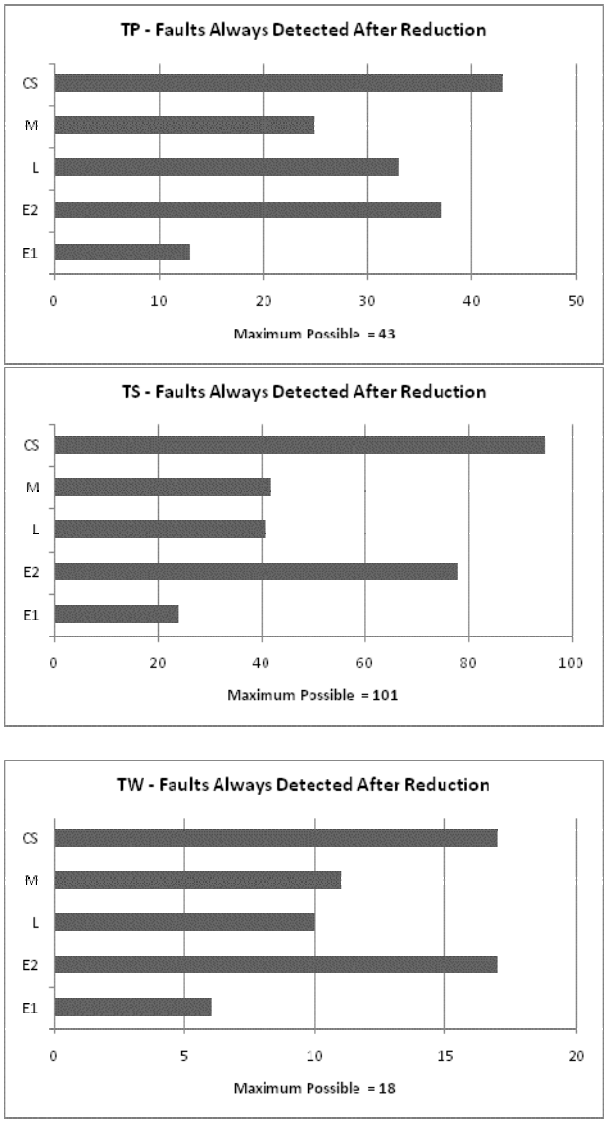


**Figure 4: Faults Always Detected After Reduction**

## 4.3. Faults Which May Be Missed After Reduction

We performed an analysis of the faults that *can* be missed by each technique as indicated by the average probability of detecting each fault. We chose to characterize each fault by its *difficulty*. Wong *et al*. define four quartiles of faults, Quartile-I, II, III, and IV, which can be detected by [0-25)%, [25-50)%, [50-75)%, and [75-100]%, respectively, of the test cases in the test pool [19]. However, by these standards, all of the known TerpOffice faults are "difficult" because they all fall solidly into the low end of Quartile-I, with the median percentage of detecting cases ranging from 0.13% for TerpPaint to 0.3% for TerpSpreadsheet. Thus, we instead characterize our faults into three buckets based on how many test cases detect them: Hard (1-2 detecting cases), Medium (3-5 detecting cases), and Easy (6 or more detecting cases). Table 3 shows the distribution of these faults by subject application.

**Table 3: Fault Difficulties**

| *Fault Class* | *TP* | *TS* | *TW* |
|---|---|---|---|
| Easy | 7 | 37 | 5 |
| Medium | 3 | 28 | 3 |
| Hard | 33 | 36 | 10 |

For each subject application and coverage criterion, we categorized the faults which may be lost after coverage-preserving test suite reduction. The results of this analysis appear in Table 4.

**Table 4: Faults with No Coverage Requirements Unique to Detecting Test Cases by Criterion and Difficulty**

| | *TP* | | *TS* | | *TW* | |
|---|---|---|---|---|---|---|
| *E1* | *Easy* | *7* | *Easy* | *26* | *Easy* | *2* |
| | *Med* | *3* | *Med* | *17* | *Med* | *2* |
| | *Hard* | *20* | *Hard* | *34* | *Hard* | *7* |
| E2 | *Easy* | *0* | *Easy* | *0* | *Easy* | *0* |
| | *Med* | *0* | *Med* | *0* | *Med* | *0* |
| | *Hard* | *6* | *Hard* | *23* | *Hard* | *1* |
| L | *Easy* | *6* | *Easy* | *13* | *Easy* | *3* |
| | *Med* | *1* | *Med* | *13* | *Med* | *3* |
| | *Hard* | *3* | *Hard* | *34* | *Hard* | *5* |
| *M* | *Easy* | *7* | *Easy* | *12* | *Easy* | *0* |
| | *Med* | *3* | *Med* | *13* | *Med* | *0* |
| | *Hard* | *8* | *Hard* | *34* | *Hard* | *7* |
| *CS* | *Easy* | *0* | *Easy* | *0* | *Easy* | *0* |
| | *Med* | *0* | *Med* | *1* | *Med* | *0* |
| | *Hard* | *0* | *Hard* | *5* | *Hard* | *1* |

The CS and E2 techniques, which only have a handful of faults overall that are not necessarily detected after reduction, show a distinct tendency for those faults to fall into the "Medium" and "Hard" difficulty buckets. For the other techniques, we only see such a trend for one of the three applications (specifically, TS). This analysis suggests that fault detection reduction in coverage-adequate reduced test suites may be related to fault difficulty only for certain coverage criteria.

### 4.4. Combining Coverage Criteria

Looking at the probability data for each fault, we observed that certain faults correlated more highly with different coverage criteria. This led us to examine the average probability of detecting each fault for *pairs* of criteria. Identifying effective pairs of coverage criteria is important to guide the choice of criteria to utilize in a multi-criteria test suite reduction approach such as the one proposed by Jeffrey and Gupta [8].

We assume a test suite reduction approach that maintains coverage relative to two distinct coverage criteria. For such a coverage criteria pair, the average probability of detecting a fault is then the maximum of the individual probabilities for each criterion in isolation. Data for this analysis appears in Table 5. The pair E1+E2 is not included because E2 *subsumes* E1 – that is, an E2-adequate suite is by definition E1-adequate. The technique M+CS is omitted for the same reason, namely that CS subsumes M. Note that because M includes library coverage data and L does not, L does *not* subsume M in these experiments.

**Table 5: Average Probabilities for Coverage Criteria Pairs**

| | *TP* | *TS* | *TW* |
|---|---|---|---|
| E1+L | ***0.88*** | ***0.71*** | ***0.91*** |
| E1+M | 0.80 | ***0.71*** | ***0.82*** |
| E1+CS | 1.00 | 0.97 | 0.97 |
| E2+M | ***0.97*** | ***0.91*** | 0.96 |
| E2+L | ***0.96*** | ***0.91*** | 0.96 |
| E2+CS | 1.00 | ***1.00*** | ***1.00*** |
| L+M | ***0.90*** | ***0.70*** | ***0.83*** |
| L+CS | 1.00 | 0.97 | 0.97 |

In Table 5, data points are highlighted in bold and italic where the combination of coverage criteria results in a better average probability of detecting each fault than either criterion in isolation. We see such an improvement in over half (14 of 24) of the combinations. This result suggests certain faults may be more highly correlated to different criteria, and thus combining multiple coverage criteria can dramatically reduce fault detection reduction. However, maintaining coverage adequacy with respect to additional criteria in test suite reduction will lead to larger reduced test suites. Indeed, many of the improvements in average probabilities in Table 5 involve the addition of the event-interaction criterion, E2, and E2 coverage adequacy in test suite reduction is known to lead to very little size reduction for these applications and test suites [10]. In test suite reduction, the tradeoff between fault detection and size reduction must be made based on situational engineering judgments.

### 4.5. Threats to Validity

Threats to external validity are factors that may affect generalizing the results to other situations. In this case, we have only applied the average probability of detecting each fault to a limited number of subject applications. These subject applications, as well as their test cases and known faults, are similar in size and origin and therefore may not be fully representative of the wider population of software applications. Other applications and types of faults may display different behavior under test suite reduction which could affect the utility of the metric. Additionally, we have evaluated the metric with respect to only five coverage criteria. The metric may appear more or less effective at evaluating other coverage criteria.

Threats to construct validity are factors that may cause our experiments to inadequately measure concepts of interest. Our primary threat to construct validity is the simple model of cost which treats all faults as equally severe. In practice, certain faults may be more or less critical to identify during regression testing. A more complex and realistic approach to creating the average probably of detecting each fault metric would account for this.

## 5. Related Work

There have been many prior studies of test suite reduction while holding coverage constant relative to some criterion. Wong *et al.* [19] use the all-uses coverage criterion and observe little or no fault detection effectiveness reduction in the reduced suites. They also consider fault difficulty and find a direct relationship between the ease of finding faults and the likelihood that they will be detected after reduction. In contrast, Rothermel *et al.* [16] reduce while holding all-edges coverage constant and find significant reductions in fault detection effectiveness. They contrast their results with an earlier study by Wong *et al.* [19] and suggest possible causes for the different conclusions. The fact that prior studies do not necessarily agree on the costs and benefits of test suite reduction suggests a need for new metrics beyond size reduction and fault detection effectiveness reduction. Our work provides an alternative analysis in the form of the average probability of detecting each fault.

Jeffery and Gupta [8] introduce a test suite reduction approach that combines "primary" and "secondary" coverage criteria in the reduction algorithm. The "selective redundancy" technique is so

named because certain test cases are known to add no additional coverage of the primary criterion, but by selecting such tests based on the second criterion, they are able to generate reduced test suites with fault detection effectiveness better than using either criterion alone. Our results discussed in Section 4.4 for the average probability of detecting each fault when using pairs of coverage criteria provide some additional evidence that combining criteria can be particularly effective in test suite reduction.

In their study of test suite reduction for model-based tests, Heimdahl and George raise the notion of an "ideal coverage criterion" which "would detect *all* faults in the system under test and *any* test-suite, large or small, providing this coverage would reveal the same faults [5]. Along the same line, Rothermel *et al.* point out that assuming an equal likelihood of selecting one of $k$ test cases that hit a coverage requirement, and only one test case detects a given fault, the probability of omitting the fault-detecting test case under coverage-based test suite reduction is $(k\text{-}1)/k$ [16]. To our knowledge, our work is the first to attempt to formalize, fully quantify, and evaluate these notions.

There are alternative approaches to test suite reduction that do not explicitly maximize test coverage relative to a traditional criterion. One such alternative is the "operational difference" technique of Harder *et al*. [3]. This approach builds up a reduced suite by pulling test cases from the test pool and adds them to the suite if they change the "operational abstraction" of the program's dynamic behavior. This process terminates when a certain number of consecutive cases produce no abstraction changes. Another approach that does not explicitly attempt to maximize test coverage is the cluster sampling of Leon and Podgurski [9]. The average probability of detecting each fault could be used to identify the best coverage criteria to be used as inputs for cluster formation.

## 6. Conclusions and Future Work

In this work, we defined a new metric for coverage-based test suite reduction based on the average probability of detecting each fault. We applied this metric to an existing set of test suite reduction experiments on GUI-intensive subject applications and contrasted the results using several different coverage criteria as well as combinations of criteria. We extended the analysis to count faults detected by a full test suite which must necessarily be detected by any coverage-adequate reduced test suite for the different criteria, and we considered the impact of fault difficulty.

Based on the analysis enabled by the average probability of detecting each fault metric, we are able to draw some conclusions about the relative utility of each coverage criterion in test suite reduction. For the subject applications in these experiments, test suite reduction based on call stacks provides the highest probability of detecting each fault in reduced test suites. Method (including libraries) and line coverage perform comparably, and length-1 event sequences are the least effective. This relative ranking is consistent with empirical performance of the various criteria against the traditional percentage fault detection reduction metric as observed in [10]. Thus, we conclude that the average probability of detecting each fault shows promise for identifying coverage criteria that work well for test suite reduction.

The "best" test suite reduction coverage criterion as measured by the metric presented in this paper may differ among other classes and styles of application. Thus, future work shall include non-GUI subject applications as well as GUI applications which were constructed using other approaches. Other cost functions may also affect the relative desirability of the different techniques. For example, an extended model may incorporate the notion of fault severity. In the future, we intend to extend test suite reduction metrics to incorporate such factors.

## 7. Acknowledgements

The authors would like to thank Xun Yuan for preparing the TerpOffice subject applications.

## 8. References

[1] Beck, K. (2000): Extreme Programming Explained: Embrace Change. Addison Wesley, 2000.

[2] Elbaum, S.; Gable, D. & Rothermel, G. (2001),The impact of software evolution on code coverage information, in 'Software Maintenance, 2001. Proceedings. IEEE International Conference on', pp. 170--179.

[3] Harder, M., Mellen, J., and Ernst, M. D. (2003), Improving test suites via operational abstraction. Proceedings of the 25th International Conference on Software Engineering, pp. 60-71, 2003, Porland, Oregon, United States.

[4] Harrold, M.; Gupta, R. & Soffa, M. (1990),A methodology for controlling the size of a test suite, in 'Software Maintenance, 1990., Proceedings., Conference on', pp. 302--310.

[5] Heimdahl, M. & George, D. (2004),Test-suite reduction for model based tests: effects on test quality and implications for testing, in 'Automated Software Engineering, 2004.

Proceedings. 19th International Conference on', pp. 176--185.

[6] JavaCCTAgent information on the web at http://sourceforge.net/projects/javacctagent/, April, 2007.

[7] jcoverage information on the web at http://www.jcoverage.com/, April, 2006.

[8] Jeffrey, D. and Gupta, N. (2005), Test suite reduction with selective redundancy. IEEE International Conference on Software Maintenance (ICSM) 2005, pages 549-558, Budapest, Hungary, 2005.

[9] Leon, D. and Podgurski, A. (2003), A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), November 2003, Denver, Colorado, United States.

[10] McMaster, S, and Memon, A. (2006), Call Stack Coverage for GUI Test-Suite Reduction, Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006), Raleigh, NC, USA, Nov. 6-10 2006.

[11] McMaster, S. and Memon, A. (2005). Call stack coverage for test suite reduction. IEEE International Conference on Software Maintenance (ICSM) 2005, pages 539-548, Budapest, Hungary, 2005.

[12] Memon,A., Banerjee, I., & Nagarajan, A. (2003), GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, Proceedings of The 10th Working Conference on Reverse Engineering, Nov. 2003.

[13] Memon, A., Nagarajan, A., and Xie, Q. (2005), Automating regression testing for evolving GUI software. Journal of Software Maintenance and Evolution: Research and Practice, 17(1):27.64, 2005.

[14] Memon, A., Soffa, M. L., and Pollack, M. (2001), Coverage criteria for GUI testing. ESEC / SIGSOFT FSE 2001, pages 256-267, Vienna, Austria, 2001.

[15] Memon, A. and Xie, Q (2005). Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE Transactions on Software Engineering, vol. 31, no. 10, pp. 884-896, October, 2005.

[16] Rothermel, G.; Harrold, M.; Ronne, J. & Hong, C. (2002), 'Empirical studies of test suite reduction', Journal of Software Testing, Verification, and Reliability 4(2).

[17] Rothermel, G., Untch, R., Chu, C., and Harrold, M.J. (2001). Test case prioritization. IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929-948, October, 2001.

[18] Rothermel, G., Harrold, M.J., Ostrin, J., and Hong, C. (1998), An empirical study of the effects of minimization on the fault detection capabilities of test suites. Proceedings of the International Conference on Software Maintenance, pages 34-43, November 1998.

[19] Wong, W., Horgan, J., London, S., & Mathur, A. (1995), Effect of Test Set Minimization on Fault Detection Effectiveness, Proceedings of the International Conference on Software Engineering, Seattle, WA, 1995.