TEST SUITE REDUCTION WITH SELECTIVE REDUNDANCY

by

Dennis Bernard Jeffrey

A Thesis Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 0 5

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: _____

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

_____ _____

Neelam Gupta
Assistant Professor of Computer Science

Date

ACKNOWLEDGEMENTS

Special thanks are in order for my advisor, Dr. Neelam Gupta of the Department of Computer Science, the University of Arizona, who invested considerable time and effort to provide me with much-needed guidance and advice during my work on this thesis, and who is serving on my thesis committee. I also owe a debt of gratitude to Dr. Rajiv Gupta and Dr. Richard Snodgrass, also of the Department of Computer Science, the University of Arizona, who are serving on my thesis committee and helping me to succeed in my current research endeavors.

I would further like to thank Dr. Gregg Rothermel of the Department of Computer Science and Engineering, the University of Nebraska, for providing me with access to the Siemens suite of subject programs, faulty versions, and test case pools.

DEDICATION

This thesis is dedicated to my dad, Dennis Gerard Jeffrey, as well as to my brother, Orion John Jeffrey, and my sister, Aurora Anne Jeffrey.

I would also like to dedicate this thesis to my grandparents, Dennis Bernard Jeffrey and Mary Margaret Jeffrey.

Thanks for everything, everyone!

TABLE OF CONTENTS

TABLE OF CONTENTS – *Continued*

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Maintaining test suites for software can become increasingly difficult as suite sizes grow over time. Test suite minimization techniques are therefore used to remove the test cases from a suite that have become redundant with respect to a particular coverage criterion. However, minimizing a suite with respect to one coverage criterion may cause the suite to lose coverage with respect to other coverage criteria, and this may compromise the fault detection effectiveness of the suite. To address this, we propose the idea of including selective coverage redundancy during suite reduction. Our approach for suite reduction keeps tests that are redundant with respect to the primary coverage criterion as long as they are not redundant with respect to another secondary criterion. Empirical results show that, compared to existing minimization techniques, our approach has a strong tendency to improve the fault detection effectiveness of reduced suites without significantly impacting suite size reduction.

CHAPTER 1

Introduction

The development of software is a lengthy process that involves much more than writing code. Requirements of the software must be generated and understood so that a specification of the software can be created, the code must be written, the software must be thoroughly tested to eliminate bugs and to ensure that the software matches the specification, and over time, the software must be maintained. All of these stages together comprise the *software development lifecycle*. While there are many models that software developers may follow in order to carry out this process, the fact remains that software will change over time. As a result, the testing and retesting of software occurs continuously during the software development lifecycle.

Software testing is the process of analyzing software to promote confidence that the actual behavior of the software correctly adheres to its specification. Software testing usually involves execution of the software on a particular set of input and a comparison of the actual software output with the expected output. This set of program input and the corresponding expected output is called a *test case* for the program. Here, we will generally think of a test case as simply being a particular set of input for a given program. A collection of test cases is called a *test suite* or a *test set*. Software testers will typically maintain a variety of test suites to be used for the testing of software.

Each test case that is created for a test suite exercises certain *requirements* of the software. A requirement is some entity in the software that may be exercised (covered) by a test case, and may be either *white-box* (dealing with the code itself) or *black-box* (dealing with the specification of the software). Such requirements may include coverage of statements, decisions, definition-use pairs, or paths of interest (all white-box), or coverage of special input values and output values derived from the specification (black-box). A test case will often be created specifically to cover

a certain requirement or set of requirements, since exercising more unique requirements implies that more of the software is being tested. For example, a test case may be created to exercise a particular statement or decision in the software that no other tests yet exercise. As another example, Hutchins et al. [20] conducted an experiment that involved creating test cases for programs such that every exercisable edge and definition-use pair in the program was exercised by at least 30 test cases.

As software grows and evolves, so too do the accompanying test suites. More test cases will be required over time to test for new or changed functionality that has been introduced to the software, or to guard against a particular bug that has been previously discovered. As time progresses, some test cases in a test suite will likely become *redundant* with respect to a particular coverage criterion, as the specific coverage requirements exercised by those redundant test cases are also exercised by other test cases in the suite. Notice that the property of a test case being redundant is relative to a specific set of coverage requirements. For example, a test case exercising a certain set of statements A is redundant relative to the statement coverage of a test suite if the union B of all the statements exercised by the other test cases in the suite is such that $A \subseteq B$. However, that same test case may actually not be redundant relative to, for instance, definition-use pair coverage, if the test case exercises a unique definition-use pair that is not exercised by any other test case in the suite. It is important, therefore, to remember that redundancy of a test case is a property that is relative to some specific set of requirements.

As test suites grow in size, they may become so large that it becomes desirable to reduce the sizes of the suites. This is especially true in situations where an *extreme programming* approach is followed which, among other guidelines, stresses the daily testing of software from the very first day of software development. Test suite minimization is one general technique that has been proposed to address the problem of excessively large test suites.

1.1 Test Suite Minimization Defined

Test suite minimization is an optimization problem with the following goal: to find a minimally-sized subset of the test cases in a suite that exercises the same set of coverage requirements as the original suite. The key idea behind minimization techniques is to remove the test cases in a suite that have become redundant in the suite with respect to the coverage of some particular set of program requirements. The minimization problem can be formally stated as follows:

The Formal Test Suite Minimization Problem

Given: a set (test suite) T of candidate test cases t_1, t_2, ..., t_n and some set of coverage requirements R, where each test case covers a set of software requirements r_1, r_2, ..., r_n, respectively, such that $r_1 \cup r_2 \cup ... \cup r_n = R$

Problem: find a minimally-sized subset of test cases $T' \subseteq T$, comprised of tests t'_1, t'_2, ..., t'_m, each test covering a set of software requirements r'_1, r'_2, ..., r'_m, respectively, such that $r'_1 \cup r'_2 \cup \cdots \cup r'_m = R$

The test suite minimization problem is an instance of the more general *set-cover problem*, which when given as input a collection S of sets, each set covering a particular group of entities, is to find a minimally-sized subset of S providing the same amount of entity coverage as the original set S. The set-cover problem has been shown to be **NP**-Complete [13], and therefore there does not exist any known polynomial-time algorithm to optimally solve the minimization problem in general. Nevertheless, there has been some research work [5, 19] in the area of computing optimally-minimized suites. Most other research work into minimization has relied on heuristics for computing near-optimal solutions. Chvatal [8] presented a simple greedy heuristic for the set-cover problem in which each candidate set has a cost associated with it. Jones and Harrold [23] described two minimization heuristics that are designed specifically to be used in conjunction with the relatively strong modified condition/decision coverage criterion; one algorithm builds a minimized

suite incrementally by identifying essential and redundant test cases, while the other algorithm is based on a prioritization technique that simply stops computing before all test cases in a suite have been prioritized. Agrawal's work [2] implies a framework for minimization of suites using the notions of mega blocks and global dominator graphs. An algorithm based on a greedy heuristic for reducing the size of a test suite (referred to henceforth as the HGS algorithm) was developed by Harrold, Gupta and Soffa [16]. This heuristic is presented in detail in Appendix A of this thesis.

1.2 Making the Case for Test Suite Reduction and the Intuition Behind Our Approach

It is often the case that software testers are subject to time and resource constraints when testing software. Due to such constraints being present for software retesting every time the software is modified, it is important to develop techniques that keep test suite sizes manageable for testers. When a collection of test suites becomes very large, a tester may not have enough time or resources available to test the software using every test case in each suite. In such a situation, the tester has no choice but to run fewer test cases to stay within the allowed time and resource constraints. The problem for the tester is then to decide which test cases are the most important and should therefore be run. This is where test suite minimization techniques become helpful.

Virtually all previous research in the area of test suite minimization has shown that suite sizes can indeed be reduced significantly under various minimization techniques. A lingering question deals with how well those minimized suites compare to their non-minimized counterparts when evaluated according to other criteria besides suite size.

Comparing minimized suites to their non-minimized counterparts in terms of another criterion (besides suite size reduction) may involve a measure of suite quality. Since the purpose of test cases is to reveal faults in software, one measure of suite quality is the ability of a suite to detect faults in software. Since test suite

minimization removes test cases from suites, minimized suites may be weaker at detecting faults in software than their non-minimized counterparts.

Fault detection effectiveness is intuitively a measure of the ability of a test suite to detect faults in software. Of course, it is a problem in itself just to determine the best way of measuring the fault detection effectiveness for a particular suite. The approach taken in existing research has been to take a base program (an *oracle*) and create multiple *faulty versions* of the program such that each faulty version is identical to the base version, except a single error has been seeded in the software. When a test case is executed on a particular faulty version, that fault may or may not be *detected* (exposed). Researchers define a test case as *detecting a fault* if the output of the faulty version, when run on a particular test case, differs from the output of the oracle when run on that same test case.

As an example, consider a base program for which we have constructed 10 faulty versions. Suppose we have a test suite T_1 that detects 8 of the 10 faults from errors that we have seeded. Suppose a different test suite T_2 detects only 3 of the 10 faults from errors that we have seeded. Then suite T_1 can be viewed as being "better" than T_2 in terms of fault detection effectiveness, since T_1 is more effective at detecting faults with respect to our set of faulty versions. Assuming that we measure the effectiveness of a suite as the percentage of faults detected, then T_1 would be 80% effective while T_2 would only be 30% effective. Clearly, the fault detection effectiveness of suites computed in this way is highly dependent upon the set of faulty versions used, including the number of faulty versions, how the errors are distributed in the software, and what types of errors are seeded. Just because one test suite T_1 is more effective than another suite T_2 with respect to one set of faulty versions does not necessarily imply that T_1 is also more effective than T_2 with respect to some of other set of faulty versions.

Intuitively, whenever a test case is thrown away from a suite, the suite loses an opportunity for detecting faults. Test suite reduction, therefore, ultimately involves a trade-off between the size of a suite and its fault detection effectiveness. However, it is reasonable to expect that if two distinct test cases in a suite are very similar

in terms of how they each cover the software, it should be relatively safe to throw away one of those test cases without significantly compromising the fault detection effectiveness of the suite.

Previous research [18, 34, 35] has suggested that test suite minimization may achieve high suite size reduction, but at the expense of severe or unacceptable fault detection loss, when minimization is carried out with respect to structural coverage criteria such as edge-coverage. We contend, to the contrary, that these results are actually *encouraging* for test suite minimization! For example, Rothermel et al. [34] showed many suites achieving over 80% suite size reduction while achieving considerably less percentage fault detection loss on average (around 50% detection loss on average). Also, Heimdahl and George [18] showed suites experiencing between 82% and 94% size reduction on average while losing only between 7% and 16% fault detection effectiveness. It is rather remarkable that throwing away nearly all the test cases from a suite generally results in a significant degree of retention of the original suite's ability to detect faults. This fault detection retention can be attributed to the use of coverage criteria during minimization: in another work by Rothermel et al. [35], it was shown that suites minimized with respect to edge coverage consistently retained more fault detection effectiveness than randomly-reduced suites of the same sizes.

Despite these encouraging results, however, there is still clearly much room for improvement. The goal of test suite minimization techniques is to achieve significant suite size reduction without experiencing significant fault detection loss. The work of Wong et al. [43] suggests that this goal is possible, as their experiments regarding test suite reduction showed suites achieving 9% to 68% size reduction while only experiencing 0.19% to 6.55% fault detection effectiveness loss. The goal of this thesis is to further improve the fault detection capabilities of reduced suites without significantly impacting suite size reduction. To achieve this, we view test suite minimization not as an optimization problem, but rather, as the problem of *test suite reduction*: making test suite sizes small — but not necessarily minimal — with respect to minimization criteria.

Because removing redundant tests from a suite according to one criterion will almost certainly throw away some important tests that are not redundant according to other criteria, we suggest the following: test suite reduction with the goal of achieving high suite size reduction with little to no loss in fault detection effectiveness, in general, should incorporate some notion of *keeping certain test cases that are redundant* with respect to the particular set of program requirements by which minimization is carried out. In this thesis, we therefore propose that minimization should be viewed from the opposite perspective from which it is traditionally viewed: reduction techniques should seek to include selective redundancy in reduced suites with respect to a particular coverage criterion, rather than, as is traditionally done, trying to remove as much coverage redundancy as possible.

We now present a specific motivating example for our new approach to test suite reduction, which led us to the algorithms described in detail in Chapter 2, and which motivated the particular experiments we conducted as described in Chapter 3.

1.3 Motivational Example

We now present a simple example program that motivated our idea of selectively keeping some of the test cases in a reduced test suite that are redundant according to the coverage criterion for suite reduction. The example program and a corresponding branch coverage adequate test suite T with some redundant test cases (with respect to branch coverage) are shown in Figure 1.1.

Suppose suite T was generated specifically to achieve only branch-coverage adequacy. Our approach in this example is then to minimize suite T by removing branch-redundant test cases. However, we want to select *some* of those branch-redundant tests for inclusion in the reduced suite. In order to decide which branch-redundant tests we want to include, we decide that once a test case is determined to be redundant with respect to branch coverage, we will include it in the reduced suite if and only if it increases the cumulative definition-use pair

```
1:      read(a,b,c,d);
B1:     if ( a > 0 )
2:          x = 2;
3:      else
4:          x = 5;
5:      endif
B2:     if ( b > 0)
6:          y = 1 + x;
7:      endif
B3:     if ( c > 0)
B4:         if ( d > 0)
8:              output(x);
9:          else
10:             output(10);
11:         endif
12:     else
13:         output(1 / (y − 6));
14:     endif
```

A Branch Coverage Adequate Suite T

T_1: (a = 1, b = 1, c = −1, d = 0)
T_2: (a = −1, b = −1, c = 1, d = −1)
T_3: (a = −1, b = 1, c = −1, d = 0)
T_4: (a = −1, b = 1, c = 1, d = 1)
T_5: (a = −1, b = −1, c = 1, d = 1)

Figure 1.1: An example program with a branch coverage adequate test suite T.

coverage of the reduced suite (i.e., if and only if it is *not* redundant with respect to definition-use pair coverage at the time it becomes branch-redundant). This will allow us to select branch-redundant tests that still happen to exercise "unique situations" in the code, and therefore that are likely to detect new faults. In order to accomplish this, we require for each test case the set of branches and definition-use pairs covered by that test. The branches covered by each test case are marked with an X in the respective columns in Table 1.1 (for example, branch B_1^T refers to the TRUE branch of condition B_1), and the definition-use pairs covered by each test case are marked with an X in the respective columns in Table 1.2.

To minimize T, our goal is to find a subset of T that provides the same requirement coverage as T. We use branch coverage as the primary criterion for minimization, and definition-use pair coverage as a secondary criterion that allows us to determine whether to keep a branch-redundant test case. We begin by first noticing that test case T_1 is the only test case covering branch B_1^T and test case T_2 is the only test case covering branch B_4^F. Therefore, the reduced suite must include both tests T_1 and T_2 to retain branch coverage adequacy since T_1 and T_2

Test Case:	B_1^T	B_1^F	B_2^T	B_2^F	B_3^T	B_3^F	B_4^T	B_4^F
T_1:	X		X			X		
T_2:		X		X	X			X
T_3:		X	X			X		
T_4:		X	X		X		X	
T_5:		X		X	X		X	

Table 1.1: Branch coverage information for test cases in T. Each column except the left-most column describes the coverage of branches in the program.

Test Case:	x(2,6)	x(4,6)	x(4,8)	y(6,13)	a(1, B_1)	b(1, B_2)	c(1, B_3)	d(1, B_4)
T_1:	X			X	X	X	X	
T_2:					X	X	X	X
T_3:		X		X	X	X	X	
T_4:		X	X		X	X	X	X
T_5:			X		X	X	X	X

Table 1.2: Definition-use pair coverage information for test cases in T. Each column except the left-most column describes the coverage of def-use pairs in the program.

each uniquely cover a branch. Notice that after selecting both T_1 and T_2, test case T_3 becomes redundant with respect to branch coverage since all the branches covered by T_3 are also covered by T_1 and T_2. However, T_3 covers the definition-use pair $x(4, 6)$, which is not covered by either T_1 or T_2. Hence, T_3 executes a unique situation not executed by either T_1 or T_2, and therefore it is important to include T_3 in the reduced suite so that we can retain more of the fault detection effectiveness of the original suite. Thus, at this point our reduced suite includes T_1, T_2, and T_3. Notice that our reduced suite now covers all the branches except for B_4^T, so either T_4 or T_5 may be selected to achieve full branch-coverage in the reduced suite. Suppose that T_4 is selected. Then the reduced suite now contains T_1, T_2, T_3, and T_4, which covers not only all the branches covered by the original suite, but also all of the definition-use pairs covered by the original suite. Thus, the remaining test case T_5 is redundant with respect to both branch coverage and definition-use pair coverage, so it is not selected. The final reduced suite that is computed is $\{T_1, T_2, T_3, T_4\}$. In the example in Figure 1.1, notice that test case T_3 exposes a divide-by-zero error

at line 13 (none of the other test cases expose this fault). Hence, our reduced suite retains the fault detection effectiveness from the original suite, with respect to this divide-by-zero error.

Our approach as described above seeks to remove branch-redundancy, except for when that redundancy adds new definition-use pair coverage to the reduced suite. It is interesting to see how the results of our approach described above would compare to the results of a traditional minimization technique that would seek to remove as much branch-redundancy as possible, without any regard for definition-use pair coverage. A technique that accomplishes this is the HGS algorithm [16] presented in Appendix A of this thesis. Following this approach with respect to branch coverage only, tests T_1 and T_2 are selected first because they are necessary in the reduced suite. Then either T_4 or T_5 can finally be selected to achieve full branch-coverage adequacy. In either case, the reduced suite will be one test case smaller than the reduced suite computed using our approach, but the reduced suite computed here will *not* contain the error-revealing test case T_3 that our approach includes.

A further reasonable question is how our approach using both branch coverage and definition-use pair coverage may be different from using simply the definition-use pair coverage criterion as the minimization criterion. Using the HGS algorithm, the minimally-sized reduced suite $\{T_1, T_4\}$ will be computed, which covers all of the definition-use pairs covered by the original suite. Notice that here, the error-revealing test case T_3 is again not present in the reduced suite, and further, the reduced suite is not even branch-coverage adequate (it does not cover branch B_2^F or branch B_4^F).

Yet another reasonable question is how our approach with selective redundancy may compare to using a traditional approach where the minimization requirement set is comprised of the union of branch coverage and definition-use pair coverage. Using the HGS algorithm, the computed reduced suite in this case turns out to be $\{T_1, T_2, T_4\}$. Notice that again, the error-revealing test case T_3 is not included in the reduced suite since the algorithm happens to never select this particular test case. This is true even though the minimization algorithm is taking into account

both sets of criteria in this example.

The above examples suggest that our approach to test suite reduction with retaining selective redundancy in the reduced test suite may be preferable to the traditional minimization approaches. The above examples also provide some insight into why this is so. Definition-use pair $x(4, 6)$ is exercised by both T_3 and T_4, but test case T_3 exercises a combination of branch outcomes (namely $x(4, 6)$ and $y(6, 13)$) that are not executed by any other test case, and this combination of branch outcomes happens to expose a divide-by-zero error. In the minimization schemes without selective redundancy described above, T_3 always becomes redundant due to the particular other test cases that happen to be added early on in the minimization process. However, in our approach, as soon as a test case becomes redundant according to branch coverage, we add it to the reduced suite if it adds new definition-use pair coverage. Therefore, this allows T_3 to be added to the reduced suite before T_4 is added, since T_3 is not definition-use pair redundant at the time it becomes branch-redundant. Thus, while our approach in this example achieves slightly less suite size reduction, it is also more likely to retain test cases that execute different combinations of data-flow, and therefore it is more likely to retain more of the fault detection effectiveness of the original suite.

1.4 Chapter Summary and Thesis Overview

This chapter has introduced the notion of test suite minimization and the general problem that minimization is meant to address: keeping test suite sizes small enough to be reasonably manageable by testers. The chapter has also discussed the current problem in research of trying to understand how test suite minimization can influence the fault detection effectiveness of reduced suites. It is argued that techniques for test suite *reduction* (rather than test suite *minimization*) should be pursued, and these techniques should include some approach to including selective coverage redundancy in reduced suites in order to improve upon existing minimization techniques in terms of retaining more fault detection effectiveness while still achieving

relatively high suite size reduction.

The remainder of this thesis proposes a technique for reduction with selective redundancy that is based on ideas stemming from our motivational example described earlier. The idea for our approach is to reduce a suite using an existing technique for minimization, but as soon as a test case is identified as redundant with respect to the minimization criterion, it will be selected if and only if it increases the cumulative coverage of some *other* criterion. This is because such tests will likely cover "new situations" in the code (new combinations of exercised requirements), and therefore, they are important to be kept in order to increase the likelihood of retaining fault detection effectiveness. We present one specific implementation of our approach based on the HGS minimization algorithm.

This thesis also includes an empirical study in which we implemented our algorithm and conducted experiments using the well-known Siemens suite of subjects [20] that is used in other minimization research [5, 14, 20, 34, 41]. Our results show that our approach has a strong tendency to improve upon traditional minimization techniques in terms of retaining more fault detection effectiveness of reduced suites without severely compromising suite size reduction.

The remainder of the thesis is organized as follows. In the next chapter, we describe in detail our new technique for test suite reduction with selective redundancy. In Chapter 3, we describe an experimental study comparing traditional minimization techniques with our new reduction technique that attempts to include selective coverage redundancy. Chapter 4 discusses previous work that is related to test suite minimization and fault detection effectiveness. We present the conclusions of our work and our plan for future work in Chapter 5. Finally, Appendix A describes the original HGS algorithm in detail.

CHAPTER 2

Our Approach to Reduction with Selective Redundancy

2.1 Our General Approach

Our proposed approach to test suite reduction was motivated by the following key observation: test suite minimization techniques attempt to throw away test cases that are redundant with respect to the coverage criterion for minimization. However, throwing away redundant test cases may result in significant loss in fault detection capability, since test cases that are redundant with respect to a particular criterion may still exercise "unique situations" in software (these tests may *not* be redundant with respect to other coverage criteria). Consequently, we believe that the test suite minimization problem should be viewed from the perspective of keeping redundant test cases that may exercise different situations in program execution, even though they may be redundant with respect to the coverage criterion for minimization. The success of this approach relies on determining how to identify when a test case that is redundant with respect to a coverage criterion may actually exercise a unique situation during program execution that is highly likely to expose new faults in software. We need an additional set of requirements to determine whether a redundant test case (with respect to the criterion for minimization) actually exercises a new combination of requirements and should therefore be kept in the reduced suite. In order to make a distinction between the primary coverage criterion used for test suite reduction and the additional requirements whose coverage will determine whether a redundant test case should be added to the reduced suite, we respectively refer to these two sets of requirements as the *primary* and *secondary* criteria.

One possible approach that does not keep selective coverage redundancy would be to simply *minimize* with respect to both the primary and secondary criteria, removing those test cases that are redundant with respect to the coverage of the

union of the primary and secondary criteria. However, this approach does not account for the coverage of *other* criteria that may not be contained in the primary and secondary requirement sets; coverage loss of this other criteria may still occur if minimization is carried out with respect to both the primary and secondary criteria only. One possible solution to this problem would be to choose a very strong coverage requirement for the primary or secondary criterion, such as all-paths coverage, which would leave very little room for other requirement coverage to be lost during minimization. In practice, however, this would likely severely compromise the size reduction of suites, leading to an unacceptably small amount of suite size reduction.

Instead, a new approach is required that selectively keeps coverage redundancy (for those coverage requirements that are still likely to promote significant suite size reduction), that may still allow for the retention of additional requirement coverage for those requirements that we may not be explicitly considering during reduction. The general idea for our approach is as follows: when a minimization algorithm selects the next test case to add to the reduced suite according to the primary minimization criterion, we then identify the other tests that, given the test case just selected by the minimization algorithm, have just become redundant with respect to the reduced suite according to the primary coverage criterion. Among those redundant tests, we then check whether or not each test is also redundant with respect to the secondary criterion. If a test is redundant with respect to the secondary criterion as well, we throw it away. If a test is not redundant with respect to the secondary criterion, then we add the test case to the reduced suite. We then allow the original minimization algorithm to continue and select the next test according to the primary coverage criterion.

Notice that in our approach, a test case will be selected for inclusion in a reduced suite either (1) according to the primary coverage criterion if the test covers a new primary coverage requirement, or (2) according to the secondary coverage criterion if the test is redundant with respect to the primary criterion but covers a new secondary coverage requirement. Thus, our approach clearly selects some tests that are redundant according to the primary coverage criterion. Further, our approach

may also select some tests that are redundant according to the secondary coverage criterion. For example, suppose that edge coverage is used as the primary coverage criterion and definition-use pair coverage is used as the secondary coverage criterion. A test case T_1 may be chosen for selective edge coverage redundancy if it covers a unique definition-use pair d_1 not already covered by the reduced suite. Later, another test case T_2 may also be chosen for selective edge coverage redundancy if it covers a unique definition-use pair d_2 not already covered by the reduced suite. But suppose T_2 also happens to cover d_1. Then test case T_1, which is now already in the reduced suite, may become redundant with respect to both the primary and secondary coverage criteria. However, observe that according to this approach, *all test cases selected for inclusion in the reduced suite will cover a unique path.* The reason is because a test case will always only be selected for inclusion in the reduced suite if it covers either a unique edge or a unique definition-use pair not already covered by another test case in the reduced suite; a new edge or a new definition-use pair can only be covered by exercising a new path. Our approach therefore allows for the inclusion of additional tests that may be redundant with respect to both the primary and secondary criteria, but that are still likely to exercise "new situations" in the code in the sense that they exercise new combinations of the primary coverage requirements. This allows our reduction algorithm to implicitly select those additional tests exercising some other coverage requirements that are not being explicitly accounted for during suite reduction, and these tests are important for retaining more of the fault detection effectiveness of the suites. Moreover, since our algorithm does not select *all* the tests covering unique paths, we may still expect a significant amount of suite size reduction using our approach.

Notice that our approach is very general in terms of selecting the primary and secondary criteria. Even requirements derived from black-box testing can be used as secondary requirements in place of or in conjunction with the statement or branch coverage criteria that may be used as a primary criterion in this approach. Also, any existing test suite minimization algorithm that seeks to eliminate coverage redundancy can be modified to incorporate our idea of generating reduced test suites

that selectively retain some of the test cases that are redundant with respect to the given coverage criterion for minimization.

Our general approach to test suite reduction is presented in the pseudocode in Figure 2.1.

```
input:
        An existing minimization technique M
        A test suite T
        Primary/secondary coverage information for all tests in T
output:
        RS: a representative set of tests from T

algorithm ReduceWithSelectiveRedundancy
begin
        RS := {};
Step 1: Start running M;
        while T is not empty do
Step 2:     nextTest := the next test selected by M for inclusion in RS;
            RS := RS ∪ {nextTest};
            T := T − {nextTest};
Step 3:     redundant := other tests from T that, given the updated RS, have just
                become redundant w.r.t. the primary coverage criterion;
            T := T − redundant;
Step 4:     while ∃ test t in redundant s.t. t not redundant w.r.t. secondary criterion do
                toAdd := the test in redundant contributing max additional secondary
                    coverage to RS;
                RS := RS ∪ {toAdd};
            endwhile
            redundant := {};
        endwhile
        return RS;
end ReduceWithSelectiveRedundancy
```

Figure 2.1: Pseudocode for our general approach to reduction with selective redundancy.

The main steps of our general approach are as follows.

Step 1: Start Running the Existing Minimization Algorithm

We first allow the existing minimization algorithm to begin and start looping, continually selecting the next test case to include in the reduced suite with respect to the primary minimization criterion until the original set of candidate test cases becomes empty. Our approach is implemented within this outer loop.

Step 2: Select the Next Test Case According to the Primary Criterion

The existing minimization algorithm selects the next test case to include in the reduced suite according to the primary coverage requirements. We add this test to the reduced suite and remove it from the candidate set of test cases.

Step 3: Identify Redundant Tests with Respect to the Primary Criterion

Next, given the test case just selected according to the primary criterion, we identify the other test cases in the candidate set of tests that have become redundant with respect to the reduced suite according to the primary criterion. We remove these redundant tests from the candidate set of tests, since they will never be selected in the future by the existing minimization algorithm according to the primary requirement set.

Step 4: Add Selective Primary Coverage Redundancy to the Reduced Suite

This is the most important part of our approach, where selective coverage redundancy is considered. In this step, we analyze the set of primary coverage-redundant tests and, as long as there exists a test case in this redundant set that contributes to the secondary requirement coverage of the reduced suite, we select the next test case that contributes the most secondary requirement coverage to the reduced suite. After all redundant tests have been processed (some may be selected and some may not be selected), we empty the redundant set and allow the existing minimization algorithm to continue selecting the next test case according to the primary coverage criterion.

This section has presented a high-level, general view of our approach. In the next

section, we present a specific implementation of our approach based on a particular existing minimization heuristic.

2.2 Application of Our Approach to an Existing Minimization Heuristic

In this thesis, we specifically consider the HGS algorithm [16] for test suite minimization (as described in Appendix A) as a basis for our approach. The reason for this choice is that for our empirical studies, we wanted to be able to compare our new technique with the technique studied by Rothermel et al. [34]. Since these authors chose to study the HGS algorithm, we have made the same choice. We developed our algorithm for test suite reduction with selective coverage redundancy by adding a new step to the HGS heuristic: instead of always throwing away a test case that is redundant with respect to the primary requirement coverage criterion for test suite minimization in the original HGS algorithm, our new step examines the redundant test case with respect to a set of secondary requirements, and uses this additional information to decide whether or not to add the test case to the reduced suite. Figures 2.2, 2.3 and 2.4 show our modified version of the HGS algorithm, updated to include selective coverage redundancy in reduced suites.

The input and output for our algorithm are described in Figure 2.2. The main algorithm for test suite reduction with selective redundancy is described in Figure 2.3, and Figure 2.4 shows a helper function called *SelectTests* that is used by the main algorithm and which comes from the original HGS algorithm.

define:
Set of primary requirements for minimization: r_1, r_2, ..., r_n.
Set of secondary requirements: r'_1, r'_2, ..., r'_m.
Test cases in original (non-reduced) test suite: t_1, t_2, ..., t_{nt}.
input:
T_1, T_2, ..., T_n: test case sets for r_1, r_2, ..., r_n respectively.
T'_1, T'_2, ..., T'_m: test case sets for r'_1, r'_2, ..., r'_m respectively.
output:
RS: a reduced subset of t_1, t_2, ..., t_{nt}

Figure 2.2: Input and output for our algorithm.

As shown in Figure 2.2, our algorithm takes as input two collections of

associated test case sets. T_1, T_2, ..., T_n are the testing sets corresponding to primary requirements such that T_i contains the set of test cases that cover the primary requirement r_i. Similarly, T'_1, T'_2, ..., T'_m are the testing sets corresponding to secondary requirements such that T'_i contains the set of test cases that cover the secondary requirement r'_i. We now describe the steps of the main reduction algorithm shown in Figure 2.3.

Step 1: Initialization

This step simply initializes the variables and data structures that will be maintained throughout the execution of the algorithm. After initialization, the main program loop begins which attempts to greedily select test cases that cover the hardest-to-cover primary requirements that are currently uncovered by the reduced suite (initially, the reduced suite is empty). To consider the hardest-to-cover requirements first, the uncovered primary requirements are considered in increasing order of associated test case set cardinality. This is because requirements that are exercised by the fewest number of test cases are exactly those requirements that are the most difficult to cover by test cases in the suite.

Step 2: Select the Next Test Case According to the Primary Requirements

The algorithm next collects together all of the test cases comprising the testing sets of the current cardinality that are associated with uncovered primary requirements. This is the candidate pool from which the next test case (with respect to the primary requirement set) will be selected for inclusion in the reduced suite. The algorithm decides which of the tests in the pool to select within the SelectTest helper function. This function gives preference to the test case that covers the most uncovered requirements whose testing sets are of the current cardinality. In the event of a tie, the algorithm recursively gives preference to the test case among the tied

```
algorithm ReduceWithSelectiveRedundancyHGS(T_1 ··· T_n, T'_1 ··· T'_m)
Step 1: unmark all r_i and r'_i;
        redundant := {}; RS := {}; curCard := 0; maxCard := max cardinality of all T_i's;
        for each test case t do
            numUnmarked[t] := number of T_i's containing t;
            numUnmarked'[t] := number of T'_i's containing t;
        endfor
Step 2: loop
            curCard := curCard + 1;
            while ∃ T_i of size curCard s.t. r_i is unmarked do
                list := all tests in T_i's of size curCard s.t. r_i is unmarked;
                nextTest := SelectTest(curCard, list, maxCard);
                RS := RS ∪ {nextTest}; mayReduce := FALSE;
Step 3:         for each T_i containing nextTest s.t. r_i is unmarked do
                    mark r_i;
                    for each test case t in T_i do
                        numUnmarked[t] := numUnmarked[t] − 1;
                        if numUnmarked[t] == 0 and t ∉ RS then
                            redundant := redundant ∪ {t};
                    endfor
                    if cardinality of T_i == maxCard then mayReduce := TRUE;
                endfor
                for each T'_i containing nextTest s.t. r'_i is unmarked do
                    mark r'_i;
                    for each test t in T'_i do
                        numUnmarked'[t] := numUnmarked'[t] − 1;
                endfor
Step 4:         initialize addCoverage[t] := 0 for all tests t;
                for each test t in redundant do addCoverage[t] := numUnmarked'[t];
                while ∃ t in redundant s.t. addCoverage[t] > 0 do
                    toAdd := any test t in redundant with maximum addCoverage[t];
                    RS := RS ∪ {toAdd};
                    for each T'_i containing toAdd s.t. r'_i is unmarked do
                        mark r'_i;
                        for each test t in T'_i do
                            numUnmarked'[t] := numUnmarked'[t] − 1;
                    endfor
                    redundant := redundant − {toAdd};
                    initialize addCoverage[t] := 0 for all tests t;
                    for each test t in redundant do addCoverage[t] := numUnmarked'[t];
                endwhile
                redundant := {};
                if mayReduce then maxCard := max cardinality of T_i's s.t. r_i is unmarked;
            endwhile
        until curCard == maxCard;
end ReduceWithSelectiveRedundancyHGS
```

Figure 2.3: Our main algorithm for reduction with selective redundancy, based on the HGS heuristic.

```
function SelectTest(size, list, maxCard)
    for each test t in list do
        count[t] := number of unmarked T_i's of cardinality size containing t;
    testList := all tests t in list s.t. count[t] is maximum;
    if cardinality of testList == 1 then
        return the test in testList;
    else if size == maxCard then
        return any test in testList;
    else
        return SelectTest(size+1, testList, maxCard);
    endif
end SelectTest
```

Figure 2.4: A helper function from the original HGS algorithm to select the next test case according to the primary requirement set.

tests that covers the most uncovered requirements whose testing sets are of successively higher cardinalities. If the cardinality reaches the maximum cardinality and there are still ties, an arbitrary test case is selected from among the tied tests. The selected test case is then added to the reduced suite.

Step 3: Mark the Newly-Covered Requirements and Update Coverage Information

At this point, we have added a new test case to the reduced suite. This test case covers some set of primary requirements, so any newly-covered primary requirements are marked as covered and the algorithm updates its data structures to reflect the current primary coverage information of the reduced suite. Additionally, if any test case is discovered to become redundant with respect to the primary requirement set in this step, then that test case is added to a set of currently-redundant test cases, which will later be examined and from which redundant test cases may possibly be selected for inclusion in the reduced suite. Similarly for the secondary requirements, the algorithm marks any newly-covered secondary requirements and updates its data structures to reflect the current secondary coverage information of the reduced suite.

Step 4: Select Redundant Test Cases

This step is where our new idea of including selective coverage redundancy takes effect. For each test case currently known to be redundant with respect to the primary criterion, the number of additional secondary requirements that each redundant test case could add to the coverage of the reduced suite is computed. If there exists some redundant test case that adds to the cumulative secondary requirement coverage of the reduced suite, then the test case adding the most secondary requirement coverage is selected (ties are broken arbitrarily). The additional secondary requirement coverage of the remaining redundant test cases is recomputed, and the algorithm continues selecting redundant test cases that add to the cumulative secondary requirement coverage of the reduced suite until either (1) all the redundant test cases have been selected, or (2) no other redundant test case adds to the cumulative secondary requirement coverage. For each redundant test case that is selected, the algorithm marks any newly-covered secondary requirements and updates its secondary requirement coverage data structures. When either case (1) or (2) is reached, the algorithm has completed processing the current set of redundant test cases, and any remaining unselected redundant test cases are thrown away. The algorithm then loops again to consider the next-smallest unmarked primary requirement set, repeating steps 2 – 4 until all primary requirements (and indeed all secondary requirements) are covered by the reduced suite.

A critical aspect of our algorithm is determining the exact point at which a test case becomes redundant with respect to the primary criterion. During the execution of the algorithm, a particular test case will be in one of two possible states: (1) it may be selected in the future according to the primary criterion, or (2) it will definitely never be selected in the future according to the primary criterion. It is not trivial to determine when, during the execution of the original algorithm, a particular test case transitions from state (1) to state (2). By studying the behavior

of the original HGS algorithm, we determined that the only time a test case may possibly be selected according to the primary criterion is when that particular test case exists in some unmarked primary test case set. In other words, as soon as a test case has all of its covered primary requirements marked by the algorithm, then it will be guaranteed that this test case will never be selected by the algorithm with respect to the primary criterion. At this point, the test case becomes redundant with respect to the primary criterion, and becomes a candidate for redundant selection.

We now analyze the worst-case runtime of our algorithm. Our algorithm has the complexity of the original HGS algorithm [16], plus the additional complexity required to account for the secondary coverage requirement during reduction. Let n denote the number of test case sets (requirements) of the primary coverage criterion, and let n' denote the number of test case sets of the secondary coverage criterion. Let MC denote the maximum cardinality among the primary requirement test case sets, and let MC' denote the maximum cardinality among the secondary requirement test case sets. Finally, let nt denote the number of test cases. The behavior of our algorithm is composed of 3 general steps that need to be analyzed: (1) determining the occurrences of test cases in the primary and secondary test case sets; (2) selecting the next test case according to the primary coverage criterion; and (3) selecting the next test case according to the secondary coverage criterion. For the primary criterion, determining the occurrences of test cases in the test case sets takes $O(n * n * MC)$ total time, since this step is performed at most n times by the HGS algorithm (each time a test is selected according to the primary criterion, at least one primary requirement set becomes marked and is not considered again), and each time this step is performed, the algorithm considers at most n sets and examines each element in these sets once (each set is of maximum size MC). For the secondary criterion, determining the occurrences of test cases in the test case sets requires a total of $O(n' * n' * MC')$ time, because there are n' secondary requirement sets, each with maximum cardinality MC', and the algorithm examines occurrences of tests in the secondary test case sets at most n' times (each time a test is selected according to the secondary criterion, at least

one secondary requirement set becomes marked and is not considered again). Thus, the total runtime for determining the occurrences of test cases in the test sets is $O(n * n * MC) + O(n' * n' * MC')$. Next, selecting the next test case according to the primary criterion requires $O(nt * n * MC)$ time, since the HGS algorithm selects at most nt tests, and selecting each test requires an examination of the primary coverage information of tests contained in at most all primary requirement sets. Finally, selecting the next test case according to the secondary criterion requires at most $O(nt * nt)$ time, since at most nt tests will be selected according to the secondary criterion, and selecting each of these tests requires the examination of the secondary requirement coverage of potentially all other test cases (checking the secondary requirement coverage of one test case occurs in constant time since this information is maintained and updated throughout the algorithm; it does not need to be re-computed each time). Therefore, the total runtime of our algorithm is upper-bounded by $O(n * n * MC) + O(n' * n' * MC') + O(nt * n * MC) + O(nt * nt)$, where $O(n * n * MC) + O(nt * n * MC)$ time is required by the original behavior of the HGS algorithm (the same runtime as reported by Harrold, Gupta, and Soffa [16]), and $O(n' * n' * MC') + O(nt * nt)$ additional time is required by the new functionality incorporated into the algorithm by our approach. Notice that if we assume that every test case exercises at least one primary coverage requirement, we have that $nt <= n * MC$ since each test case will be present in at least one primary test case set. Therefore, under this assumption we can collapse the terms $O(nt * n * MC) + O(nt * nt)$ into the single term $O(nt * n * MC)$. Further, if we assume that the secondary criterion is more fine-grained than the primary criterion such that $n <= n'$, and if we assume that $MC <= MC'$, then we may collapse the terms $O(n * n * MC) + O(n' * n' * MC')$ into the single term $O(n' * n' * MC')$. Under these assumptions, the total runtime of our algorithm then becomes $O(n' * n' * MC') + O(nt * n * MC)$, which is the same runtime as for the original HGS algorithm with the exception that the runtime is now bounded by the number and sizes of the *secondary* test case sets, rather than the primary test case sets.

We next work through an example showing the behavior of our algorithm and emphasizing the differences in this behavior from that of the original HGS algorithm.

2.3 An Example

We work through an example using a relatively small, yet meaningful program to illustrate how the behavior of our algorithm differs from the behavior of the original HGS algorithm. A sample program is provided in Figure 2.5, along with a branch-coverage adequate test suite T containing 7 test cases, T_1 through T_7. This program was taken from the Internet [22], and it computes the month and day of Easter for any specified year in the Gregorian Calendar from 1583 to 4099.

The branches covered by each test case are marked with an X in the respective columns in Table 2.1. From this information, we first describe how a traditional minimization algorithm (in particular, the HGS algorithm) will compute a minimized suite with respect to the branch coverage criterion.

Test Case:	B_1^T	B_1^F	B_2^T	B_2^F	B_3^T	B_3^F	B_4^T	B_4^F	B_5^T	B_5^F	B_6^T	B_6^F	B_7^T	B_7^F	B_8^T	B_8^F
T_1:		X		X		X		X		X	X			X	X	
T_2:	X			X	X			X		X	X		X		X	
T_3:		X		X		X		X		X		X		X		X
T_4:	X		X			X		X	X			X		X	X	
T_5:	X		X			X	X			X		X		X	X	
T_6:	X			X		X		X		X		X		X	X	
T_7:		X		X		X		X		X		X		X	X	

Table 2.1: Branch coverage information for test cases in T. Each column except the left-most column describes coverage of branches in the program.

2.3.1 Example Using a Traditional Minimization Algorithm

We use the HGS algorithm described in Appendix A as the traditional minimization algorithm for this example. Initially, all 16 branches are unmarked. The HGS algorithm first considers unmarked branches that are exercised by only 1 test case each. Branches B_3^T, B_4^T, B_5^T, B_7^T, and B_8^F are each only covered by exactly one test case, and the involved test cases are T_2, T_3, T_4, and T_5. The *SelectTest* helper

```
1:      read(year);
2:      a = year / 100;
3:      b = year % 19;
4:      c = ((a − 15) >> 1) + 202 − 11 * b;
B1:     if (a > 26)
5:          c = c − 1;
6:      endif
B2:     if (a > 38)
7:          c = c − 1;
8:      endif
B3:     if (a==21 || a==24 || a==25 || a==33 || a==36 || a==37)
9:          c = c − 1;
10:     endif
11:     c = c % 30;
12:     tA = c + 21;
B4:     if (c == 29)
13:         tA = tA − 1;
14:     endif
B5:     if (c == 28 && b > 10)
15:         tA = tA − 1;
16:     endif
17:     tB = (tA − 19) % 7;
18:     c = (40 − a) & 3;
19:     tC = c;
B6:     if (c > 1)
20:         tC = tC + 1;
21:     endif
B7:     if (c == 3)
22:         tC = tC + 1;
23:     endif
24:     c = year % 100;
25:     tD = (c + (c >> 2)) % 7;
26:     tE = ((20 − tB − tC − tD) % 7) + 1;
27:     day = tA + tE;
B8:     if (day > 31)
28:         day = day − 31;
29:         month = 4;
30:     else
31:         month = 3;
32:     endif
33:     output(month, day);
```

A Branch Coverage Adequate Suite T

T_1: (year = 1865)
T_2: (year = 3769)
T_3: (year = 2005)
T_4: (year = 4004)
T_5: (year = 4031)
T_6: (year = 2777)
T_7: (year = 1601)

Figure 2.5: An example program with a branch coverage adequate test suite T.

function is called to choose a test case to add to the reduced suite from among these four test cases. Since T_2 covers two unmarked branches with test case set cardinality 1, while T_3, T_4, and T_5 each only cover one unmarked branch with test case set cardinality 1, then T_2 is selected because it covers the most unmarked branches with test set cardinality 1. At this point, the following branches are marked because T_2 covers them: B_1^T, B_2^F, B_3^T, B_4^F, B_5^F, B_6^T, B_7^T, B_8^T. Note that at this point, no other test cases have yet become redundant since all other test cases cover at least one branch that is still unmarked.

Next, the algorithm considers branches B_4^T, B_5^T, and B_8^F, since these branches are still unmarked and have test case set cardinality 1. The involved test cases are T_3, T_4, and T_5. All three of these tests tie for each covering 1 unmarked branch with test set cardinality 1. $SelectTest$ therefore makes a recursive call and notices that among these three tied tests, tests T_4 and T_5 each cover 1 unmarked branch with test set cardinality 2, while T_3 covers no unmarked branches with test set cardinality 2. Thus, T_4 and T_5 are still tied. $SelectTest$ calls itself recursively again, but T_4 and T_5 happen to remain tied until the maximum cardinality is reached. At this point, $SelectTest$ makes an arbitrary choice between T_4 and T_5. Let T_4 be the test case selected. Then branches B_2^T, B_3^F, B_5^T, B_6^F, and B_7^F become marked since they were previously unmarked but T_4 exercises them. Note that at this point, test case T_6 becomes redundant because all of its covered branches are now marked.

Now the algorithm considers branches B_4^T and B_8^F since they are only covered by one test case each, namely T_5 and T_3, respectively. Both T_5 and T_3 each only cover one unmarked branch of cardinality 1, so they remain tied and $SelectTest$ calls itself recursively with unmarked branches with test sets of cardinality 2. However, both T_3 and T_5 each cover 0 unmarked branches with test sets of cardinality 2, so another recursive call is made with cardinality 3. Here, T_3 covers unmarked branch B_1^F, which has an associated test set of cardinality 3. However, T_5 covers no unmarked branches with test sets of cardinality 3. Thus, T_3 is selected. This marks branches B_1^F and B_8^F, leaving only branch B_4^T unmarked. At this point, notice that T_1 and T_7 become redundant because they both cover only marked branches.

Finally, test case T_5 is selected because it alone covers the remaining unmarked branch B_4^T. This causes branch B_4^T to be marked, and the algorithm terminates because all branches are now marked. The calculated reduced suite is thus $\{T_2, T_3, T_4, T_5\}$, a reduction of about 43%.

2.3.2 Example Using our New Algorithm

While the example program given in Figure 2.5 is relatively small, there are still quite a few distinct definition-use pairs that are exercised by the test cases in suite T. However, many of these def-use pairs are unimportant in the sense that every test case in T exercises them. Let the set of "unimportant" def-use pairs be called P. These particular def-use pairs do not play any role in the redundancy-selection behavior of our new algorithm, because as soon as at least one test case is selected for inclusion in the reduced suite with respect to the primary minimization criterion, then all of the def-use pairs in P immediately become marked before the algorithm even starts to consider redundant test cases. Therefore, the existence of the pairs in P does not alter in any way the behavior of the new algorithm. Consequently, we save space and simplify our presentation by not listing the many unimportant def-use pairs present in P. Tables 2.2 and 2.3 list the important def-use pairs (which are not present in P) exercised by test cases in T. The def-use pairs covered by each test case are marked with an X in the respective columns in these two tables. We now work through an example using our new algorithm where the primary requirement is branch coverage as given in Table 2.1, and the secondary requirement is def-use pair coverage as given in Tables 2.2 and 2.3.

Initially, all branches and all def-use pairs are considered unmarked, and the set of tests currently known to be redundant with respect to the primary criterion is empty. The algorithm begins just as in the original HGS algorithm, considering uncovered branches with test set cardinalities of 1. The *SelectTest* function first chooses test T_2 to be included in the reduced suite as was done originally. The corresponding branches that are covered by T_2 are marked, and no other test cases

Test Case:	b (3, B_5)	c (4, 11)	c (4, 5)	c (5, 7)	c (5, 9)	c (5, 11)	c (7, 11)	c (9, 11)	tA (12, 13)	tA (12, 15)	tA (12, 17)	tA (13, 17)	tA (15, 17)
T_1:		X									X		
T_2:			X		X			X			X		
T_3:		X									X		
T_4:	X		X	X			X			X			X
T_5:			X	X			X		X			X	
T_6:			X			X					X		
T_7:		X									X		

Table 2.2: Definition-use pair coverage information for test cases in T. Each column except the left-most column describes coverage of def-use pairs in the program.

Test Case:	tA (12, 27)	tA (13, 27)	tA (15, 27)	tC (19, 20)	tC (19, 26)	tC (20, 22)	tC (20, 26)	tC (22, 26)	day (27, 28)	day (27, 33)	day (28, 33)	month (29, 33)	month (31, 33)
T_1:	X			X			X		X		X	X	
T_2:	X			X		X		X	X		X	X	
T_3:	X				X					X			X
T_4:			X		X				X		X	X	
T_5:		X			X				X		X	X	
T_6:	X				X				X		X	X	
T_7:	X				X				X		X	X	

Table 2.3: More definition-use pair coverage information for test cases in T.

are yet identified as redundant with respect to branch coverage, and so the redundant set remains empty. Additionally, the algorithm now marks all of the def-use pairs covered by selected test T_2: $c(4,5)$, $c(5,9)$, $c(9,11)$, $tA(12,17)$, $tA(12,27)$, $tC(19,20)$, $tC(20,22)$, $tC(22,26)$, $day(27,28)$, $day(28,33)$, and $month(29,33)$.

The algorithm continues now as in the original HGS algorithm and the *SelectTest* function makes an arbitrary choice between tests T_4 and T_5. Let T_4 be the next test selected. Then the corresponding unmarked branches that are covered by T_4 are marked, and test case T_6 is identified as redundant with respect to branch coverage. T_6 is therefore added to the redundant set. Next, the unmarked def-use pairs that are covered by T_4 are marked: $b(3,B_5)$, $c(5,7)$, $c(7,11)$, $tA(12,15)$, $tA(15,17)$, $tA(15,27)$, $tC(19,26)$.

At this point, the redundant set is non-empty so the algorithm attempts to selectively add primary requirement coverage redundancy to the reduced suite. Only test T_6 is considered because it is the only test case not yet selected that is currently known to be redundant. Since T_6 covers the unmarked def-use pair $c(5,11)$, it adds to the cumulative def-use coverage of the reduced suite and so it is selected.

The redundant set now becomes empty, def-use pair $c(5, 11)$ is marked, and control returns to the original behavior of the HGS algorithm in which $SelectTest$ next chooses test case T_3 to add to the reduced suite. The corresponding unmarked branches covered by T_3 are marked, and tests T_1 and T_7 are identified as redundant since all of their covered branches have now been marked. Thus, T_1 and T_7 are added to the redundant set. The unmarked def-use pairs covered by T_3 are now marked: $c(4, 11)$, $day(27, 33)$, and $month(31, 33)$.

The algorithm next tries to add selective primary coverage redundancy by checking redundant tests T_1 and T_7. Test T_1 happens to exercise unmarked pair $tC(20, 26)$, while test T_7 does not exercise any unmarked def-use pairs. Hence, T_1 is selected for redundancy but T_7 is not, and the redundant set becomes empty. Def-use pair $tC(20, 26)$ is then marked because it is covered by selected test T_1. Notice that at this point, the only unmarked def-use pairs remaining are $tA(12, 13)$, $tA(13, 17)$, and $tA(13, 27)$.

Control returns next to the original behavior of the HGS algorithm and test T_5 is selected because it alone covers the remaining unmarked branch. This causes all branches to become marked, and the redundant set remains empty because there are no other test cases remaining that become redundant as a result of selecting T_5. The algorithm next marks the remaining three unmarked def-use pairs since T_5 covers them. At this point, all def-use pairs are now marked. The algorithm finally terminates because all branches (and indeed all def-use pairs) are now marked. The calculated reduced suite is thus $\{T_1, T_2, T_3, T_4, T_5, T_6\}$, in which tests T_1 and T_6 were selected by our new algorithm because when they became redundant with respect to branch coverage, they were not redundant with respect to def-use pair coverage.

In the above example, the reduced suite computed by our new algorithm was a superset of the reduced suite computed by the original HGS algorithm. In general, our technique will not always compute a superset of the reduced suite computed by the HGS algorithm due to randomness in breaking ties within the $SelectTest$ function. As a result, it will not necessarily be the case that reduced suites computed by our algorithm will always detect at least as many faults as reduced suites

computed by the original HGS algorithm. However, we expect that in practice, our algorithm will have a strong tendency to compute reduced suites that are slightly larger and better at detecting faults than the reduced suites computed by the original HGS algorithm. Our experimental results (discussed shortly) do indeed confirm this expectation.

2.4 Chapter Summary

This chapter has introduced our general approach to test suite reduction with selective redundancy, and presented a specific implementation of our approach based on the existing HGS heuristic for test suite minimization. An illustrative example was provided in which we executed both the original HGS heuristic and our new algorithm on a sample test suite for a small, yet meaningful program. The next chapter discusses a detailed empirical study comparing our new reduction technique with several existing minimization techniques.

CHAPTER 3

Experimental Study

3.1 Experiment Setup

3.1.1 Subject Programs, Faulty Versions, and Test Case Pools

Our experiments followed an experimental setup similar to that used by Rothermel et al. [34]. We used the well-known Siemens suite of programs described in Table 3.1 as our experimental subjects.

Program Name	Lines of Code	Number of Faulty Versions	Test Case Pool Size	Program Description
tcas	138	41	1608	altitude separation
totinfo	346	23	1052	info accumulator
schedule	299	9	2650	priority scheduler
schedule2	297	10	2710	priority scheduler
printtokens	402	7	4130	lexical analyzer
printtokens2	483	10	4115	lexical analyzer
replace	516	32	5542	pattern substituter

Table 3.1: Siemens suite of experimental subjects.

Each subject program is associated with a test case pool composed of tests that were created for various white and black-box criteria. We do not have the information mapping each test case to the set of requirements for which it was created to cover. Our suite reduction is therefore done with respect to criteria of our choice for which we measure the coverage of each test case.

Each subject program is also associated with a set of faulty versions such that each faulty version is identical to the base program except for a particular seeded error. Most seeded errors involved changing just a single line of code, but some of the faulty versions involved changing several lines. All faulty versions were devised such that they are detectable by at least 3 and at most 350 test cases in the corresponding

test case pool for the given subject program. We examined the types of errors introduced in the faulty versions and identified six distinct categories of seeded errors:

- Changing the operator in an expression
- Changing an operand in an expression
- Changing the value of a constant
- Removing code
- Adding code
- Changing the logical behavior of the code (usually involving a few of the other categories of error types simultaneously in one faulty version)

However, the faults are not evenly distributed among the subject programs in the sense that there is a wide variety in the number of faulty versions for each program, ranging from 7 faulty versions for printtokens to 41 for tcas. Thus, tcas has the most faulty versions available despite the fact that it is the smallest subject program in terms of the number of lines of code. In particular, subject programs schedule, schedule2, printtokens, and printtokens2 have relatively few faulty versions available compared to the other three subject programs. This influences our experimental results (discussed later in this chapter) because it is harder to notice the benefits of our new reduction technique over existing minimization techniques when few available faulty versions limit the amount of fault detection improvement that can be achieved. After all, if there are only 7 faulty versions available and a minimized suite detects 5 of those faults, that leaves only 2 remaining faults that may be used to demonstrate an improvement in fault detection effectiveness. Despite this, it will be shown that our new reduction technique still leads to significant improvements in measured fault detection over existing minimization techniques in our experiments.

All of the programs, faulty versions, and test case pools used in our experiments were assembled by researchers at Siemens Corporation [20]. We obtained the

Siemens programs along with their associated faulty versions and test case pools online [21].

3.1.2 Test Suite Generation and Reduction

As in the work by Rothermel et al. [34], we created each test suite to be edge-coverage adequate. The edge-coverage criterion is also known as the "branch coverage" criterion, and is defined on control-flow graphs; the final result of a (possibly compound) condition is counted as a single edge, and further, entry into each function (besides *main*) is counted as a unique edge.

We selected tests for each suite from the test cases contained in the test pools associated with each subject program. To measure the edge coverage of each test case, we used an instrumented version of each subject program that outputs a unique identifier for each distinct edge executed by the test case. We defined a test suite as being edge-coverage adequate if it achieved the same edge coverage as the entire test case pool for the given subject program. Some particular edges were infeasible with respect to the test pool because they were either unreachable (tcas contains at least one such edge), or they were simply not executed by any test case in the pool. Such edges were left unexercised by our suites if they were not exercised by any test case in the given test case pool.

The specific process we followed for generating edge-coverage adequate suites is as follows: we first randomly selected a randomly-varying number of test cases from the associated test case pool to add to the suite. Then, we added any additional randomly-selected test cases as necessary, so long as they increased the cumulative edge coverage of the suite, until edge-coverage adequacy was achieved. We made sure to select a particular test case from a pool at most once for each generated suite. As in the work by Rothermel et al. [34], the random number of test cases we initially added to each suite varied over sizes ranging from 0 to 0.5 times the number of lines of code in the subject program. We constructed 1000 such edge-coverage adequate test suites for each program. This allowed for a variety of suite sizes in which many suites were highly edge-redundant, and also allowed for the possibility of multiple

test cases within each suite to generate similar execution traces (particularly for the tcas subject program, which contains no loops). Besides this 1000-suite set, we additionally created, for each subject program, four more collections of 1000 suites each. Each collection had suite sizes ranging from 0 to 0.4, 0 to 0.3, 0 to 0.2, and 0 to 0.1 times the number of lines of code in the subject program. Finally, we created one more set of 1000 suites where we simply started with an empty suite and then added tests as necessary (so long as each test increased the cumulative edge coverage of the suite) until edge-coverage adequacy was achieved. The purpose of these five additional 1000-suite collections was to allow us to compare the results due to suite reduction as the average non-reduced suite sizes varied. Altogether, therefore, we created 6000 branch coverage adequate test suites for each program comprising six different suite size ranges. These six suite size ranges correspond to the six rows of experimental data for each subject program listed in Tables 3.2 and 3.4 (described in the next section of this chapter).

For each test case in each suite, we recorded the set of edges covered by that test case. This was accomplished by executing each test case on the edge-coverage instrumented version of the corresponding subject program. The edge-coverage information serves as one of the criteria used in our reduction experiments.

For experiments using our new algorithm, we also required for each test case the information about some set of secondary (finer) requirements covered by that test case. We chose to use the *all-uses* coverage criterion for this purpose. Our primary motivation for this choice of secondary criterion was that all-uses is generally considered to be a stronger (more fine-grained) criterion than edge coverage, and all-uses coverage is also easily measured with an existing tool to which we have access. Of course, other choices for secondary criteria are available, especially choices that also incorporate black-box requirements derived from the specifications of the subject programs. We would expect such stronger choices for the secondary criterion to lead to improved fault detection retention. However, besides retaining fault detection, another one of our goals is to not severely compromise suite size reduction. A secondary criterion that is too strong may possibly lead to significantly less suite size

reduction. We felt that as a first step for exploration, using all-uses coverage as the secondary criterion would lead to a good compromise between achieving improved fault detection retention without severely impacting suite size reduction.

The ATAC tool [19] was used to measure the all-uses coverage of each test case in each suite; this tool is used to automatically generate an instrumented version of each subject program that can be used to measure the all-uses coverage of a particular test case. The all-uses coverage criterion is the same as the all-definition-use pair coverage criterion, with one difference: for predicate uses, a third parameter (besides the definition and use) describes the destination basic block next executed as a result of the predicate's value. Thus, there may be *two* uses of the same variable in a given predicate: one for the predicate evaluating to true, and one for the predicate evaluating to false. Thus, the all-uses criterion we used is a finer-grained criterion than the all-definition-use pair coverage criterion.

The focus of our experiments is to compare the *reduction* results using our new technique with the *minimization* results using an existing technique. For the existing technique, we chose to use the HGS algorithm for test suite minimization [16], which was studied by Rothermel, Harrold, Ostrin, and Hong [34]. Rothermel et al. empirically evaluated the HGS algorithm with edge coverage as the minimization criterion; we shall refer to this particular technique as the "RHOH technique", in reference to the initials of the authors of this empirical study.

In Chapter 2, we showed how our approach for test suite reduction could be implemented by modifying the HGS algorithm for test suite minimization. The reason is now clear: this particular implementation of our new technique allows us to better compare the results of our new technique with the RHOH technique because both techniques are based on the HGS algorithm.

For experiments using our new technique, we chose to use edge coverage as the primary criterion for minimization, and all-uses coverage as the secondary criterion. We shall refer to this new technique as the "RSR technique" (which stands for "Reduction with Selective Redundancy").

We implemented both the RHOH technique and the RSR technique in Java. In

order to compare the results using our new RSR technique with the existing RHOH technique, we conducted the following two experiments:

- **Experiment RHOH**: Minimize each suite using the RHOH technique used by Rothermel et al. [34]. This experiment is meant to reproduce the experimental results reported by Rothermel et al.
- **Experiment RSR**: Reduce each suite using our new RSR technique.

Additionally, to compare the results of our new RSR technique against those of an existing minimization technique with respect to other minimization criteria, we chose to also conduct the following two experiments:

- **Experiment U**: Minimize each suite as in Experiment RHOH, except now minimize with respect to the all-uses coverage criterion.
- **Experiment E+U**: Minimize each suite as in Experiment RHOH, except now minimize with respect to the union of edge coverage and all-uses coverage. This effectively minimizes with respect to both the primary and secondary criteria simultaneously.

Further, to show that our new RSR technique selects the additional primary coverage-redundant test cases that are good at detecting new faults, we conducted the following experiment:

- **Experiment RAND**: Minimize each suite as in Experiment RHOH, with one difference: when a minimized suite is computed by the RHOH technique, we then check whether the corresponding reduced suite computed by RSR is larger or not. If so, we *randomly* add additional tests to the RHOH-minimized suite until the size matches that of the corresponding RSR-reduced suite. Thus, this experiment computes minimized suites of the same sizes as in Experiment RSR, but the additional tests selected here are selected randomly, rather than by analysis of the secondary coverage information as is done in Experiment

RSR. To not overwhelm the reader with another large data table, here we only report these experimental results for suite size range 0 – 0.5 of each subject program.

For each experiment, we recorded the following information about each suite:

- The number of test cases in the original suite ($|T|$)
- The number of test cases in the minimized/reduced suite ($|T_{red}|$)
- The number of distinct faults detected by the original suite ($|F|$)
- The number of distinct faults detected by the minimized/reduced suite ($|F_{red}|$)

Given the above information, we also computed the following information for each suite due to minimization:

- The percentage suite size reduction (% Size Reduction). This is the difference in suite size due to minimization, divided by the original suite size. It is computed as follows:
$$\frac{(|T| - |T_{red}|)}{|T|} * 100$$
- The percentage fault detection effectiveness loss (% Fault Loss). This is the difference in the number of faults detected by the original and minimized/reduced suites, divided by the number of faults detected by the original suite. It is computed as follows:
$$\frac{(|F| - |F_{red}|)}{|F|} * 100$$
- For the suites in suite size range 0 – 0.5 such that the RSR technique computes a larger reduced suite than the corresponding RHOH-minimized suite, we also computed the additional-faults-to-additional-tests ratio. This ratio is a measure of, for each additional test case selected into an RSR-reduced suite than into the corresponding RHOH-minimized suite, the number of additional faults detected by the RSR-reduced suite. The ratio is therefore a

computation based upon the relationship between an RSR-reduced suite and its corresponding RHOH-minimized suite, and is computed as follows:

$$\frac{(|F_{red}|_{RSR} - |F_{red}|_{RHOH})}{(|T_{red}|_{RSR} - |T_{red}|_{RHOH})}$$

We next present the results of our empirical study, and we provide analysis and discussion of these results.

3.2 Experimental Results, Analysis, and Discussion

The results for Experiment RHOH and Experiment RSR are shown respectively in the columns labeled "RHOH" and "RSR" in Table 3.2. The table caption describes the information provided in each column of the table. The values reported in each row of the table are the averages computed across all 1000 suites for the given suite size range of a subject program. To save space, the program names schedule, schedule2, printtokens, and printtokens2 have been abbreviated.

We make the following observations from Table 3.2:

- As the average original suite sizes increase (from suite size range 0 through 0 – 0.5), the sizes of the RHOH-minimized suites tend to decrease slightly. This is because larger suites have a wider variety of test cases to choose from when minimizing, so better initial choices from larger original suites leads to fewer total test cases required in a minimized suite to achieve the same coverage as the original suite.

- As the average original suite sizes increase, the sizes of the RSR-reduced suites tend to increase. This is due to the redundant selection of certain tests using the RSR technique; when original suite sizes are larger, slightly more test cases are marked for selective redundancy using the RSR technique. This is likely because larger suites tend to execute more distinct all-uses than smaller suites.

- While the number of faults detected by the RHOH-minimized suites do not have a strong tendency to either increase or decrease as the original suite

Prog/Suite Size Range	$\|T\|$	$\|F\|$	$\|T_{red}\|$		$\|F_{red}\|$		% Size Reduction		% Fault Loss	
			RHOH	RSR	RHOH	RSR	RHOH	RSR	RHOH	RSR
tcas 0	5.71	7.47	5.00	5.16	6.78	6.92	11.34	8.87	8.18	6.39
tcas 0-0.1	9.56	9.15	5.00	6.20	6.84	7.46	41.60	30.18	22.35	16.53
tcas 0-0.2	15.20	11.73	5.00	6.94	6.73	7.83	57.66	45.54	37.07	28.56
tcas 0-0.3	21.39	14.02	5.00	7.32	6.85	8.25	66.34	55.23	44.60	35.62
tcas 0-0.4	29.07	16.29	5.00	7.71	6.80	8.56	73.09	62.95	52.09	41.79
tcas 0-0.5	35.63	17.76	5.00	7.91	6.67	8.59	76.77	67.57	56.23	46.13
totinfo 0	7.30	12.49	5.18	5.47	11.44	11.87	26.66	23.06	7.91	4.77
totinfo 0-0.1	18.68	14.62	5.11	5.96	11.44	12.63	64.58	60.04	20.31	12.85
totinfo 0-0.2	35.61	16.73	5.05	6.29	11.43	13.11	77.47	73.54	30.01	20.48
totinfo 0-0.3	52.07	17.70	5.04	6.44	11.36	13.19	82.60	79.21	34.05	24.05
totinfo 0-0.4	69.62	18.55	5.04	6.46	11.42	13.27	86.48	83.82	36.92	27.07
totinfo 0-0.5	87.73	19.16	5.02	6.46	11.34	13.15	88.96	86.62	39.42	30.15
sched 0	7.31	3.38	5.11	5.61	2.88	3.09	28.70	21.99	13.57	7.76
sched 0-0.1	18.44	4.58	4.99	6.03	2.89	3.25	66.77	60.77	35.05	27.16
sched 0-0.2	32.09	5.18	4.98	6.30	2.81	3.23	77.29	72.57	44.63	36.79
sched 0-0.3	47.91	5.61	4.86	6.45	2.91	3.33	83.29	79.12	47.39	39.81
sched 0-0.4	58.83	5.77	4.78	6.49	2.87	3.37	85.03	81.28	49.35	40.62
sched 0-0.5	74.94	5.96	4.74	6.61	2.88	3.27	87.91	84.51	51.18	44.46
sched2 0	8.01	2.21	5.37	5.79	1.89	1.98	31.51	26.38	12.43	8.46
sched2 0-0.1	18.61	2.57	5.18	6.12	1.95	2.08	66.17	60.80	20.49	15.99
sched2 0-0.2	33.19	3.23	5.04	6.23	1.90	2.13	77.67	73.53	36.80	30.37
sched2 0-0.3	47.44	3.77	4.94	6.38	1.89	2.15	83.29	79.74	45.07	38.27
sched2 0-0.4	61.60	4.35	4.82	6.54	2.09	2.42	86.16	82.80	47.26	40.05
sched2 0-0.5	76.34	4.73	4.74	6.71	2.02	2.44	88.45	85.36	51.87	43.15
printtok 0	15.76	3.38	7.12	7.63	2.90	3.03	53.69	50.39	12.36	9.19
printtok 0-0.1	27.64	3.64	7.11	7.76	2.85	3.06	71.14	68.62	19.25	14.21
printtok 0-0.2	46.03	3.96	6.93	7.75	2.87	3.11	80.26	78.26	25.00	19.53
printtok 0-0.3	63.84	4.28	6.81	7.76	2.93	3.15	83.92	82.16	28.66	24.07
printtok 0-0.4	83.44	4.54	6.70	7.80	2.89	3.19	86.89	85.27	33.40	27.36
printtok 0-0.5	101.87	4.75	6.58	7.73	2.89	3.22	88.77	87.38	36.02	29.46
printtok2 0	11.77	7.36	7.16	9.04	7.05	7.25	37.35	21.96	4.04	1.45
printtok2 0-0.1	27.56	7.80	6.78	11.79	7.08	7.49	68.39	50.02	8.90	3.82
printtok2 0-0.2	49.74	8.17	6.25	12.76	6.99	7.63	79.76	65.06	13.94	6.34
printtok2 0-0.3	75.01	8.45	5.85	13.22	7.13	7.86	86.03	73.68	15.34	6.78
printtok2 0-0.4	100.34	8.58	5.61	13.41	7.17	7.89	88.98	78.57	16.18	7.82
printtok2 0-0.5	121.73	8.60	5.49	13.51	7.13	7.94	90.19	80.71	16.72	7.52
replace 0	18.63	11.13	11.93	14.92	8.82	10.42	35.34	19.43	19.72	6.20
replace 0-0.1	34.59	14.10	11.75	17.49	9.03	12.00	61.18	44.46	33.98	13.97
replace 0-0.2	56.67	16.80	11.33	19.13	8.85	13.12	73.20	58.45	44.75	20.49
replace 0-0.3	82.49	19.01	11.09	20.54	8.83	13.82	79.77	66.84	50.93	25.54
replace 0-0.4	105.06	19.96	10.90	21.27	8.77	14.11	82.35	70.63	53.04	27.34
replace 0-0.5	134.59	21.43	10.66	22.39	8.77	14.53	86.70	76.10	56.77	30.38

Table 3.2: Experimental results for Experiment RHOH and Experiment RSR showing, for each suite size range for each subject program: the average original suite size ($|T|$), the average number of faults detected by the original suite ($|F|$), the average minimized/reduced suite size ($|T_{red}|$), the average number of faults detected by the minimized/reduced suite ($|F_{red}|$), the average percentage suite size reduction (% Size Reduction), and the average percentage fault detection loss (% Fault Loss).

sizes vary, the number of faults detected by the RSR-reduced suites tends to increase as the original suite sizes increase. This is due to larger original suites leading to larger RSR-reduced suites, which are likely to detect more distinct faults than the other RSR-reduced suites that are smaller in size.

- Both the RSR and RHOH suites lead to increased percentage suite size reduction as the original suite sizes increase. This is simply due to larger original suites undergoing greater percentage size reduction than smaller original suites because of more coverage redundancy being removed.

- In all cases, RHOH suites achieve greater percentage suite size reduction on average than the corresponding RSR-reduced suites. This is expected since RSR includes selective redundancy in the reduced suites while RHOH does not.

- Both the RSR and RHOH suites lead to increased percentage fault detection loss as the original suite sizes increase. This is highly correlated with the percentage suite size reduction; the suites achieving the most suite size reduction also strongly tend to experience the most fault detection loss.

- In all cases, RHOH suites experience greater percentage fault detection loss on average than the corresponding RSR-reduced suites. This is expected since RSR-reduced suites are generally larger than the RHOH-minimized counterparts, and we expect that the RSR technique does a good job of selecting additional tests that are likely to detect new faults. This will be analyzed in greater detail later in this chapter.

In general, Table 3.2 shows that the RSR technique leads to less suite size reduction, but greater fault detection retention in general, than the RHOH technique. Further, both the RSR and RHOH techniques achieve considerable suite size reduction in all cases. Given the significant improvement in average percentage fault detection retention for RSR over RHOH, there seems to be a potential benefit of including selective coverage redundancy during test suite reduction.

Since Experiment RHOH is meant to reproduce the experimental results of Rothermel et al., it is also interesting to look at how well our experimental results compare to those reported by Rothermel et al [34]. These authors do not list the average fault detection loss values across all 1000 suites for each subject program. Instead, they illustrate their findings on a per-test-suite basis using scatter plots. However, in a longer study [35], the authors do list the average percentage fault detection loss values across all 1000 suites for each subject program (they only conducted experiments for suite size range 0 – 0.5). These reported values were obtained by following the same experimental setup as in our Experiment RHOH for suite size range 0 – 0.5 (minimizing edge-coverage adequate suites using the HGS algorithm with respect to the edge coverage criterion). Table 3.3 shows the results reported by Rothermel et al. [35] and the corresponding results obtained in our own Experiment RHOH (for suite size range 0 – 0.5).

Program	% Fault Loss (Rothermel et al. [35])	% Fault Loss (Our Experiment RHOH)
tcas	60.90	56.23
totinfo	39.20	39.42
schedule	51.10	51.18
schedule2	56.70	51.87
printtokens	40.80	36.02
printtokens2	21.30	16.72
replace	57.20	56.77

Table 3.3: Experimental results for average percentage fault loss reported by Rothermel et al. [35] versus the results obtained in our own reproduction Experiment RHOH (suite size range 0 – 0.5).

From this table we see that our results are very close to the results reported by Rothermel et al. for subject programs totinfo, schedule, and replace. The remaining programs tcas, schedule2, printtokens, and printtokens2 show our results understating the average percentage fault detection loss by about 4% or 5% as compared to the average fault losses reported by Rothermel et al. Most likely these differences are due simply to the fact that we're using different suites than those used by Rothermel et al. [35], despite the fact that we both used the same method of generating suites randomly from the test case pools. However, since our RHOH results are generally understating the fault losses as compared to the results reported by Rothermel et

al., then RHOH appears to perform generally worse in the Rothermel work than it does in our own experiments. Therefore, the benefit of RSR over our own RHOH results is even more pronounced in general when considering that our RHOH results show RHOH to generally be better than it appears in the Rothermel work!

Table 3.4 shows the results for Experiment U and Experiment E+U. These results are shown respectively in the columns labeled "U" and "E+U" in the table.

Prog/Suite Size Range	$\|T\|$	$\|F\|$	$\|T_{red}\|$		$\|F_{red}\|$		% Size Reduction		% Fault Loss	
			U	E+U	U	E+U	U	E+U	U	E+U
tcas 0	5.71	7.47	5.02	5.02	6.80	6.81	11.02	11.02	7.87	7.83
tcas 0-0.1	9.56	9.15	5.68	5.68	7.02	6.97	35.22	35.22	20.53	20.82
tcas 0-0.2	15.20	11.73	6.08	6.08	7.07	7.00	50.90	50.90	34.21	34.97
tcas 0-0.3	21.39	14.02	6.27	6.27	7.17	7.11	60.34	60.34	42.60	42.96
tcas 0-0.4	29.07	16.29	6.48	6.48	7.24	7.21	67.47	67.47	49.50	49.53
tcas 0-0.5	35.63	17.76	6.56	6.56	7.05	7.05	71.74	71.74	54.19	54.06
totinfo 0	7.30	12.49	5.34	5.34	11.83	11.83	24.70	24.70	5.08	5.08
totinfo 0-0.1	18.68	14.62	5.30	5.30	12.47	12.43	63.26	63.26	13.85	14.13
totinfo 0-0.2	35.61	16.73	5.19	5.19	12.84	12.79	76.71	76.71	22.03	22.35
totinfo 0-0.3	52.07	17.70	5.15	5.16	13.03	13.01	82.00	81.99	25.02	25.09
totinfo 0-0.4	69.62	18.55	5.12	5.12	13.16	13.20	86.15	86.15	27.78	27.51
totinfo 0-0.5	87.73	19.16	5.09	5.09	13.16	13.18	88.68	88.67	30.21	30.04
sched 0	7.31	3.38	5.36	5.54	2.90	3.09	25.30	22.90	13.53	8.02
sched 0-0.1	18.44	4.58	5.47	5.63	2.98	3.21	63.81	62.80	33.08	28.21
sched 0-0.2	32.09	5.18	5.54	5.74	2.83	3.16	75.20	74.39	44.15	38.22
sched 0-0.3	47.91	5.61	5.55	5.83	2.80	3.21	81.40	80.66	49.01	42.01
sched 0-0.4	58.83	5.77	5.52	5.83	2.75	3.24	83.41	82.65	51.08	42.88
sched 0-0.5	74.94	5.96	5.56	5.88	2.67	3.19	86.35	85.79	54.31	45.93
sched2 0	8.01	2.21	4.79	5.73	1.97	1.98	39.13	27.04	8.95	8.65
sched2 0-0.1	18.61	2.57	4.83	5.77	2.04	2.05	68.78	62.62	16.93	16.99
sched2 0-0.2	33.19	3.23	4.80	5.75	2.06	2.05	79.15	75.02	31.78	32.17
sched2 0-0.3	47.44	3.77	4.81	5.77	2.10	2.08	84.22	81.11	39.30	39.55
sched2 0-0.4	61.60	4.35	4.88	5.84	2.25	2.28	86.66	84.04	43.60	43.14
sched2 0-0.5	76.34	4.73	4.89	5.86	2.28	2.25	88.84	86.60	46.25	46.67
printtok 0	15.76	3.38	7.44	7.51	2.98	2.99	51.61	51.15	10.32	9.90
printtok 0-0.1	27.64	3.64	7.49	7.56	3.04	3.05	69.62	69.34	14.68	14.50
printtok 0-0.2	46.03	3.96	7.38	7.44	3.05	3.06	79.12	78.95	21.04	20.62
printtok 0-0.3	63.84	4.28	7.29	7.36	3.09	3.09	82.95	82.77	25.12	25.16
printtok 0-0.4	83.44	4.54	7.26	7.32	3.11	3.12	86.01	85.89	28.97	28.65
printtok 0-0.5	101.87	4.75	7.17	7.23	3.15	3.15	88.03	87.91	30.88	30.73
printtok2 0	11.77	7.36	8.78	8.78	7.24	7.25	23.96	23.96	1.51	1.49
printtok2 0-0.1	27.56	7.80	10.05	10.05	7.45	7.45	55.55	55.54	4.23	4.24
printtok2 0-0.2	49.74	8.17	10.06	10.05	7.63	7.63	70.35	70.35	6.31	6.38
printtok2 0-0.3	75.01	8.45	9.92	9.92	7.78	7.79	78.56	78.56	7.66	7.58
printtok2 0-0.4	100.34	8.58	9.90	9.90	7.86	7.84	82.59	82.59	8.18	8.40
printtok2 0-0.5	121.73	8.60	9.88	9.89	7.84	7.85	84.43	84.43	8.62	8.52
replace 0	18.63	11.13	14.53	14.53	10.33	10.32	21.50	21.50	6.92	7.11
replace 0-0.1	34.59	14.10	15.86	15.86	11.59	11.61	48.83	48.83	16.73	16.61
replace 0-0.2	56.67	16.80	16.31	16.31	12.50	12.52	63.15	63.14	24.00	23.90
replace 0-0.3	82.49	19.01	16.70	16.70	13.06	12.98	71.45	71.45	29.25	29.60
replace 0-0.4	105.06	19.96	16.79	16.80	13.33	13.28	74.96	74.96	31.00	31.27
replace 0-0.5	134.59	21.43	16.94	16.95	13.49	13.52	80.49	80.48	35.09	34.97

Table 3.4: Experimental results for Experiment U and Experiment E+U. The table is organized the same way as Table 3.2.

We now make the following observations from Table 3.4:

- U-minimized suites generally achieve the same or slightly greater suite size reduction than the corresponding E+U-minimized suites on average. This is expected because covering the union of all-edges and all-uses should generally require at least as many test cases as those required to cover only the all-uses. Hence, we expect the E+U-minimized suites to be the same size or larger than the U-minimized suites on average.

- In all subject programs except for schedule, the U-minimized suites experience nearly the same fault detection loss as the E+U-minimized suites on average. Program schedule is the only notable exception, in which the E+U-minimized suites achieve significantly less fault detection loss than the U-minimized suites. However, the E+U-minimized suites for schedule still result in slightly more fault detection loss on average than the corresponding RSR-reduced suites from Table 3.2.

Given the results from Tables 3.2 and 3.4, we can clearly see that RHOH achieves greater percentage suite size reduction on average than that achieved by RSR. For instance, in the largest suite size range for tcas, printtokens2, and replace, RHOH achieves about 10% more size reduction than RSR. The U and E+U techniques are very similar to each other in terms of their ability to achieve suite size reduction. In general, they tend to achieve a "middle-ground" between the RHOH and RSR techniques when reducing suite sizes. Thus, in terms of achieving the most suite size reduction, RHOH is generally the best and RSR is generally the worst. However, keep in mind that all four techniques still achieve high suite size reduction, relative to the large sizes of the original suites. Thus, even though RHOH may be better in terms of achieving suite size reduction than RSR, it is still true that RSR by itself is still quite good in this regard.

Looking in terms of percentage fault detection loss, we can clearly see that RHOH allows for greater percentage fault detection loss of suites on average than that allowed by RSR. For instance, in the largest suite size range, RSR improves

upon the fault detection retention of RHOH by about 10% with tcas, and by about 26% with replace. Again, the U and E+U techniques are generally similar to each other in terms of their ability to retain the fault detection capabilities of suites, and they generally achieve a "middle-ground" between RHOH and RSR in terms of fault detection retention. Therefore, RHOH generally achieves the most suite size reduction at the expense of yielding the most fault detection loss, while RSR generally achieves the least suite size reduction with the benefit of yielding the least fault detection loss. Considering that even RSR is still able to achieve relatively high suite size reduction, the benefit of RSR in retaining more fault detection effectiveness in our experiments is clear.

The reason our RSR technique is still able to achieve relatively high suite size reduction in our experiments is that Experiment RSR only allows for one level of selective redundancy: something redundant with respect to the primary criterion but not redundant with respect to the secondary criterion is selected. Since many of the original suites used in our experiments were highly edge-redundant and even all-uses redundant (some suites contained over 200 test cases each!), even suites reduced using RSR often did not contain anywhere near the high levels of edge and all-uses coverage redundancy contained in some of the original, non-reduced suites.

Given that RSR achieves the greatest fault detection retention at the cost of the least suite size reduction, a reasonable question is whether or not the increased fault detection retention of RSR is due *merely* to the fact that the RSR-reduced suites are larger than the other minimized suites. It turns out this is not the case, as indicated by the results of Experiment RAND. The results for Experiment RAND are reported in Table 3.5, compared against the corresponding RSR results taken from Table 3.2. In this table, there is only one row for each subject program because we only report the results for suite size range $0 - 0.5$ of each subject program. Further, the table only lists columns related to fault detection since the size reduction values exactly match those reported by the RSR technique in Table 3.2.

As indicated by the comparison of results depicted in Table 3.5, the RSR technique does a good job on average of selecting just those additional tests that are

Program	$\lvert F_{red} \rvert$		% Fault Loss	
	RAND	RSR	RAND	RSR
tcas 0-0.5	8.45	8.59	46.55	46.13
totinfo 0-0.5	11.96	13.15	36.32	30.15
schedule 0-0.5	3.26	3.27	44.60	44.46
schedule2 0-0.5	2.15	2.44	49.02	43.15
printtokens 0-0.5	2.94	3.22	35.12	29.46
printtokens2 0-0.5	7.58	7.94	11.62	7.52
replace 0-0.5	12.13	14.53	41.66	30.38

Table 3.5: Experimental results for Experiment RAND compared against the corresponding results for Experiment RSR showing the average number of faults detected by the reduced suites ($|F_{red}|$) and the average percentage fault detection loss due to reduction (% Fault Loss). The average reduced suite sizes match those of Experiment RSR listed in Table 3.2.

likely to expose new faults in the software. We thus make the following observation from Table 3.5:

- In all cases, the average number of faults detected by the RAND-reduced suites is less than the average number of faults detected by the corresponding RSR-reduced suites. Accordingly, the average percentage fault detection loss of the RAND-reduced suites is always more than the average fault detection loss of the RSR-reduced suites. It turns out that for programs tcas and schedule, the RSR suites are only slightly better on average than the same-sized RAND-reduced suites in terms of retaining fault detection. However, for the remaining five subject programs, the RSR suites achieve between about 4% and 11% less fault detection loss than the same-sized RAND-reduced counterparts. Thus, there is a strong tendency for RSR-reduced suites to do a better job of retaining fault detection effectiveness than other suites of the same size where the additional selected tests are selected randomly (rather than by considering a secondary coverage criterion as is done in RSR).

Intuitively, the fact that the RSR suites have a strong tendency to retain more fault detection effectiveness than the same-sized RAND suites is expected because the additional tests selected by RSR are chosen only if they exercise new uses not already exercised by the current tests that are selected for the reduced suite. Since every additional test case selected by RSR executes a unique use, it must be true that

each additional test selected exercises a unique path through the software that is not already exercised by another test selected for the reduced suite (a new use cannot be covered unless a new path through the software is taken). Under the RAND technique, there is no such guarantee that these "new situations" are exercised by the additional tests selected randomly. To better analyze this, we have examined each suite reduced by RAND for each subject program to find the number of reduced suites such that each test case in the suite covers a unique path. These results give us an idea of how often RAND is selecting additional tests that happen to cover the same paths as those already selected in a reduced suite. The results are given in Table 3.6.

Program	Num. of suites with all tests covering unique paths
tcas	128
totinfo	982
schedule	994
schedule2	996
printtokens	1000
printtokens2	966
replace	976

Table 3.6: The number of RAND-reduced suites (out of 1000 total suites for suite size range 0 – 0.5) of each subject program in which every test case in a suite covers a unique path.

Program tcas has so few suites listed in Table 3.6 because tcas is the only subject program that contains no loops. Therefore, there is a small, finite number of paths through tcas, and it is relatively easy to select an additional random test case that covers a redundant path through tcas. Surprisingly, for the remaining six subject programs, virtually all of the RAND-reduced test suites are such that every test case in a suite covers a unique path through the program! Despite this, the RSR-reduced suites *still* have a strong tendency of retaining more fault detection effectiveness than the RAND-reduced suites. This suggests that merely exercising more unique paths may not be a strong indicator that fault detection capability will significantly increase. Clearly, RSR seems to be doing a better job in general of selecting the *particular* tests covering unique paths that improve fault detection effectiveness. This is most likely due to the fact that RSR is taking data-flow into account when it

selects additional tests that cover unique uses in the software. RAND does not take data-flow into account, and despite the fact that most RAND-reduced suites have all tests covering unique paths, fewer of those additional tests that exercise unique paths are likely exercising new uses in the software. This can be intuitively reasoned because if a loop executes 85 times on one test case, and 86 times on another test case, then clearly these two tests exercise unique paths. However, a loop executing 86 times instead of 85 times will probably not be covering a new def-use pair in the loop that was not covered by the 85-iteration loop. Thus, unique paths do not necessarily imply unique data-flow (though unique data-flow does imply unique paths), and the unique data-flow encouraged by RSR is likely working more towards improving the fault detection effectiveness of suites than merely exercising unique paths. Hence, there are clearly benefits of RSR over RAND in terms of improving fault detection retention.

To further aid in our experimental analysis, we present Tables 3.7 through 3.13, one table for each subject program. To save space, data is presented for suite size range 0 – 0.5 only. Each table represents a matrix in which each of the 1000 suites for the given subject program are plotted, comparing the RSR-reduced suites to their corresponding RHOH-minimized counterparts. Let $|T_{red}|$ refer to the size of a reduced suite, and let $|F_{red}|$ refer to the number of faults detected by a reduced suite. The first column of each row in a matrix (where the numbers are preceded by a "+" or "-") represents the number of additional test cases in the RSR-reduced suite over the corresponding RHOH-minimized suite:

$$|T|_{add} = (|T_{red}|_{RSR} - |T_{red}|_{RHOH})$$

The first row of each column (where the numbers are preceded by a "+" or "-") represents the number of additional faults detected by the RSR-reduced suite over the RHOH-minimized counterpart:

$$|F|_{add} = (|F_{red}|_{RSR} - |F_{red}|_{RHOH})$$

Each row/column entry in a matrix is a test suite count. Entry x at row/column $(i,\ j)$ in a table indicates that among the 1000 suites for suite size range 0 – 0.5

of the the given subject program, there were x suites such that the RSR-reduced suites contained i more test cases and detected j more faults than the corresponding RHOH-minimized suites. For example, in Table 3.7 for tcas, there were 37 of the 1000 suites such that the RSR-reduced suite was 3 test cases larger than the corresponding RHOH-minimized suite, and the RSR-reduced suite detected 3 more distinct faults than the corresponding RHOH-minimized suite. Column and row test suite sums are provided in the rows and columns labeled with a "Σ". For example, in the bottom row of Table 3.7, there were a total of 172, 212, and 616 test suites such that fault detection decreased, remained the same, and increased, respectively.

We make the following observations from Tables 3.7 through 3.13:

- For all subject programs except printtokens2 and replace, the RSR-reduced suites are only between 4 and 6 test cases larger than their RHOH-minimized counterparts. For printtokens2, most RSR suites are between 6 and 10 test cases larger than the corresponding RHOH suites. For replace, most RSR suites are between 10 and 15 test cases larger. Considering that the average sizes of the non-reduced suites range from about 35 tests for tcas suites to 135 tests for replace suites, we see that RSR generally selects relatively few additional test cases beyond those selected by RHOH.

- In all subject programs, there are far more suites with increased fault detection than decreased fault detection, when going from the RHOH technique to the RSR technique. This shows that the RSR technique, while not always improving the fault detection of suites, has a much greater likelihood of increasing fault detection effectiveness than of decreasing it.

- Programs schedule, schedule2, printtokens, and printtokens2 have a relatively larger number of suites in which the fault detection effectiveness remained unchanged in going from RHOH to RSR. This is most likely due to the fact that these four programs have the fewest number of faulty versions available, so there are fewer opportunities for detecting new distinct faults with these four subject programs.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-9	-7	-6	-5	-4	-3	-2	-1	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	Σ
+0	0	0	0	0	0	1	4	4	52	3	5	1	0	1	0	0	0	0	0	0	0	0	0	71
+1	0	0	0	1	2	1	4	6	44	8	9	6	1	1	2	0	0	0	0	0	0	0	0	85
+2	0	0	0	2	3	0	9	14	43	24	22	20	12	4	2	3	2	2	0	0	0	0	0	162
+3	1	0	3	2	5	9	19	27	39	43	38	37	40	17	7	12	3	5	3	0	0	1	0	311
+4	0	3	1	5	3	6	9	20	31	28	36	32	32	20	22	23	13	6	3	1	1	2	0	297
+5	0	0	0	0	2	3	0	2	2	10	8	9	7	6	5	2	4	5	1	1	1	0	1	69
+6	0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0	0	5
Σ	1	3	4	10	15	21	45	73	212	116	118	105	93	49	38	40	23	18	8	2	2	3	1	1000
Σ	**172**								**212**	**616**														

Table 3.7: RSR vs RHOH additional tests/additional faults matrix: **tcas**. This matrix shows the number of test suites for suite size range 0 – 0.5 such that the RSR-reduced suite contained $|T|_{add}$ additional tests and detected $|F|_{add}$ additional faults than the corresponding RHOH-minimized suite. "Σ" represents a sum of test suite counts.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-9	-8	-7	-6	-5	-4	-3	-2	-1	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	Σ
-1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1
+0	0	0	0	0	0	1	2	4	9	64	11	7	5	3	0	2	2	1	1	0	0	112
+1	1	1	3	1	9	5	6	16	21	81	113	65	37	29	16	14	21	16	6	2	0	463
+2	0	1	1	2	10	10	5	15	14	35	51	44	36	23	17	11	14	15	10	0	2	316
+3	0	0	0	1	1	5	1	3	2	8	11	14	7	6	13	6	1	9	2	2	0	92
+4	0	0	0	1	0	0	0	0	0	0	5	3	2	1	0	2	0	1	0	0	0	15
+5	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1
Σ	1	2	4	5	20	21	14	38	46	188	192	134	87	62	46	35	38	42	19	4	2	1000
Σ	**151**									**188**	**661**											

Table 3.8: RSR vs RHOH additional tests/additional faults matrix: **totinfo**. This table is organized the same way as Table 3.7.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-3	-2	-1	+0	+1	+2	+3	+4	Σ
+0	0	1	5	46	4	1	0	0	57
+1	1	6	18	165	76	20	6	0	292
+2	2	8	37	207	106	54	14	1	429
+3	0	6	15	82	43	20	7	1	174
+4	0	3	7	16	9	3	2	0	40
+5	0	1	0	2	4	1	0	0	8
Σ	3	25	82	518	242	99	29	2	1000
Σ	**110**			**518**	**372**				

Table 3.9: RSR vs RHOH additional tests/additional faults matrix: **schedule**. This table is organized the same way as Table 3.7.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-3	-2	-1	+0	+1	+2	+3	+4	+5	Σ
+0	0	3	7	110	2	2	0	0	0	124
+1	0	3	3	206	23	18	1	0	0	254
+2	1	2	9	187	48	35	4	2	1	289
+3	0	2	7	114	46	35	15	1	0	220
+4	0	1	5	39	22	17	6	2	0	92
+5	0	1	0	8	1	4	1	2	0	17
+6	0	0	0	4	0	0	0	0	0	4
Σ	1	12	31	668	142	111	27	7	1	1000
Σ	**44**			**668**	**288**					

Table 3.10: RSR vs RHOH additional tests/additional faults matrix: **schedule2**. This table is organized the same way as Table 3.7.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-2	-1	+0	+1	+2	+3	Σ
+0	1	13	223	15	0	0	252
+1	1	12	253	118	24	2	410
+2	2	11	149	89	27	1	279
+3	0	2	23	24	4	1	54
+4	0	0	3	1	1	0	5
Σ	4	38	651	247	56	4	1000
Σ	**42**		**651**	**307**			

Table 3.11: RSR vs RHOH additional tests/additional faults matrix: **printtokens**. This table is organized the same way as Table 3.7.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-3	-1	+0	+1	+2	+3	+4	+5	Σ
+0	0	0	3	0	0	0	0	0	3
+1	0	0	9	2	0	0	0	0	11
+2	0	0	17	1	0	0	0	0	18
+3	0	0	21	9	2	1	0	0	33
+4	0	0	29	15	3	1	0	0	48
+5	0	1	29	20	8	1	0	0	59
+6	0	0	65	35	22	3	1	0	126
+7	0	0	65	31	18	4	1	0	119
+8	0	1	54	53	26	8	0	0	142
+9	0	4	53	39	32	12	0	0	140
+10	1	0	36	45	18	3	3	0	106
+11	0	4	29	32	18	7	0	0	90
+12	0	0	23	15	9	2	1	1	51
+13	0	2	11	5	9	2	1	0	30
+14	0	0	5	3	5	1	0	0	14
+15	0	0	1	4	1	0	0	0	6
+16	0	1	0	2	0	0	0	0	3
+18	0	0	1	0	0	0	0	0	1
Σ	1	13	451	311	171	45	7	1	1000
Σ	**14**		**451**	**535**					

Table 3.12: RSR vs RHOH additional tests/additional faults matrix: **printtokens2**. This table is organized the same way as Table 3.7.

$\lvert T\rvert_{add}$ \ $\lvert F\rvert_{add}$	-3	-2	-1	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15	+16	Σ
+0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
+1	0	0	0	3	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
+2	0	0	0	5	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	8
+3	0	0	0	5	4	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	13
+4	0	0	1	6	6	3	7	1	1	1	0	1	0	0	0	0	0	0	0	0	27
+5	0	1	0	9	5	3	4	5	2	2	0	0	2	0	0	0	0	0	0	0	33
+6	0	0	0	4	5	2	3	0	4	3	3	1	0	0	0	0	0	0	0	0	25
+7	0	0	0	3	5	5	1	2	5	5	4	3	2	0	0	0	0	0	0	0	35
+8	1	0	0	3	5	5	5	5	11	7	4	2	2	1	0	2	0	0	0	0	53
+9	0	1	0	1	5	6	6	7	2	9	7	6	3	2	0	0	0	0	0	0	55
+10	0	0	1	0	6	6	7	14	6	10	6	7	2	4	0	0	0	0	0	0	69
+11	1	0	0	1	6	6	7	8	7	12	8	12	9	4	3	3	2	0	0	0	89
+12	0	0	0	1	4	4	10	13	12	16	15	14	9	6	2	2	0	0	0	0	108
+13	0	0	0	1	1	7	6	16	7	11	13	11	10	12	7	2	1	2	0	0	107
+14	0	0	0	0	2	3	13	13	13	19	18	12	9	11	7	4	1	1	1	0	127
+15	0	0	0	0	1	8	6	9	18	8	15	8	7	10	6	5	0	0	0	1	102
+16	0	0	0	0	1	3	5	6	5	5	5	3	10	9	5	3	1	1	1	0	63
+17	0	0	0	0	0	0	3	7	1	7	6	6	6	4	2	1	0	0	0	0	43
+18	0	0	0	0	0	2	2	3	1	1	3	3	5	2	1	1	0	0	0	0	24
+19	0	0	0	0	0	0	0	0	2	0	1	2	0	0	0	1	0	1	0	0	7
+20	0	0	0	0	0	0	0	1	0	0	1	2	1	0	0	1	0	0	0	0	6
Σ	2	2	2	43	58	67	87	111	97	116	109	93	77	65	33	25	5	5	2	1	1000
Σ		**6**		**43**	**951**																

Table 3.13: RSR vs RHOH additional tests/additional faults matrix: **replace**. This table is organized the same way as Table 3.7.

Figures 3.1 and 3.2 illustrate plots comparing Experiment RSR with Experiment RHOH. To save space, data is presented only for suite size range 0 – 0.5 of each subject program.

Figure 3.1 depicts a boxplot showing a set of boxes for each subject program. The x-axis represents the reduction technique used for each subject program (RSR and RHOH), and the y-axis represents both the percentage suite size reduction (white boxes) and the percentage fault detection loss (gray boxes). The height of each box in a boxplot represents the range of y-values for the middle 50% of the suites from the 1000-suite collection. The horizontal line within each box represents the median value. The bottom of each box represents the lower quartile, and the top of each box represents the upper quartile. The vertical line stretching below each box ends at the minimum value, and represents the range of the lowest 25% of the values. The vertical line stretching above each box ends at the maximum value, and represents the range of the highest 25% of the values. The average value is depicted by a small x.

We make the following observations from Figure 3.1:

- The white boxes in this figure indicate clearly that RSR generally achieves less suite size reduction than RHOH, but both techniques still achieve relatively high suite size reduction (no less than about 65% average reduction and 75% median reduction for either technique across all experimental subjects).

- The gray boxes clearly show across all programs that the RSR technique achieves less average percentage fault detection loss of suites than the RHOH technique. In fact, the difference in average fault detection loss values between the two techniques seems to always be about the same or greater than the difference in average percentage suite size reduction values. This shows a strong tendency for the cost of RSR in yielding slightly larger reduced suites to be well worth the relatively significant improvements in fault detection retention.

- For all programs except schedule and printtokens2, the median fault detection loss of RSR is significantly less than the median fault detection loss of RHOH

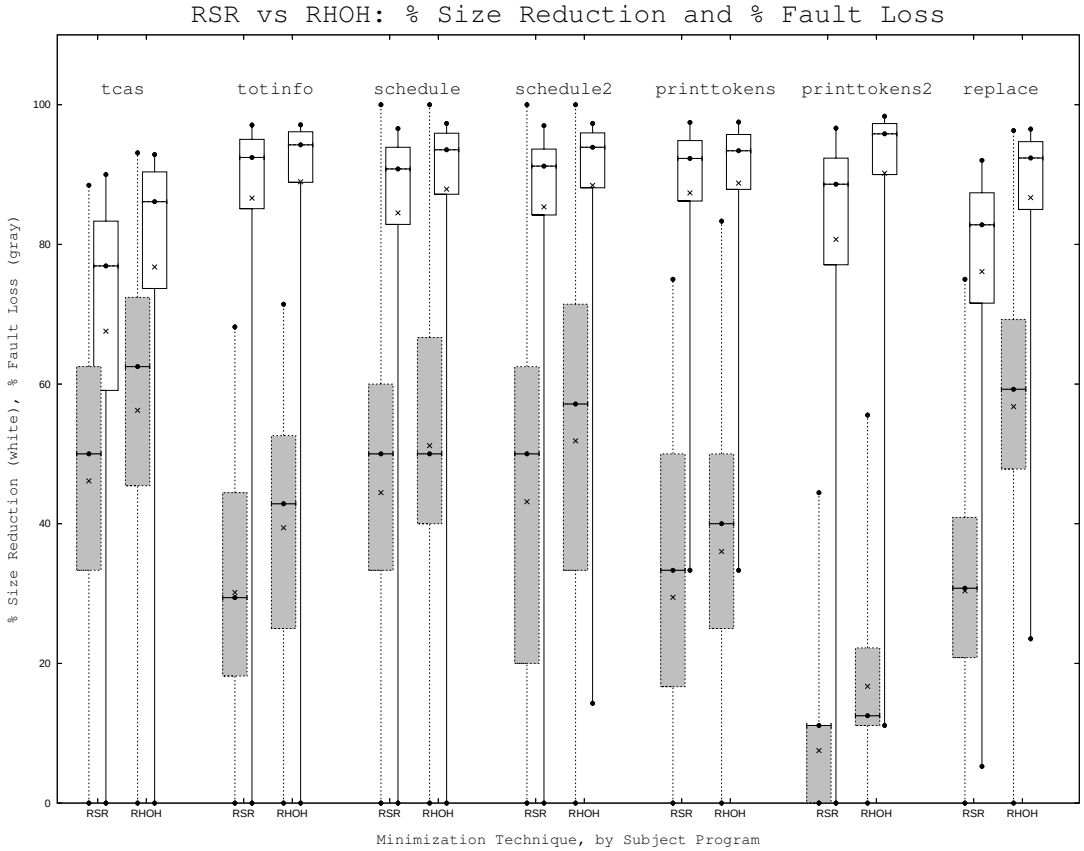

Figure 3.1: The percentage suite size reduction and percentage fault detection loss for the RSR and RHOH techniques, in boxplot format for each subject program.

(with a greater difference than the difference in median values between the two techniques for the percentage suite size reduction). For schedule, the median fault loss values are virtually the same between the two techniques, and for printtokens2, the RSR median is only slightly less than the RHOH median.

Figure 3.2 shows another boxplot, with one box for each subject program. Here, the x-axis represents the subject program, and the y-axis represents the additional-faults-to-additional-tests ratio when comparing the RSR-reduced suites over the corresponding RHOH-minimized suites. This ratio is defined only for those suites in which RSR computes a larger reduced suite than the corresponding suite computed by RHOH. The ratio intuitively shows, among those particular test suites for which RSR computes a larger reduced suite than RHOH, how well those additional test cases in each suite perform in terms of increasing the number of distinct faults detected by the reduced suite. Specifically, the ratio represents, for each additional test case in an RSR-reduced suite over the corresponding RHOH-minimized suite, the number of additional faults detected by that RSR-reduced suite. The ratio can be negative if the RSR suite is larger but detects fewer faults than the RHOH suite. Further, the ratio can be a fractional value if, for instance, the RSR suite contains 3 additional tests but only detects 1 additional fault. Moreover, the ratio can be greater than 1 if, for instance, the RSR suite contains 2 additional tests but detects 4 additional faults (this is possible since the RSR-reduced suite is not necessarily a superset of the corresponding RHOH-minimized suite).

Figure 3.2 is perhaps the strongest evidence showing the benefit of RSR over RHOH. From this figure, we make the following observations:

- For every subject program, the average ratio value is above 0.
- For tcas, totinfo, printtokens2, and replace, the median ratio value is above 0, indicating that over half of the ratio values are greater than 0. For schedule, schedule2, and printtokens, the median value is at 0 with a lower quartile also at 0, indicating that over half of the ratio values are greater than or equal to 0. Note that schedule, schedule2, and printtokens are the three subject programs

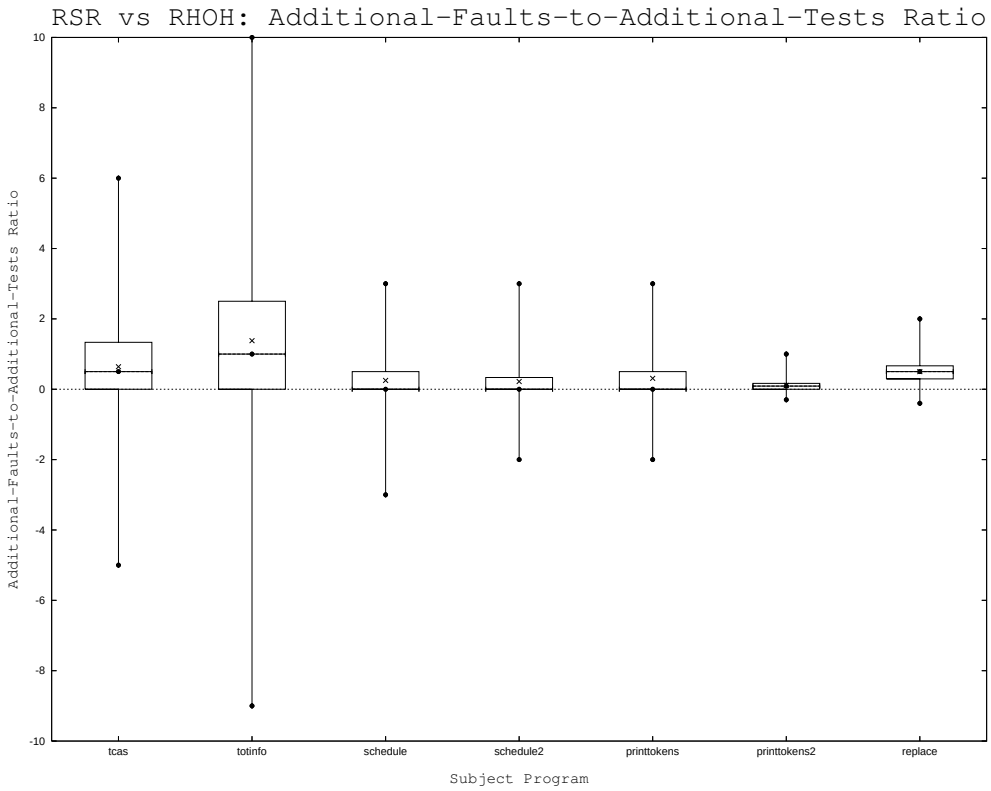

Figure 3.2: The additional-faults-to-additional-tests ratio computed from the RSR-reduced suites over the RHOH-minimized suites, in boxplot format for each subject program.

with the fewest number of faulty versions available (10 or fewer faulty versions each), so we can expect many RSR suites to not detect new faults simply because there are not many faulty versions available for these three programs. This is especially true considering that for these three experimental subjects, RSR only selects between 4 and 6 additional tests than RHOH. Thus, we can expect a large number of suites with ratio 0 for schedule, schedule2, and printtokens. This is also supported by the suite counts described in Tables 3.9, 3.10, and 3.11, in which most suites experienced no change in fault detection effectiveness when going from RHOH to RSR.

- For tcas, the upper quartile is over 1 (more than 25% of suites have ratio value greater than 1), and for totinfo, the upper quartile is over 2 (more than 25% of suites have ratio value greater than 2!). For replace, even the lower quartile is greater than 0 (over 75% of suites have a positive ratio value). This indicates very clear tendencies for these particular programs to have RSR selecting just those additional test cases that are very likely to detect new distinct faults. Interestingly, these three particular subjects have the most faulty versions available (over 20 each). A conclusion is that the more faulty versions that are available, the more likely it is that RSR will experience noticeable improvements in the fault detection effectiveness of a reduced suite over RHOH. This is consistent with our intuition that RSR selects additional tests that are highly likely to detect new faults; the more faults that can potentially be detected, the more likely it is that RSR will detect additional faults. It makes sense that if a wider variety of faulty versions is available, the benefit of RSR over RHOH will be more pronounced in terms of considering the additional-faults-to-additional-tests ratio. Using a wider variety of faulty versions would be an interesting area of exploration for future work.

An interesting point of discussion is to consider the timing requirements of the RSR technique. This consideration is important because the primary motivation for test suite reduction techniques is that testers may be under severe time and resource

constraints. If the RSR technique requires more time than it takes to just run the test cases in the first place, this will defeat the purpose of our technique. We argue that the timing requirements of the RSR technique are negligible when compared to the time it takes to execute a test suite over and over during multiple testing iterations of software.

The RSR technique requires a mapping of each test case to sets of coverage requirements exercised by that test case. In order to compute this in general, we must execute the test case. Only after computing this for each test in a suite can the RSR technique finally be executed. Clearly, the RSR technique therefore takes at least as much time and resources as it takes to execute a test suite once. However, recall that software testing occurs continuously during the software development lifecycle. This implies that one test suite will be executed potentially many times during the evolution of a software system. The total time and resources required to continually execute a single test suite may then become quite large over multiple testing iterations. On the other hand, our RSR technique only needs to be executed once to potentially throw away many test cases from a suite. Therefore, we expect that in practice, the time and resource requirements of RSR are negligible compared to the potential savings obtained from removing significant numbers of test cases with RSR.

Overall, our experimental results show a clear benefit of RSR over RHOH in terms of trading a small amount of suite size reduction for a significant improvement in the fault detection capability of suites. Our results also suggest that RSR may even improve upon the fault detection retention experienced by both the U technique and the E+U technique. Nevertheless, some may argue that fault detection loss still seems quite high across all studied reduction techniques, even for RSR. We argue, to the contrary, that the results for all techniques, even for technique RHOH, are *good and encouraging* for the process of test suite reduction! In all of the techniques we studied, the fault detection loss of reduced suites was generally significantly less than the amount of suite size reduction. For instance, for totinfo, suite size range 0 – 0.5, the suites on average experienced over 88% suite size reduction while only losing

about 40% of the fault detection effectiveness using technique RHOH. If on average, we can remove nearly 90% of the test cases from suites while losing only less than half of the fault detection effectiveness, these results are quite remarkable! It is also argued [3] that hand-seeded faults (like those used in our experimental study) seem to be generally harder to detect than "real" faults, leading to an understatement of the fault detection abilities of suites. This implies that our experimental results may be overstating the fault detection loss of suites due to reduction, relative to what might be experienced in practice with real faults. Given this empirical suggestion, the fault detection loss values reported in our experimental study seem all the more encouraging.

3.3 Chapter Summary

This chapter has described a set of experiments in which we compared the reduction results of our new technique with the minimization results of several existing minimization techniques. It was observed that our new technique has a strong tendency to improve the fault detection retention of reduced suites without significantly compromising the size reduction of the suites, compared to traditional minimization techniques. Further, our empirical evidence suggests that our technique does a good job of selecting just those additional tests (beyond those selected by existing minimization techniques) that are highly likely to expose new faults in software.

Even with our encouraging experimental results, there is still much room for improvement in reduction techniques. Our ideal goal is to achieve significant suite size reduction while losing little to none of the fault detection effectiveness of suites. Our RSR technique is a first step in the right direction toward this goal. Future work will include ways of exploring the notion of reduction with selective coverage redundancy to improve the fault detection retention of reduced suites. For instance, multiple secondary criteria may be used to further improve fault detection retention.

In the next chapter, we present related work in the areas of test suite minimization and fault detection effectiveness.

CHAPTER 4

Related Work

The research work related to this thesis falls under the two general categories of *test suite minimization* and *fault detection effectiveness*. These two categories are not mutually-exclusive, since work in test suite minimization has also sometimes incorporated the notion of fault detection effectiveness as a means for evaluating and comparing minimization techniques. We first discuss the related work focusing on test suite minimization (which may also involve the notion of fault detection effectiveness) and then afterward we will discuss additional related work involving fault detection effectiveness that has a focus other than test suite minimization.

4.1 Test Suite Minimization Research

Previous test suite minimization research has involved two main categories of minimization techniques: optimal techniques, which attempt to compute optimally-minimized suites at the cost of potentially high runtime, and heuristics, which attempt to find near-optimal solutions more quickly. Since the test suite minimization problem is an instance of the set-cover problem and is therefore **NP**-Complete [13], most related work has pursued minimization heuristics. In the following paragraphs discussing work related to test suite minimization, entries 1 – 3 involve optimal minimization research, and entries 4 – 15 involve work with minimization heuristics.

(1) ATACMIN

A tool called ATACMIN, which is part of the ATAC tool package [19], contains

an implementation that computes optimally-minimized suites, given a set of coverage requirements and the set of test cases satisfying each requirement. The approach taken by the tool is to implicitly enumerate subsets until an optimal minimized suite is found. While this algorithm clearly has theoretical runtime exponential in the worst case, in practice the execution is quite fast with relatively small suites. The group of researchers Wong et al. have conducted a series of empirical studies into test suite minimization using ATACMIN's optimally-minimized suites.

(2) Empirical Studies Using ATACMIN: Wong et al.

Wong et al. [42, 43] showed that optimally minimizing suites with respect to all-uses coverage can lead to significant suite size reduction with only very slight losses in fault detection effectiveness. Indeed, these results are in stark contrast to those presented by many other researchers [18, 23, 34], in which empirical evidence suggested that severe and unpredictable fault detection loss of suites may be the rule rather than the exception. While this introduces a conflict in the research community as to how test suite minimization techniques may affect the fault detection effectiveness of suites, it is also true that the experimental setup followed by Wong et al. is quite distinct from those of the other researchers. For instance, the experiments conducted by Wong et al. differed from those of Rothermel et al. [34] in the following significant ways:

- The subject programs were different: Rothermel used programs that were larger on average than the Wong programs.
- The minimization technique used was different: Rothermel used the HGS heuristic to compute near-optimal minimized suites, while Wong computed optimally-minimized suites using the ATACMIN tool.
- The difficulty of the faulty versions was different: Rothermel's work involved faulty versions that were generally harder to detect by the corresponding suites

than the faults introduced in the Wong work.

- The coverage criteria used for generating and minimizing suites were different: Rothermel generated suites for edge-coverage adequacy and also minimized suites with respect to edge coverage; Wong generated suites for varying levels of non-adequate block coverage and minimized suites with respect to all-uses coverage.

- The types of test cases used in the suites were different: Rothermel used test cases generated by the Siemens researchers for various white-box and black-box testing criteria, while Wong used randomly-generated test cases.

A more detailed work conducted by Wong et al. [45] showed similar results to the other work [42, 43]. This work followed a similar experimental setup as in the previous work, but minimization was carried out with respect to three distinct coverage criteria. Further, the authors tried minimizing test pools, effectively considering each test case pool to be a test suite. The test case pool results showed that fault detection loss was higher than for the other, smaller suites, but the amount of detection loss was not severe in general. An implication of the work is that the coverage of a test suite is more important in determining its fault detection effectiveness than the size of the suite.

Wong et al. [46] further empirically studied the effects of minimization on the fault detection effectiveness of suites using a larger program called *space*. Again, the results were consistent with the other Wong studies that significant suite size reduction can be achieved with little to no accompanying loss in fault detection effectiveness.

As a result of the work by Wong et al., the research community now seems to agree that test suite minimization techniques can often significantly reduce suite size. However, there is some disagreement in terms of how much fault detection effectiveness loss occurs. Clearly, situations exist in which high suite size reduction can be achieved with very little fault detection effectiveness loss (according to some researchers), but situations also exist in which high suite size reduction can be

achieved with significant fault detection effectiveness loss (according to others).

(3) A Bi-Criteria Approach to Achieve Optimal Minimizations

Black et al. [5] recently proposed a new model for test suite minimization that explicitly considers two objectives: minimizing a test suite with respect to a particular level of coverage, while simultaneously trying to maximize error detection rates with respect to one particular fault. In other words, this bi-criteria approach to test suite minimization requires ahead of time fault detection information for each test case with respect to a particular faulty version of software. This is distinct from most other minimization research, in which fault detection effectiveness information is used primarily as a means for evaluating reduced suites *after* they have been minimized. The key idea for this approach is to formulate the minimization problem as a binary integer linear programming problem, involving the notion of a *weighting factor*, which determines the degree to which each of the two objectives contribute influence toward the final result. A maximum weighting factor of 1.0 places sole emphasis on minimizing suite size, thus allowing optimally-minimized suites to be computed. A minimum weighting factor of 0.0 places sole emphasis on keeping all the fault-detecting tests, without regard to final suite size and making sure only that all requirements are covered by the reduced suite. Thus, this technique provides great flexibility for testers to balance the trade-off between suite size reduction and fault detection effectiveness loss, by way of setting an appropriate weighting factor to suit their needs. A limitation of this approach is that fault detection information is only considered for each test case with respect to a single fault (rather than a collection of faults), and therefore there may be limited confidence that a suite reduced to keep all fault-detecting tests will be useful in detecting a wide variety of other faults. An empirical study was conducted across a variety of weighting factors to show how suite sizes and fault detection levels could vary depending upon the particular weighting factor used.

(4) The HGS Algorithm

Harrold, Gupta, and Soffa [16] presented a heuristic (the "HGS algorithm") for test suite reduction that attempts to minimize a test suite with respect to a given set of program requirements. The approach is to greedily choose the next test case that covers the requirement which is the next-hardest to satisfy by the suite, continuing until all requirements covered by the original suite are also covered by the reduced suite. In this way, the algorithm attempts to remove as much requirement coverage redundancy from the suite as possible. An empirical study conducted by the authors showed that the sizes of test suites could be significantly reduced by their algorithm. Indeed, this result is supported by subsequent research work [34, 35, 36], including the empirical results from this thesis.

(5) Empirical Studies Using the HGS Algorithm: Rothermel et al.

Rothermel et al. [34, 35, 36] conducted a series of empirical investigations into test suite minimization using the algorithm proposed by Harrold, Gupta, and Soffa [16].

It was argued [34] that contrary to the results suggested by previous work [42, 43, 45, 46], test suite minimization can severely compromise the fault detection capabilities of test suites. The goal of this work was to extend the empirical study conducted by Wong et al. [43] by minimizing suites that were larger, including coverage-adequate suites with varying levels of coverage redundancy, and that were spread out over a larger range of sizes. An empirical study demonstrated that although significant suite size reduction could be achieved by the HGS minimization algorithm, the fault detection effectiveness loss of suites varied widely, regardless of the sizes of the original suites and the amount of suite size reduction. Such results suggest that the fault detection effectiveness loss of suites due to minimization can be severe and unpredictable.

More detailed studies [35, 36] involved three primary contributions in addition

to the work presented earlier [34]. First, an empirical study was conducted to compare the fault detection loss of suites minimized by the HGS algorithm with the fault detection loss of suites minimized randomly. As expected, the results showed that there was significantly greater fault detection loss of the suites on average for the randomly-minimized suites. Second, the authors minimized suites using the HGS algorithm with respect to two different coverage criteria. However, the results showed that in both cases, significant suite size reduction was achieved at the expense of significant loss in fault detection effectiveness. Third, the authors conducted a new empirical study involving the subject program *space*, which was significantly larger than any of the subject programs used in the other work [34]. The results of the *space* study showed again that suites sizes could be significantly reduced due to minimization, but the fault detection loss could still vary widely regardless of the sizes of the non-minimized suites. However, the average percentage fault detection loss for a particular suite in *space* was significantly less than the average percentage fault detection loss for any suite among the other smaller subject programs used by the authors for empirical study. Nevertheless, the work presented here supports the conclusion in the other work by Rothermel et al. [34] that minimization can lead to severe and unpredictable losses in fault detection effectiveness of suites, even though suite sizes can be reduced significantly.

(6) Minimizing for Probabilistic Statement Sensitivity Coverage

An honors thesis written by von Ronne [41] extends the work of Rothermel et al. [34] and makes a significant contribution to minimization research: modifying the original HGS minimization heuristic to come up with a new "multi-hit" algorithm to minimize with respect to a new coverage criterion developed from *mutation analysis*. Mutation analysis involves creating a set of *mutants* from a base version of a program (which are intuitively like "faulty versions"), and then analyzing the behavior of test cases with respect to the mutants and base version of the program. Using the idea of mutation analysis and techniques presented in a

previous work [40], von Ronne describes a new coverage criterion called *probabilistic statement sensitivity coverage* (PSSC). This criterion requires, for each statement in a program, an estimation of that statement's *sensitivity*. The sensitivity of a statement is a measure of the number of times the statement must be executed to achieve a given level of confidence that if a fault existed in that statement, then the fault would be revealed. Thus, PSSC requires that each statement be executed enough times such that there is a minimum likelihood that if a fault exists in the statement, then it will be revealed by one of the test cases in the reduced suite. In the original HGS algorithm, a particular requirement is considered covered if only one test case covers it. However, PSSC requires that each statement be covered an arbitrary number of times before it is considered "covered", and von Ronne therefore presents a modified version of the HGS algorithm that makes sure each requirement is covered some specified minimum number of times in the reduced suite (von Ronne assumes the situation will not occur in which a particular requirement cannot be covered the specified minimum number of times). An empirical study was conducted across a range of suite sizes and PSSC confidence levels (a higher confidence level implies a larger number of times each statement must be exercised). Significant suite size reduction was still achievable by the new technique, although as expected, the higher confidence levels required significantly more tests in the reduced suites than the lower confidence levels. In terms of fault detection effectiveness, the study shows that fault detection retention can be dramatically improved at higher confidence levels, likely due largely to the fact that higher confidence levels require significantly larger reduced suites. However, at lower confidence levels, fault detection loss is still severe on average. This suggests that under the new technique, testers would need to either set a low confidence level and risk severely compromising fault detection effectiveness, or else set a high confidence level and suffer significantly less suite size reduction in order to significantly improve the chances of increasing the fault detection retention of reduced suites. This approach thus provides flexibility for testers to better balance suite size reduction and fault detection loss to suit their minimization needs.

(7) Concept Analysis-Based Minimization Techniques

Sampath et al. [37] presented a new technique for the minimization of test suites, and the incremental update of minimized suites, in the context of *concept analysis* and user-session-based testing of web applications. Concept analysis is a mathematical technique for data analysis that involves clustering objects that have common discrete attributes. In the study, an "object" is considered to be a user session (a test case consisting of URLs requested by the user) and an "attribute" is considered to be a requested URL. A *concept lattice* is a diagram depicting a partial ordering of *concepts*, which are comprised of sets of objects and attributes. The key idea for the minimization technique is to take advantage of the following structural property of a concept lattice: selecting one object from each node in the concept lattice at the lowest level and the second-to-lowest level of the lattice will guarantee that the selected set of objects will have all of the attributes present in the entire lattice. In other words, selecting one test case from each node of the bottom two levels of the concept lattice will guarantee the same URL coverage as the entire test suite, and this is how such suites can be minimized with respect to URL coverage. The authors also present an incremental update algorithm to update a reduced suite based on changes to the concept lattice (changing the test cases present in the suite). While the incremental update algorithm can be carried out in time linear to the number of test cases in a suite, the batch algorithm (involving building a lattice) may run in exponential time due to the fact that a concept lattice can grow to exponential size. An empirical investigation reducing a single test suite found that minimizing with this technique resulted in very little to no statement or function coverage loss of the suite. However, a moderate (but not severe) loss in fault detection effectiveness of the suite was witnessed (namely, 20% detection loss). The fault loss, however, was relatively predictable in this case because the undetected faults were mostly related to name-value pairs in the web application. Thus, these results somewhat support the results of Rothermel et al. [34] that significant fault detection loss of suites may

occur due to minimization.

Following up to the work of Sampath et al. is the work of Sprenkle et al. [38], which seeks to empirically compare the results of three concept analysis-based minimization approaches (each approach selects a different type of test case from each node in the concept lattice) to the results of three requirements-based minimization approaches (including the HGS algorithm). Empirical results with two subject programs showed that the requirements-based approaches with respect to code coverage computed larger minimized suites overall than the concept analysis-based approaches with respect to URL requests. However, other requirements-based approaches with respect to URL request coverage computed smaller minimized suites than the concept analysis approaches. In terms of fault detection effectiveness, the results showed that the percentage effectiveness loss of suites was moderate, but there was no clear winner between the requirements-based techniques and the concept analysis-based techniques in terms of one type consistently retaining more fault detection capability in suites than the other. However, timing measurements indicated that running the requirements-based techniques took considerably longer time as a result of the step mapping each test case to the set of various program requirements it covers. Running the concept analysis-based techniques, on the other hand, did not require as much time, even accounting for the time taken to construct the concept lattice. This result is quite interesting, considering that the time taken to execute the concept analysis-based techniques is theoretically exponential in the worst case. As a result, this work provides empirical evidence that concept analysis-based minimization techniques may be considered a valid alternative to the usual requirements-based techniques in the area of user-session-based testing of web applications.

(8) Minimizing for Modified Condition/Decision Coverage

Jones and Harrold [23] proposed two new techniques for test suite minimization that are tailored to be used specifically in conjunction with the relatively complex

modified condition/decision coverage (MC/DC) criterion. This criterion requires that every condition in a decision be shown by execution to independently affect that outcome of the decision. To show this, the criterion requires that every condition be covered by a particular *MC/DC pair*, which is a pair of truth vectors (truth values for each condition in a decision), each of which causes a different result for the decision, but that differ only by the value of exactly one condition. There may be multiple MC/DC pairs for a particular condition, and execution of any of these pairs provides MC/DC coverage for that condition. This also allows two different test cases to each cover one of the truth vectors for a particular MC/DC pair to cover the pair. In other words, an MC/DC pair can be partially covered by one test case, then fully covered by a second test case. Thus, whereas coverage-based techniques require that each requirement be covered by at least one test case in a suite (where each requirement is fully covered by one test case), the MC/DC criterion requires that each condition be covered by at least one (of possibly multiple) MC/DC pairs exercised by a suite (where each MC/DC pair is fully covered by either one or two test cases). The two new minimization techniques presented in this work account for these significant differences between the coverage of the MC/DC criterion and the coverage of other traditional coverage criteria. The key idea for the first technique is that it is a "break-down" technique, starting with the original suite and breaking it down by removing the test cases that are the weakest in terms of contributing to the MC/DC coverage, meanwhile identifying a set of essential test cases that achieves the same coverage as the original suite. The key idea for the second technique is that it is a "build-up" technique, starting with an empty set and building it up by selecting the test case that is the next strongest in terms of contributing to the MC/DC coverage, until the selected set achieves the same coverage as the original suite. An empirical study showed that both minimization techniques could achieve significant reductions in suite size. However, for both techniques the fault detection effectiveness loss was still relatively severe and unpredictable, consistent with the results reported by Rothermel et al. [34]. However, the build-up technique has a performance advantage in that empirical

results suggest a linear increase in required computation time as suite sizes increase, while the results for the break-down technique suggest a quadratic-time increase in required computation time as suite sizes increase.

(9) Minimizing Using Mega Blocks and Global Dominator Graphs

A new framework for the minimization of test suites in the context of *mega blocks* and *global dominator graphs* was (implicitly) proposed by Agrawal [1, 2]. In this work, Agrawal presents a technique for identifying a subset of the statements and branches in a program with the property such that covering this subset implies covering the rest of the statements and branches. Thus, this implies a new approach to minimization in which test suites are minimized with respect to only those requirements involved in the subset identified by Agrawal's technique. To identify the subset of requirements with this special property, Agrawal's technique requires the computation of mega blocks and a global dominator graph for a particular program. A mega block is a set of basic blocks (possibly spanning multiple procedures) with the property that any one basic block within it is executed if and only if every basic block within it is executed. A global dominator graph is then a directed acyclic graph showing dominator relationships among mega blocks. One then need only to choose tests aimed at executing one basic block within each leaf node of the graph; these tests will then be guaranteed to cover all of the other basic blocks. Clearly, the idea presented in this algorithm can be applied to other requirements-based techniques such as the HGS algorithm, because it provides a way of reducing the size of the set of requirements for which a minimization technique needs to provide coverage.

(10) Minimizing Mutation-Based Test Suites By Varying Test Execution Order

Offutt et al. [32] presented a new idea for test suite minimization in the context of mutation-based testing, suggesting that changing the order in which test cases

are executed on mutants can lead to different minimization results. *Mutation-based testing* is a fault-based testing technique that relies on the assumption that a program will be well tested if all "simple faults" are detected and removed. This approach is considered valid due to the *coupling effect*, for which empirical evidence [9, 31] suggests that test suites detecting simple faults are sensitive enough to also detect more complicated types of faults. That is, explicitly testing for simple faults also strongly tends to implicitly test for complex faults, and therefore fault-based testing is an effective way to test software. Under mutation testing, a collection of *mutants* are created for a base program such that each mutant is the same as the base program except for a simple error that has been introduced. A mutant is said to be *killed* by a test case if the test case exposes the fault in the mutant (causes the mutant to produce incorrect output). A killed mutant is then removed from the set of mutants before other test cases are executed. The *mutation score* of a test suite is the ratio of dead mutants to the total number of mutants (that are not functionally equivalent to the base program). This is a measure of the adequacy of a mutation-based test suite. A minimized test suite is considered to be the smallest subset of the suite that achieves the same mutation score as the original suite. The key idea for minimizing such a suite is to execute all the test cases on the mutants and then remove those test cases which are ineffective in contributing to the mutation score. Since whether a particular test case is effective depends on the order in which tests are executed (recall that a killed mutant is immediately removed so this affects whether subsequently-run tests detect it), then different minimized suites can be computed by executing tests in different orders. An empirical study comparing the minimization results of a variety of different execution orderings of test cases (affectionately called the "ping-pong" heuristics by the authors) showed that test suites could be moderately reduced in size (even up to about 50% reduction) and still maintain the same mutation score as the original suites. As expected, different orderings of test executions did lead to different reduced suites with slightly varying levels of suite size reduction.

(11) Minimization Applied to Model-Based Test Suites

Heimdahl and George [18] pursued the notion of applying a simple greedy heuristic for test suite reduction to model-based (specification-based) tests, and argued as in the work by Rothermel et al. [34] that significant suite size reduction could be achieved due to minimization, but at the expense of an unacceptable loss in the fault detection effectiveness of suites. Under the *model checking* techniques that are the context of this work, test cases are generated from and executed against formal models of software. Thus, in the empirical study conducted by these authors, test suites were created and minimized with respect to a variety of specification-based coverage criteria. While the average fault detection loss values reported in this paper were significantly less than those reported in the Rothermel work [34], the authors still argued that in the critical systems domain in which model checking is often used, even fault detection effectiveness loss of around 10% is unacceptable.

(12) Minimizing By Way of the Operational Difference Technique

Harder et al. [14] proposed a new technique for generating, augmenting, and minimizing test suites called the *operational difference technique*. Similar to the work by Heimdahl and George [18], this work is conducted in the context of analyzing program properties, rather than analyzing the actual program code. However, unlike the work by Heimdahl and George, the program properties analyzed here are due to actual program behavior, and not due to some specification of intended program behavior. The presented technique involves the notion of an *operational abstraction*, which is defined as being identical to a formal specification, except it describes actual behavior (which may or may not match desired behavior). When a set of test cases is run on a program, an operational abstraction describing the program behavior is generated. The major assumption underlying this work is that an operational abstraction generated from a larger test suite is better at

representing actual program behavior than the abstraction generated from a smaller test suite. The key idea for minimizing in this context is then to keep selecting tests into a reduced suite so long as they keep changing the operational abstraction. Any tests that do not change the operational abstraction are not selected. Conversely, a similar approach is to remove those test cases from a suite that do not alter the operational abstraction. An empirical study showed that suites minimized under this new technique were as small or smaller than other branch-coverage adequate suites (implying significant suite size reduction), and were even better at detecting certain types of faults. An implication is that this technique for minimization may be complementary to other coverage-based minimization techniques.

(13) Divide-and-Conquer Minimization Techniques

Chen and Lau [6, 7] studied the general notion of applying divide-and-conquer techniques to the test suite minimization problem to obtain near-optimal solutions. Such a technique involves decomposing the original problem (test suite) into subproblems (subsets of the original suite), computing a reduced suite for each subproblem separately, and then combining the reduced suites to obtain a final reduced suite for the original problem. The authors propose two distinct methods for dividing a test suite into subsets and show how a final near-optimal reduced suite can then be computed [6]. The authors also conducted an empirical investigation using minimization heuristics developed from their dividing strategies and showed that the resulting minimized suites had a relatively high probability of being of optimal size [7].

(14) Comparing Coverage-Based and Distribution-Based Minimization Techniques

Leon and Podgurski [27] conducted an empirical investigation with the goal of comparing two distinct kinds of techniques for minimizing and prioritizing test

suites: *coverage-based techniques* and *distribution-based techniques*. Coverage-based techniques involve considering the coverage of certain program entities in determining whether to select a particular test case. Distribution-based techniques instead make this decision by considering how the execution profiles of test cases are distributed in the total execution profile space. The experimental results suggested that both types of techniques were complementary to each other in terms of their effectiveness in promoting the fault detection capabilities of reduced suites.

(15) A Heuristic for the General Set-Cover Problem

Chvatal [8] presented a greedy heuristic for the more general set-cover problem in which each candidate set has a cost associated with it. The goal of the technique is to find a subset of the collection of candidate sets, of smallest possible cost, in which the subset covers the same elements as the original set. The technique can be applied to finding a near-optimal solution in terms of size if all candidate sets are given equal cost. Chvatal supported the notion that his heuristic computes near-optimal solutions by proving that his heuristic guarantees a logarithmic approximation factor to the optimal solution.

4.2 Additional Fault Detection Effectiveness Research

The majority of research involving the notion of fault detection effectiveness has often employed this notion as a means for comparison between two or more concepts or techniques. For example, the minimization research that has already been described often used the notion of fault detection effectiveness to compare different minimization techniques. Similarly, fault detection effectiveness has often been used outside the realm of test suite minimization in order to compare things such as coverage criteria. We will now discuss work related to fault detection effectiveness that is not applied to the area of test suite minimization. In the paragraphs that follow, entries 1 – 7 refer to work that uses fault detection effectiveness as a means

for comparison, and entries 8 – 10 refer to work involving other applications of the notion of fault detection.

(1) Relationships Between Criteria and Implications for Fault Detection

Frankl and Weyuker [10] presented a theoretical study in which they defined two arbitrary testing criteria, along with five relations between the criteria, and then proved whether each particular relationship between the criteria sometimes or always guaranteed that one criterion would yield a greater capability of detecting faults than the other criterion. Three distinct measures were used for determining the fault detection effectiveness of the criteria; the effectiveness of a criterion is the probability that a test suite satisfying that criterion will expose a fault. It was shown that it is relatively rare for a particular relationship between two criteria to guarantee that one criterion is definitely always better at detecting faults than the other criterion. In most cases, the strongest statement that can be made, given a relationship between two criteria, is that one criterion will only sometimes be better at detecting faults than the other criterion.

(2) Comparison of Different Decision Coverage Measures

Kapoor and Bowen [24] used the notion of fault detection effectiveness to conduct an empirical study showing that modified condition/decision coverage (MC/DC) is more effective and stable in detecting faults than either decision coverage (DC) or full predicate coverage (FPC). Experiments were conducted using boolean decisions from a particular program as the experimental subjects, across varying numbers of conditions. For each subject, all possible test suites satisfying a particular criterion were computed, and the effectiveness of each test suite was measured. This allowed the researchers to compute the effectiveness of each criterion, because this is simply the probability that a test suite randomly selected from the set of all possible test suites satisfying a criterion will detect a fault.

Empirical results showed that unlike for MC/DC, the effectiveness of both DC and FPC tended to deteriorate as the number of conditions in the experimental subjects increased (largely due to the fact that as the number of conditions increases, the total number of possible test suites satisfying a criterion grows exponentially, leading to a decrease in effectiveness if many of these test suites do not detect faults). In general, the results showed that the MC/DC criterion was generally more effective than the FPC criterion, which was generally more effective than the DC criterion.

(3) Comparison of the All-Edges and All-Definition-Uses Coverage Criteria

Hutchins et al. [20] sought to use the notion of fault detection effectiveness to empirically compare the all-edges control-flow coverage criterion to the all-definition-uses data-flow criterion. The experimental setup used in this experiment motivated the creation of the well-known Siemens subject programs, faulty versions, and test case pools used in this thesis. The approach of the experiments was to generate test suites for varying levels of coverage for each of the two coverage criteria, and then measure the fault detection effectiveness of the generated suites. It was shown that in general, the definition-use pair coverage suites were better at detecting more faults than the all-edges suites, in particular at the higher percentage coverage ranges. However, for both criteria, the average number of faults detected by the suites increased in a near-quadratic fashion as the percentage coverage increased to 100%. Thus, there are significant improvements in effectiveness for both sets of suites at the highest coverage ranges, such as from 90% to 100% coverage. This implies that 100% coverage-adequate suites are generally preferable to suites that may be even 90% or 95% coverage-adequate. The authors also suggest that the types of faults detected by the suites for each of the two criteria tended to be different, implying that the two criteria could be complementary to each other in terms of their fault-exposing potential. However, the authors caution that even at high coverage ranges, individual test suites tended to have significant variations in their fault detection

effectiveness. Thus, a suite achieving high coverage of a criterion did not always guarantee a high capability of exposing faults.

The work of Frankl and Weiss [12] supports the results of Hutchins et al. [20]. Here, the authors again sought to use the notion of fault detection to empirically compare the all-edges criterion to the all-uses criterion ("all-uses" was defined the same way as "all-definition-uses" in the work by Hutchins et al.). The main difference between this work and that of Hutchins et al. is in the experimental setup, which involved a different set of subject programs. It was shown here that suites created for the all-uses criterion were usually, but not always, significantly better at detecting faults than the suites created for the all-edges criterion. Further, it was shown that suites created for the all-edges criterion were not significantly better at detecting faults than randomly-generated suites of the same sizes. The results also showed only a moderate correlation between the fault-exposing potential of a suite and the percentage coverage achieved by the suite. These results suggest that high coverage of a particular criterion does not always guarantee proportionally-high fault-exposing potential.

Further empirical studies were conducted by Frankl and Iakounenko [11] to again compare the fault-detecting abilities of suites created for edge coverage with suites created for all-uses coverage. The main difference here is that the experimental subjects were all versions of a large, real-world program (each version is over 10,000 lines of code) developed for the European Space Agency. In contrast to previous results [12, 20], the results showed that the edge-coverage suites and the all-uses suites were very similar in terms of their fault-detecting ability, and both types of suites were significantly better at detecting faults than randomly-created suites of the same sizes. However, while there were significant improvements in fault-detecting ability at the higher percentage coverage ranges, in general, even the suites achieving the most edges or all-uses coverage were not highly effective at detecting faults.

(4) Mutation-Based Test Suites versus All-Uses Test Suites

Mathur and Wong [28] conducted an empirical investigation to show that test suites generated for mutation-based testing tend to have superior fault detection effectiveness to suites generated for all-uses coverage. The experiments also analyzed two restricted versions of mutation-based testing: one in which mutations were restricted to two types, and the other in which only 10% of the available mutants were considered. The results showed that in general, the criteria listed in decreasing order of effectiveness were as follows: regular mutation criterion $>$ two-type restricted mutation criterion $>$ all-uses criterion $>$ 10% restricted criterion. Hence, mutation-based test adequacy criteria are shown to be generally more effective in terms of fault detection than the all-uses data-flow criterion.

(5) Testing Process Measures versus Product Measures

Morgan et al. [30] showed that testing process measures do a better job of predicting fault detection effectiveness than product measures. Examples of testing process measures include test suite size and various coverage measures such as block, decision, and all-uses coverage. Examples of product measures include lines of code and total counts of blocks, decisions, and all-uses. Since testing process measures are more widely-used in research as criteria than product measures, this work suggests that researchers are justified in doing this. Additionally, Morgan et al. also suggest that incorporating both testing process measures and product measures together can improve the chances of increasing fault detection effectiveness of suites. The work cautions, however, that using either type of measure still does not allow researchers to predict fault detection effectiveness to a high degree. This result is consistent with other, more specific research [11, 12, 20] indicating that fault detection effectiveness may vary unpredictably from suite to suite,

regardless of how similar those suites may be in terms of their requirement coverage.

(6) Fault Detection Applied to UML Models

Kawane [25] presents a brief case study comparing the fault detection effectiveness of various test adequacy criteria in the realm of *UML design models*. UML models are a formal specification-based language used by developers to model complex software systems. A small study involving two experimental subjects with 10 and 9 seeded errors showed that 8 and 5 faults were respectively exposed by a suite satisfying various coverage criteria based on UML model elements (class diagrams and interaction diagrams). Here, a fault detection was indicated by one of the following: a violation of constraints on the system operations, an inconsistent system configuration, or a deviation in system behavior from the specification in the use cases. The results of this work do not easily generalize due to the limited experimental setup and relatively small subject, but the work is unique in that it applies the notion of fault detection effectiveness to the area of UML models.

(7) The Correlation Between Suite Coverage/Size and Fault Detection Effectiveness

Wong et al. [44] conducted an empirical study that showed that there is a higher correlation between the block coverage of a suite and the fault detection effectiveness of the suite, than between the size of the suite and the fault detection effectiveness of the suite. While this result is intuitive, it complements other research [11, 12, 20] that seems to cast doubt on whether greater program coverage would tend to imply greater fault detection effectiveness. This study by Wong et al. is unique in that the focus of the work is to analyze the correlation between coverage and effectiveness, and to compare this with the correlation between size and effectiveness. The previous research provided only indirect suggestions about these correlations. An implication from Wong's work is that test cases that do

not add coverage to a test suite are likely to be relatively ineffective in causing the test suite to detect new faults. Clearly, this further implies that test suite minimization may indeed be able to remove coverage-redundant tests from a suite without severely compromising the fault detection effectiveness of the suite. While this implication is consistent with the empirical results of the other Wong studies [42, 43, 45, 46], it remains in stark contrast to other research [18, 23, 34], which suggests that fault detection loss of suites due to minimization can be relatively significant.

(8) A General Approach to Fault-Based Testing

Morell [29] presented a general theoretical study of fault-based testing, describing methods for proving that certain prespecified errors are absent from software. The perspective taken by the work is to view every correct execution trace of a program (derived from a symbolic execution of the program) as containing information that may prove the non-existence of certain errors in the software. The limitation of this approach is that it cannot be applied to arbitrary errors, that is, the approach cannot guarantee that software is free of all possible errors.

(9) Test Case Generation: A Spathic Approach

Hayes and Zhang [17] presented a new "spathic" technique for test case generation that is meant to represent a middle-ground between creating tests to satisfy a coverage criterion (which is relatively harder to accomplish) and generating random test cases (which is relatively easier to accomplish). The approach is to require a tester to only have to characterize the input domain for a piece of software, and then specify whether the input data should tend to be from among the most common input values or from the least common input values. In this way, the approach is similar to random test case generation, but slightly more "intelligent". A small empirical study conducted by the authors

showed that the spathic approach performed virtually the same as random test case generation in terms of generating suites with a certain level of fault detection effectiveness; in only a few cases, suites generated by the spathic approach were slightly better at detecting faults than the corresponding randomly-created suites. In terms of creating suites to achieve code coverage, the spathic approach tended to create suites that were slightly better at achieving higher levels of code coverage than the random approach, with respect to statement and branch coverage.

(10) A Hierarchy of Fault Classes

Kuhn [26] described a hierarchy of fault classes in which test cases detecting one class of faults would be guaranteed to detect other fault classes. To devise this hierarchy, Kuhn sought to determine the exact set of conditions that are required for a particular predicate to expose a fault with respect to a particular fault class. A test case covering this set of conditions will guarantee the detection of a fault from the corresponding fault class. The relationships between these sets of conditions for each fault class determined Kuhn's hierarchy. The benefit of this work is the knowledge that test suites aimed at detecting a certain subset of the classes of faults in the hierarchy will also detect faults from the other fault classes as well.

Tsuchiya and Kikuno [39] performed a follow-up, complementary study to that of Kuhn [26], in which deeper analysis revealed an extended version of Kuhn's hierarchy of fault classes.

4.3 Where Our Work Fits In

This thesis can be categorized as work related to test suite minimization heuristics, although our work involves the notion of test suite *reduction* rather than *minimization*. It is argued in this thesis that some notion of reduction with selective coverage redundancy is useful in order to preserve more fault detection effectiveness in reduced suites, since we generally do not have information about the set of all possible

requirements covered by each test case (we must usually resort to computing the requirements covered by test cases for some particular chosen coverage criteria). Our work proposes a new framework for test suite reduction that incorporates the notion of selective coverage redundancy, and therefore represents a new way of viewing the suite reduction problem.

The goal of our work is to improve the fault detection retention of reduced suites without severely impacting the amount of suite size reduction, and our work therefore uses fault detection of suites as a measure of evaluation for the comparison of our new reduction technique with existing minimization techniques. Our experiments have shown that our reduction with selective redundancy approach can improve the fault detection effectiveness of reduced suites, relative to several other existing techniques, without significantly compromising suite size reduction.

Contrary to the positions taken in other research [18, 23, 34] that fault detection loss due to minimization can be relatively severe, we take a more optimistic stance and argue that previous empirical evidence is actually *encouraging* for test suite minimization. Clearly, there is a benefit of minimizing with respect to a particular coverage criterion, rather than simply minimizing randomly [35, 36]. Moreover, when the percentage fault detection loss is significantly less than the percentage suite size reduction, this certainly implies that minimizing with respect to requirement coverage is a step in the right direction. We believe our work, which provides a fresh look at the minimization problem from a new angle, is one more step in the right direction.

4.4 Chapter Summary

This chapter has presented an overview of the previous research work related to the topics of test suite minimization and fault detection effectiveness. Some research work has gone into studying techniques for computing optimally-minimized suites, but most previous research has involved heuristics for computing near-optimal solutions for minimized suites. Further, some minimization research has used the notion

of fault detection effectiveness as a means for analyzing minimization results. Other research work that does not necessarily involve test suite minimization has also used the notion of fault detection for other purposes.

In the next and final chapter, we discuss the conclusions of our work and our plan for future work.

CHAPTER 5

Conclusions and Future Work

This thesis has introduced the new idea of viewing the test suite minimization problem from the perspective of trying to include those test cases that are redundant with respect to a primary coverage criterion, if the tests are not redundant according to some other secondary coverage criterion. These additional tests are those that are likely to expose new faults in software. Our work is important because testers are likely willing to sacrifice some small amount of test suite size reduction in order to significantly improve the chances of retaining the fault detection effectiveness present in the original, non-reduced suites. We presented a new approach to test suite reduction that attempts to selectively keep some coverage-redundant test cases, with the goal of decreasing the loss in fault detection effectiveness without severely impacting suite size reduction. Our approach is general and can be integrated into any existing test suite minimization algorithm. We presented our approach in the context of a specific new technique based on the minimization heuristic presented by Harrold, Gupta, and Soffa [16]. In our experimental study, our approach consistently performed better on average than other test suite minimization techniques that do not include selective coverage redundancy, by generating reduced test suites with less fault detection loss at the expense of only a relatively small increase in the sizes of the reduced suites.

Future work will look into the notion of using multiple levels of coverage redundancy with respect to different sets of coverage criteria. We expect that expanding our idea of selective coverage redundancy during test suite reduction will allow us to achieve even greater fault detection retention of reduced suites without significantly impacting the suite size reduction of the suites. We hope to explore many other types of coverage requirements that are measurable from test cases. This will provide us with a large set of different criteria that can be used for further empirical

analysis of our reduction with selective redundancy technique.

It will also be interesting to see how empirical results change based on the set of faulty versions used. Our experimental results have suggested that a wider variety of faulty versions (especially for those subject programs with relatively few faulty versions) may better help to highlight the potential benefit of our new technique over existing minimization techniques. Further empirical studies with different sets of faulty versions (different numbers of faulty versions, different types of seeded errors, and different levels of difficulty for detecting each fault) are required to better understand the effects of faulty versions on empirical results in test suite reduction.

Yet another area of future work will be to conduct empirical studies using our new technique on larger subject programs with real faults. While the experiments conducted in this thesis show encouraging results, further empirical study is needed to better understand how well our technique may perform on larger software systems, with real test suites and real faults.

The overall (and we believe, reachable) goal of future work into test suite reduction will be to show how to significantly reduce the size of a test suite by simultaneously allowing little to no loss in fault detection effectiveness. Other research [42, 43] has shown that this can be accomplished in certain situations. Whether it can be accomplished in general is still an open question that requires further work. However, the work presented in this thesis regarding selective coverage redundancy provides very encouraging evidence that research is moving in the right direction for the future of test suite reduction.

APPENDIX A

The Original HGS Minimization Algorithm

The HGS algorithm [16] for reducing the size of a test suite is presented in Figures A.1 and A.2.

Figure A.1 presents the pseudocode for the main HGS algorithm, and Figure A.2 presents a helper function used by the main HGS algorithm to select the next test case to include in the reduced suite. The input to the HGS algorithm is a mapping of each requirement covered by an original test suite to the set of test cases in the suite covering that particular requirement. The goal is to find a representative set of test cases, of smallest possible size, covering the same set of requirements as the original suite. The approach follows a heuristic to greedily select the test cases that cover the requirements that are the hardest to satisfy, until all requirements are covered. A requirement A is deemed harder to satisfy than a requirement B if A is covered by fewer test cases (has a smaller associated test case set size) than B.

The steps of the HGS algorithm can be summarized as follows:

Steps of the HGS Algorithm for Test Suite Minimization

1. Initially, all covered requirements are considered unmarked.

2. For each requirement that is exercised by only one test case, add that test case to the minimized suite and mark all the requirements covered by that test case.

3. Next, consider the unmarked requirements in increasing order of the cardinality of the set of test cases exercising each requirement. Among those test cases in the unmarked sets of the current cardinality under consideration, select the test case that would mark the greatest number of unmarked requirements of

```
define:
    Set of coverage requirements for minimization: r_1, r_2, ..., r_n
input:
    T_1, T_2, ..., T_n: associated test case sets for r_1, r_2, ..., r_n respectively, containing
        test cases from t_1, t_2, ..., t_nt
output:
    RS: a representative set of T_1, T_2, ..., T_n
declare:
    maxCard, curCard: one of 1, ..., nt
    list: list of t_i's
    nextTest: one of t_1, t_2, ..., t_nt
    marked: array[1..n] of boolean, initially FALSE
    mayReduce: boolean
    Max(): returns the maximum of a set of numbers
    Card(): returns the cardinality of a set

algorithm ReduceTestSuite
begin
/* initialization */
    maxCard := Max(Card(T_i));
    RS := ∪T_i such that Card(T_i) = 1;
    foreach T_i such that T_i ∩ RS ≠ ∅ do marked[i] := TRUE;
    curCard := 1;
/* compute RS according to the heuristic for sets of higher cardinality */
    loop
        curCard := curCard + 1;
        while there exists T_i such that Card(T_i) == curCard and not marked[i] do
            list := all t_j ∈ T_i where Card(T_i) == curCard and not marked[i];
            nextTest := SelectTest(curCard, list);
            RS := RS ∪ {nextTest};
            mayReduce := FALSE;
            foreach T_i where nextTest ∈ T_i do
                marked[i] := TRUE;
                if Card(T_i) == maxCard then mayReduce := TRUE;
            endfor
            if mayReduce then
                maxCard := Max(Card(T_i)), for all i where not marked[i];
        endwhile
    until curCard == maxCard;
    return RS;
end ReduceTestSuite
```

Figure A.1: The main HGS heuristic algorithm.

```
function SelectTest(size, list)
/* this function selects the next test case to be included in RS */
declare
  count: array[1..nt]
begin
  foreach t_i in list do compute count[t_i], the number of unmarked T_j's of
    cardinality size containing t_i;
  testList := tests from list for which count[i] is the maximum;
  if Card(testList) == 1 then return (the test case in testList);
  else if size == maxCard then return (any test case in testList);
  else return SelectTest(size+1, testList);
end SelectTest
```

Figure A.2: A helper function used by the original HGS algorithm to select the next test case from a candidate list of tests.

this cardinality. If multiple such test cases are tied, break the tie in favor of the test case that would mark the greatest number of unmarked requirements with test case sets of successively higher cardinalities; if the highest cardinality is reached and some test cases are still tied, arbitrarily select a test case among those that are tied. Then, mark the requirements exercised by the selected test.

4. Repeat Step 3 until all testing requirements are marked, and then return the reduced suite.

REFERENCES

[1] H. Agrawal. "Dominators, Super Blocks, and Program Coverage." *21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 25-34, Portland, Oregon, January 1994.

[2] H. Agrawal. "Efficient Coverage Testing Using Global Dominator Graphs." *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. Toulouse, France, 1999.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche "Is Mutation an Appropriate Tool for Testing Experiments?" *27th International Conference on Software Engineering*. 402-411, St. Louis, Missouri, 2005.

[4] M. Balcer, W. Hasling, and T. Ostrand. "Automatic Generation of Test Scripts from Formal Test Specifications." *Proc. of the 3rd Symp. on Softw. Testing, Analysis, and Verification*. 210-218, Key West, Florida, December 1989.

[5] J. Black, E. Melachrinoudis, and D. Kaeli. "Bi-Criteria Models for All-Uses Test Suite Reduction." *26th Int'l Conf. on Software Engineering*. Edinburgh, Scotland, May 2004.

[6] T. Y. Chen and M. F. Lau. "Dividing Strategies for the Optimization of a Test Suite." *Information Processing Letters*. 60(3):135-141, March 1996.

[7] T. Y. Chen and M. F. Lau. "Heuristics Towards the Optimization of the Size of a Test Suite." *Proc. 3rd Int'l Conf. on Softw. Quality Management*. Vol. 2, 415-424, Seville, Spain, April 1995.

[8] V. Chvatal. "A Greedy Heuristic for the Set-Covering Problem." *Mathematics of Operations Research*. 4(3), August 1979.

[9] R. A. DeMillo and A. P. Mathur. "On the Use of Software Artifacts to Evaluate the Effectiveness of Mutation Analysis for Detecting Errors in Production Software." Technical Report SERC-TR-92-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, August 19, 1994.

[10] P. G. Frankl and E. J. Weyuker. "A Formal Analysis of the Fault-Detecting Ability of Testing Methods." *IEEE Transactions on Software Engineering.* 19(3):202-213, March 1993.

[11] P. Frankl and O. Iakounenko. "Further Empirical Studies of Test Effectiveness." *Proc. of the ACM SIGSOFT Int'l. Sym. on Foundations on Softw. Eng.* 153-162, Lake Buena Vista, Florida, November 1998.

[12] P. G. Frankl and S. N. Weiss. "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing." *IEEE Trans. on Software Engineering.* 19(8):774-787, 1993.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company, New York, NY, 1979.

[14] M. Harder, J. Mellen, and M. D. Ernst "Improving Test Suites via Operational Abstraction." *Proceedings of the 25th International Conference on Software Engineering.* 60-71, Portland, Oregon, May 6-8, 2003.

[15] M. J. Harrold and G. Rothermel. "Aristotle: A System for Research on and Development of Program-Analysis-Based Tools." Technical Report OSU-CISRC-3/97-TR17, Ohio State University, March 1997.

[16] M. J. Harrold, R. Gupta, and M. L. Soffa. "A Methodology for Controlling the Size of a Test Suite." *ACM Trans. on Softw. Eng. and Methodology.* 2(3):270-285, July 1993.

[17] J. H. Hayes and P. Zhang. "Fault Detection Effectiveness of Spathic Test Data." *Proc. of the IEEE Int'l Conf. on Eng. of Complex Computer Systems.* Greenbelt, Maryland, December 2002.

[18] M. P. E. Heimdahl and D. George. "Test-Suite Reduction for Model-Based Tests: Effects on Test Quality and Implications for Testing." *Proc. of the 19th IEEE Int'l Conf. on Automated Softw. Eng.* Linz, Austria, September 2004.

[19] J. R. Horgan and S. A. London. "ATAC: A Data Flow Coverage Testing Tool for C." *Proceedings of Symposium on Assessment of Quality Software Development Tools.* 2-10, New Orleans, Louisiana, May 1992.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria." *Proc. of the 16th Int'l Conf. on Softw. Eng.* 191-200, Sorrento, Italy, May 1994.

[21] `http://www.cse.unl.edu/~galileo/sir`

[22] `http://paul.rutgers.edu/~rhoads/Code/easter.c`

[23] J. A. Jones and M. J. Harrold. "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage." *IEEE Transactions on Software Engineering.* 29(3):195-209, March 2003.

[24] K. Kapoor and J. Bowen. "Experimental Evaluation of the Variation in Effectiveness for DC, FPC and MC/DC Test Criteria." *Proc. IEEE International Symposium on Empirical Software Engineering.* 185-194, Rome, Italy, 2003.

[25] N. Kawane. "Fault Detection Effectiveness of UML Design Model Test Adequacy Criteria." *14th International Symposium on Software Reliability Engineering.* Denver, Colorado, November 17-20, 2003.

[26] D. R. Kuhn. "Fault Classes and Error Detection Capability of Specification Based Testing." *ACM Trans. Softw. Eng. Methodol.* 8(4):411-424, October, 1999.

[27] D. Leon, A. Podgurski. "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases." *14th International Symposium on Software Reliability Engineering.* Denver, Colorado, November 17-20, 2003.

[28] A. P. Mathur and W. E. Wong. "Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An Empirical Study." Technical Report SERC-TR-146-P, Software Engineering Research Center, December 1993.

[29] L. J. Morell. "A Theory of Fault-Based Testing." *IEEE Transactions on Software Engineering.* 16(8):844-857, August 1990.

[30] J. A. Morgan, G. J. Knafl, and W. E. Wong. "Predicting Fault Detection Effectiveness." *Proc. of the 4th IEEE Int'l. Software Metrics Symposium.* 82-89, Albuquerque, New Mexico, November 1997.

[31] A. J. Offutt. "Investigations of the Software Testing Coupling Effect." *ACM Trans. on Softw. Eng. Methodology.* 1(1):3-18, January 1992.

[32] A. J. Offutt, J. Pan, and J. M. Voas. "Procedures for Reducing the Size of Coverage-based Test Sets." *Proc. 12th Int'l Conf. Testing Computer Software*. 111-123, Washington, DC, June 1995.

[33] T. Ostrand and M. Balcer. "The Category-Partition Method for Specifying and Generating Functional Tests." *Communications of the ACM*. 31(6), June 1988.

[34] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites." *International Conference of Software Maintenance*. 34-43, Bethesda, Maryland, November 1998.

[35] G. Rothermel, M.J. Harrold, J. von Ronne, C. Hong, and J. Ostrin. "Experiments to Assess the Cost-Benefits of Test-Suite Reduction." Technical Report 99-60-09, Computer Science Department, Oregon State University, December, 1999.

[36] G. Rothermel, M.J. Harrold, J. von Ronne, and C. Hong. "Empirical Studies of Test-Suite Reduction." *Journal of Software Testing, Verification, and Reliability.* 12(4):219-249, 2002.

[37] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. "A Scalable Approach to User-Session Based Testing of Web Applications through Concept Analysis." *Proc. of the 19th IEEE Int'l Conf. on Automated Softw. Eng.* Linz, Austria, September 2004.

[38] S. Sprenkle, S. Sampath, E. Gibson, A. Souter, and L. Pollock. "An Empirical Comparison of Test Suite Reduction Techniques for User-session-based Testing of Web Applications." Technical Report 2005-009, Computer and Information Sciences, University of Delaware, November 2004.

[39] T. Tsuchiya and T. Kikuno. "On Fault Classes and Error Detection Capability of Specification-Based Testing." *ACM Transactions on Software Engineering and Methodology*. 11(1):58-62, January 2002.

[40] J. M. Voas. "PIE: A Dynamic Failure-Based Technique." *IEEE Transactions on Software Engineering*. 18(8):717-727, August 1992.

[41] J. von Ronne. "Test Suite Minimization: An Empirical Investigation." University Honors College Thesis, Oregon State University, June 1999.

[42] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of Test Set Minimization on the Fault Detection Effectiveness of the All-Uses Criterion." Technical Report SERC-TR-152-P, Software Engineering Research Center, June 1994.

[43] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of Test Set Minimization on Fault Detection Effectiveness." *Proc. 17th Int'l Conf. on Softw. Eng.* 41-50, Seattle, Washington, April 1995.

[44] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness." *Proc. 5th Int'l Symposium on Softw. Reliability Eng.* 230-238, Monterey, California, November 1994.

[45] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. "Effect of Test Set Minimization on Fault Detection Effectiveness." *Software Practice and Experience.* 28(4):347-369, April 1998.

[46] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini. "Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application." *Proc. of the 21st Int'l. Computer Softw. and Applications Conference.* 522-528, Washington, DC, August 1997.