Proceedings of the International Conference on Software Maintenance, Washington, D.C., November, 1998

An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites

Gregg Rothermel
Department of Computer Science
Oregon State University
Corvallis, OR
grother@cs.orst.edu

Mary Jean Harrold
Department of Computer and Information Science
Ohio State University
Columbus, OH
harrold@cis.ohio-state.edu

Jeffery Ostrin
Department of Computer Science
Oregon State University
Corvallis, OR
ostrinj@ucs.orst.edu

Christie Hong
Department of Computer and Information Science
Ohio State University
Columbus, OH
hongc@cis.ohio-state.edu

Abstract

Test suite minimization techniques attempt to reduce the cost of saving and reusing tests during software maintenance, by eliminating redundant tests from test suites. A potential drawback of these techniques is that in minimizing a test suite, they might reduce the ability of that test suite to reveal faults in the software. A recent study showed that minimization can reduce test suite size without significantly reducing the fault detection capabilities of test suites. To further investigate this issue, we performed an experiment in which we compared the costs and benefits of minimizing test suites of various sizes for several programs. In contrast to the previous study, our results reveal that the fault-detection capabilities of test suites can be severely compromised by minimization.

1 Introduction

Because test development is expensive, software developers often save the test suites they develop for their software, so that they can reuse those suites later as the software undergoes maintenance. As the software evolves, its test suites also evolve: new test cases are added to exercise new functionality or to maintain test adequacy. As a result, test suite size increases, and the costs of managing and using those test suites increases.

Therefore, researchers have investigated the notion that when several test cases in a test suite execute the same program components, that test suite can be reduced to a smaller suite that guarantees equivalent coverage. This research has resulted in the creation of various *test suite minimization* algorithms (e.g., [2, 7, 8]).

The motivation for test suite minimization is straightforward: by reducing test suite size we reduce the costs of executing, validating, and managing those test suites over future releases of the software. A potential drawback of minimization, however, is that in minimizing a test suite we may significantly alter the fault-detecting capabilities of that test suite. This tradeoff between the time required to execute, validate, and manage test suites, and the fault detection effectiveness of test suites, is central to any decision to employ test suite minimization.

A recent study [14] suggests that test suite minimization may produce dramatic savings in test suite size, at little cost to the fault-detection effectiveness of those test suites. To further explore this issue, we performed an experiment. Our experiment, however, revealed results quite different from those of the previous study. The following sections of this paper review the relevant literature, describe our experimental design, analysis, and results, and compare our results with previous results.

2 Test Suite Minimization Summary and Literature Review

2.1 Test suite minimization

The test suite minimization problem may be stated as follows [7, p. 272]:

> *Given*: Test suite TS, a set of test case requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_is such that any one of the test cases t_j belonging to T_i can be used to test r_i.
>
> *Problem*: Find a representative set of test cases from TS that satisfies all of the r_is.

The r_is in the foregoing statement can represent various test case requirements, such as source statements, decisions, definition-use associations, or specification items.

A representative set of test cases that satisfies all of the r_is must contain at least one test case from each T_i; such a set is called a *hitting set* of the group of sets $T_1, T_2, \ldots, T_n$. To achieve a maximum reduction, it is necessary to find the smallest representative set of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the T_is, and the problem of finding such a set is NP-complete [4]. Thus, minimization techniques resort to heuristics.

Several test suite minimization heuristics have been proposed (e.g., [2, 7, 8]); in this work we utilize the methodology of Harrold, Gupta, and Soffa [7].

2.2 Previous empirical work

A number of empirical studies of software testing have been performed. Some of these studies, such as those reported in References [3, 9, 13], provide only indirect data about the effects of test suite minimization through consideration of the effects of test suite size on costs and benefits of testing. Other studies, such as the study reported in Reference [5], provide only indirect data about the effects of test suite minimization through a comparison of regression test selection techniques that practice or do not practice minimization.[1]

A recent study by Wong, Horgan, London, and Mathur [14], however, directly examines the costs and benefits of test suite minimization. We refer to this study as the "WHLM" study; we summarize its results here.

The WHLM study involved ten common C UNIX utility programs, including nine programs ranging in size from 90 to 289 lines of code, and one program of 842 lines of code. For each of these programs, the researchers randomly generated an initial test case pool containing 1000 test cases. However, if any two test cases executed the same execution trace in a program, the researchers randomly excluded one of those test cases from the pool: the number of test cases in the resultant test pools varied between 61 and 862.

The researchers next generated multiple distinct test suites for the ten programs, by randomly selecting test cases from the associated test case pools. The researchers did not attempt to achieve any particular coverage of the code; measurements taken after the test suites were generated indicated that the test suites achieved block coverage ranging from 50% to 95%. Reference [14] reports test suite sizes in terms of averages over groups of test cases that achieved similar coverage: 89 test suites belonged to groups in which average test suite size ranged from 12 to 27 test cases, and 933 test suites belonged to groups in which average test suite size ranged from 1 to 7 test cases.

The researchers enlisted graduate students to inject simple mutation-like faults into each of the subject programs. The researchers excluded faults that could not be detected by any test case. All told, 183 faulty versions of the programs were retained for use in the study.

To assess the difficulty of detecting these faults, the researchers measured the percentages of test cases, in the associated test pools, that were able to detect the faults. Of the 183 faults, 75 (41%) were "Quartile I" faults detectable by fewer than 25% of the associated test cases, 39 (21%) were "Quartile II" faults detectable by between 25% and 50% of the associated test cases, 20 (11%) were "Quartile III" faults detectable by between 50% and 75% of the associated test cases, and 49 (27%) were "Quartile IV" faults detectable by at least 75% of the associated test cases.

The researchers minimized their test suites using ATACMIN [8], a heuristic-based minization tool that found "exact solutions for minimizations of all test suites examined" [14, page 42]. This minimization was done with respect to all-uses dataflow coverage (i.e., size was reduced while keeping all-uses coverage constant). The researchers measured the reduction in test suite size achieved through minimization, and the reduction in fault-detection effectiveness of the minimized test suites.

The researchers drew the following overall conclusions from the study:

- As the block coverage achieved by test suites increases, minimization produces greater savings with respect to those test suites, at rates ranging from 1.19% (for 50-55% block coverage suites) to 44.23% (for 90-95% block coverage suites).
- As the block coverage achieved by test suites increases, minimization produces greater losses in the fault-detection effectiveness of those suites. However, losses in fault detection effectiveness were small compared to savings in test suite size: those losses ranged from 0% (for 50-55% block coverage suites) to 1.45% (for 90-95% block coverage suites).
- Fault difficulty partially determined whether minimization caused losses in fault-detection effectiveness: effectiveness reductions for Quartile-I and Quartile-II faults were .39% and .66%, respectively, while effectiveness reductions for Quartile-III and Quartile-IV faults were .098% and 0%, respectively.

The authors caution that their observations are based on a single study and may not apply to programs and test suites

[1] Whereas minimization considers a program and test suite, regression test selection considers a program, test suite, and modified program version, and selects test cases that are appropriate for that version without removing them from the test suite. The problems of regression test selection and test suite minimization are thus related but distinct. For further discussion of regression test selection see Reference [12].

with different characteristics, and they emphasize the need for further studies. They then conclude:

> Data collected during experimentation have shown that when the size of a test set is reduced while the all-uses coverage is kept constant, there is little or no reduction in its fault detection effectiveness. This observation leads us to believe that test cases that do not add coverage to a test set are likely to be ineffective in detecting additional faults [14, page 50].

2.3 Open questions

The WHLM study leaves a number of open questions. Many of these questions concern the extent to which the results observed in that study generalize to other testing situations. Among the open questions are the following, which motivate the present work.

1. How does minimization fare in terms of costs and benefits when test suites have a wider range of sizes than the test suites utilized in the WHLM study?
2. How does minimization fare in terms of costs and benefits when test suites are coverage-adequate?
3. How does minimization fare in terms of costs and benefits when test suites contain additional coverage-redundant test cases, including multiple test cases that execute equivalent execution traces?

Test suites in practice often contain test cases designed not for code coverage, but rather, designed to exercise product features, specification items, or exceptional behaviors. Such test suites may contain larger numbers of test cases, and larger numbers of coverage-redundant and trace-redundant test cases, than those utilized in the WHLM study. It is important to understand the cost-benefit tradeoffs involved in minimizing such test suites.

3 The Experiment

3.1 Hypotheses

H1: Test suite minimization can produce savings on coverage-adequate, coverage-redundant test suites.

H2: As the size of such test suites increases, the savings associated with minimizing those test suites increases.

H3: Test suite minimization can compromise the fault detection effectiveness of coverage-adequate, coverage-redundant test suites.

H4: As the size of such test suites increases, the reduction in fault-detection effectiveness associated with test suite minimization increases.

3.2 Measures

To investigate our hypotheses we need to measure the costs and savings of test suite minimization.

3.2.1 Measuring savings.

Test suite minimization lets testers spend less time executing test cases, examining test results, and managing the data associated with testing. These savings in time are dependent on the extent to which minimization reduces test suite size. Thus, to measure the savings that can result from test suite minimization, we measure the percentage reduction in test suite size resulting from minimization, as given by $(\frac{|T|-|T_{min}|}{|T|} * 100)$.

This approach makes several assumptions: it assumes that all test cases have uniform costs, it does not differentiate between components of cost such as CPU time or human time, and it does not directly measure the compounding of savings that results from using the minimized test suites over a succession of subsequent releases. This approach, however, has the advantage of simplicity, and using it we can draw several conclusions that are independent of these assumptions.

3.2.2 Measuring costs.

There are two costs to consider with respect to test suite minimization.

The first cost is the cost of executing a minimization tool to produce the minimized test suite. However, a minimization tool can be run following the release of a product, automatically and during off-peak hours, and under this process the cost of running the tool may be noncritical. Moreover, having minimized a test suite, the cost of minimization is amortized over the uses of that suite on subsequent product releases, and thus assumes progressively less significance in relation to other costs.

The second cost to consider is more significant. Test suite minimization may discard some test cases that, if executed, would reveal defects in the software. Discarding these test cases reduces the fault detection effectiveness of the test suite. The cost of this reduced effectiveness may be compounded rather than amortized over uses of the test suite on subsequent product releases, and the effects of the missed faults may be critical. Thus, in this experiment, we focus on the costs associated with discarding *fault-revealing* test cases.

We considered two methods for calculating reductions in fault detection effectiveness.

On a per-test-case basis: One way to measure the cost of minimization in terms of effects on fault detection, given faulty program P and test suite T, is to identify the test cases in T that reveal a fault in P but are not in T_{min}. This

quantity can be normalized by the number of fault-revealing test cases in T. One problem with this approach is that multiple test cases may reveal a given fault. In this case some test cases could be discarded without reducing effectiveness; this measure penalizes such a decision.

On a per-test-suite basis: Another approach is to classify the results of test suite minimization, relative to a given fault in P, in one of three ways: (1) no test case in T is fault-revealing, and, thus, no test case in T_{min} is fault-revealing; (2) some test case in both T and T_{min} is fault-revealing; or (3) some test case in T is fault-revealing, but no test case in T_{min} is fault-revealing. Case 1 denotes situations in which T is inadequate. Case 2 indicates a use of minimization that does not reduce fault detection, and Case 3 captures situations in which minimization compromises fault detection.

For this experiment we base our measurements of cost on the second approach. For each program, our measure of reduced effectiveness is the percentage of faults for which T_{min} contains no fault-revealing test cases, but T does contain fault-revealing test cases. More precisely, if $|F|$ denotes the number of faults revealed by T over the faulty versions of program P, and $|F_{min}|$ denotes the number of faults revealed by T_{min} over those versions, the percentage reduction in fault-detection effectiveness of minimization for P, T, and T_{min} is given by $(\frac{|F|-|F_{min}|}{|F|} * 100)$.

Note that this method of measuring the cost of minimization calculates cost relative to a fixed set of faults. This approach also assumes that missed faults have equal costs, an assumption that typically does not hold in practice.

3.3 Experimental instrumentation

3.3.1 Programs.

We used seven C programs as subjects (see Table 1). Each program has a variety of versions, each containing one fault. Each program also has a large universe of inputs. These programs, versions, and inputs were assembled by researchers at Siemens Corporate Research for a study of the fault-detection capabilities of control-flow and data-flow coverage criteria [9]. We describe the other data in the table in the following paragraphs.

3.3.2 Faulty versions, test cases, and test suites.

The researchers at Siemens sought to study the fault-detecting effectiveness of coverage criteria. Therefore, they created faulty versions of the seven base programs by manually seeding those programs with faults, usually by modifying a single line of code in the program. In a few cases they modified between two and five lines of code. Their goal was to introduce faults that were as realistic as possible, based on their experience with real programs. Ten people performed the fault seeding, working "mostly without knowledge of each other's work" [9, p. 196].

Program	Lines of Code	No. of Versions	Test Pool Size	Description
totinfo	346	23	1052	information measure
schedule1	299	9	2650	priority scheduler
schedule2	297	10	2710	priority scheduler
tcas	138	41	1608	altitude separation
printtok1	402	7	4130	lexical analyzer
printtok2	483	10	4115	lexical analyzer
replace	516	32	5542	pattern replacement

Table 1. Experimental subjects.

For each base program, the researchers at Siemens created a large *test pool* containing possible test cases for the program. To populate these test pools, they first created an initial set of black-box test cases "according to good testing practices, based on the tester's understanding of the program's functionality and knowledge of special values and boundary points that are easily observable in the code" [9, p. 194], using the *category partition method* and the Siemens Test Specification Language tool [1, 11]. They then augmented this set with manually-created white-box test cases to ensure that each executable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. To obtain meaningful results with the seeded versions of the programs, the researchers retained only faults that were "neither too easy nor too hard to detect" [9, p. 196], which they defined as being detectable by at least three and at most 350 test cases in the test pool associated with each program.

Figure 1 shows the sensitivity to detection of the faults in the Siemens versions relative to the test pools; the boxplots[2] illustrate that the sensitivities of the faults vary within and between versions, but overall fall between .05% and 19.77%. Therefore, all of these faults were, in the terminology of the WHLM study, "Quadrant I" faults, detectable by fewer than 25% of the test pool inputs.

To investigate our hypotheses we required coverage-adequate test suites that exhibit redundancy in coverage, and we required these in a range of sizes. As an adequacy criteria we chose *edge coverage*: a coverage criteria similar to decision coverage but defined on control flow graphs.[3]

[2] A boxplot is a standard statistical device for representing data sets [10]. In these plots, each data set's distribution is represented by a box. The box's height spans the central 50% of the data and its upper and lower ends mark the upper and lower quartiles. The middle of the three horizontal lines within the box represents the median. The vertical lines attached to the box indicate the tails of the distribution.

[3] A test suite T is *edge-coverage adequate* for program P iff, for each edge e in each control flow graph for some procedure in P, if e is dynamically exercisable, then there exists at least one test case $t \in T$ that exercises e. A test case t *exercises* an edge $e = (n_1, n_2)$ in control flow graph G iff t causes execution of the statement associated with n_1, followed immediately by the statement associated with n_2.

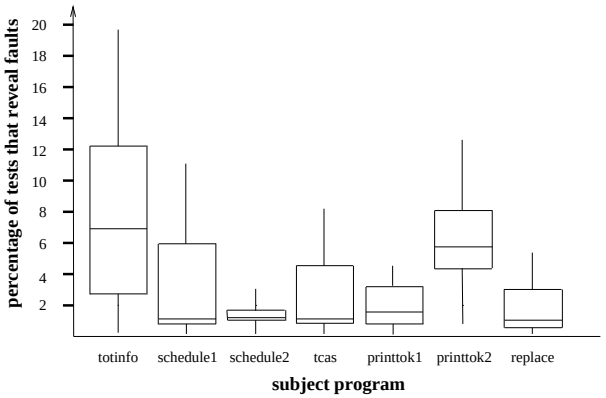

Figure 1. Boxplots that show, for each of the seven programs, the distribution, over the versions of that program, of the percentages of inputs in the test pools for the program that expose faults in that version.

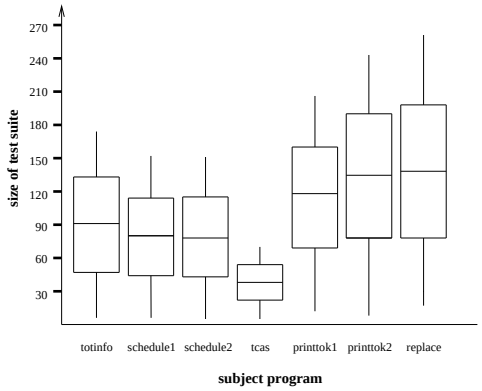

Figure 2. Boxplots that show, for each of the seven programs, the distribution of sizes amongst the unminimized test suites for that program.

We used the Siemens test pools to obtain edge-coverage adequate test suites for each subject program. Our test suites consist of a varying number of test cases selected randomly from the associated test pool, together with any additional test cases required to achieve 100% coverage of coverable edges.[4] We made no attempt to avoid adding test cases that achieve equivalent execution traces to test suites; thus, multiple test cases in the test suite for a program may traverse the same paths through that program. However, we did not add any particular test case to any particular test suite more than once. To ensure that these test suites would possess varying ranges of coverage redundancy, we randomly varied the number of randomly selected test cases over sizes ranging from 0 to .5 times the number of lines of code in the program. Altogether, we generated 1000 test suites for each program.

Figure 2 provides a view of the range of sizes of test suites created by the process just described. The boxplots illustrate that for each subject program, our test suite generation procedure yielded a universe of test suites of sizes that are relatively evenly distributed across the range of sizes utilized for that program.

Analysis of the fault-detection effectiveness of these test suites shows that, except for eight of the test suites for `schedule2`, every test suite revealed at least one fault in the set of faulty versions of the associated program. Thus, although each fault individually is difficult to detect relative to the entire test pool for the program, almost all (99.9%) of the test suites utilized in the study possessed at least some fault-detection effectiveness relative to the set of faulty programs utilized.

[4] To randomly select test cases from the test pools, we used the C pseudo-random-number generator "rand", seeded initially with the output of the C "time" system call, to obtain an integer which we treated as an index i into the test pool (modulo the size of that pool).

3.3.3 Test suite minimization tools.

To perform the experiments, we required a test suite minimization tool. We implemented the algorithm of Harrold, Gupta and Soffa [7] within the Aristotle program analysis system [6].

3.4 Experimental design

3.4.1 Variables.

The experiment manipulated two independent variables:

1. The subject program (7 programs, each with a variety of faulty versions).
2. Test suite size (between 0 and .5 times lines-of-code test cases randomly selected from the test pool, together with additional test cases as necessary to achieve 100% coverage of coverable edges).

We measured three dependent variables:

1. For each program P and test suite T, we measured the size of the minimized version of that suite, T_{min}.
2. On each run, with program P, faulty version P', and test suite T, whether one or more test cases in T reveals the fault in P'.
3. On each run, with program P, faulty version P', and test suite T_{min}, whether one or more test cases in T_{min} reveals the fault in P'.

3.4.2 Design.

This experiment uses a full-factorial design with 1000 repeated measures. That is, for each subject program, we created 1000 edge-coverage-adequate test suites from the test

pools. For each test suite, we then applied the minimization technique and evaluated the fault detection effectiveness of the original and minimized test suites.

3.4.3 Threats to validity.

In this section we discuss some of the potential threats to the validity of our studies.

Threats to internal validity are influences that can affect the dependent variables without the researcher's knowledge, and that thus affect any supposition of a causal relationship between the phenomena underlying the independent and dependent variables. In this study, our greatest concerns for internal validity involve the fact that we do not control for the structure of the subject programs or for the locality of program changes, and that we utilize only one type of test suite. To limit problems related to this, we run our test suite minimization algorithm on each test suite and each subject program.

Threats to external validity are conditions that limit our ability to generalize our results. The primary threats to external validity for this study concern the representativeness of the artifacts utilized. First, the subject programs, though nontrivial, are small, and larger programs may be subject to different cost-benefit tradeoffs. Also, there is exactly one seeded fault in every subject program; in practice, programs have much more complex error patterns. Furthermore, the faults were all deliberately chosen (by the Siemens researchers) to be faults that were neither too difficult nor too easy to detect. Finally, the test suites we utilized represent only one type of test suite that could appear in practice, if a mix of non-coverage-based and coverage-based testing were being utilized. These threats can only be addressed by additional studies, utilizing a greater range of artifacts.

Threats to construct validity arise when measurement instruments do not adequately capture the concepts they are supposed to measure. For example, in this experiment our measures of cost and effectiveness are very coarse: they treat all faults as equally severe, and all test cases as equally expensive.

4 Data and Analysis

Our analysis strategy has three steps. First, we summarize the data on test suite size reduction and fault-detection effectiveness reduction. Then we compare our results with the results of the WHLM study, and discuss factors that may be responsible for differences between the results. Finally, in Section 5, we summarize our results and make additional observations.

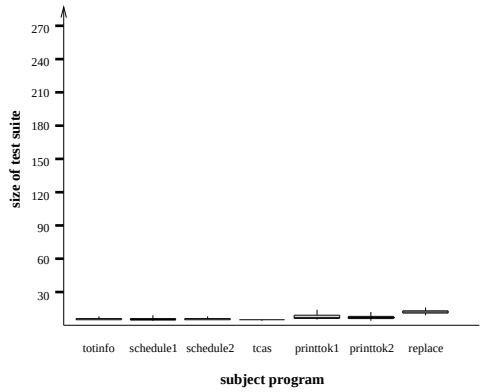

Figure 3. Boxplots that show, for each of the seven programs, the distribution of sizes amongst the minimized test suites for that program.

4.1 Test suite size reduction

Figure 3 depicts the sizes of the minimized test suites for the seven subject programs. As the figure shows, the minimized suites have average (median) sizes ranging from 5 (for `tcas`) to 12 (for `replace`) – between 87% and 95% smaller than the average sizes of the original test suites. The minimized suites for each program also demonstrate little variance in size: `tcas` showing the least variance (between 4 and 5 test cases), and `printtok1` showing the greatest (between 5 and 14 test cases). Minimized test suite size for a given program is thus, for our programs and minimization tool, relatively stable.

Figure 4 depicts the savings (percentage reduction in test suite size) produced by minimization in terms of the formula discussed in Section 3.2.1, for each of the subject programs. The data for each program P is represented by a scatterplot containing a point for each of the 1000 test suites utilized for P; each point shows the percentage size reduction achieved for a test suite versus the size of that test suite prior to minimization. Visual inspection of the plots indicates a sharp increase in test suite size reduction over the first quartile of test suite sizes, tapering off as size increases beyond the first quartile. The data gives the impression of fitting a logarithmic curve.

To verify the correctness of this impression, we performed logarithmic regression on the data depicted in these plots, and calculated the square of the Pearson product moment correlation coefficient, r^2.[5] Table 2 shows the regression results: they indicate a relatively strong logarithmic correlation between reduction in test suite size (savings of minimization) and initial test suite size.

[5] r^2 is a dimensionless index that ranges from zero to 1.0, inclusive, and reflects the extent of the correlation between two data sets. An r^2 value of 1.0 indicates a perfectly linear relationship with no noise, whereas an r^2 value of zero indicates a random relationship [10].

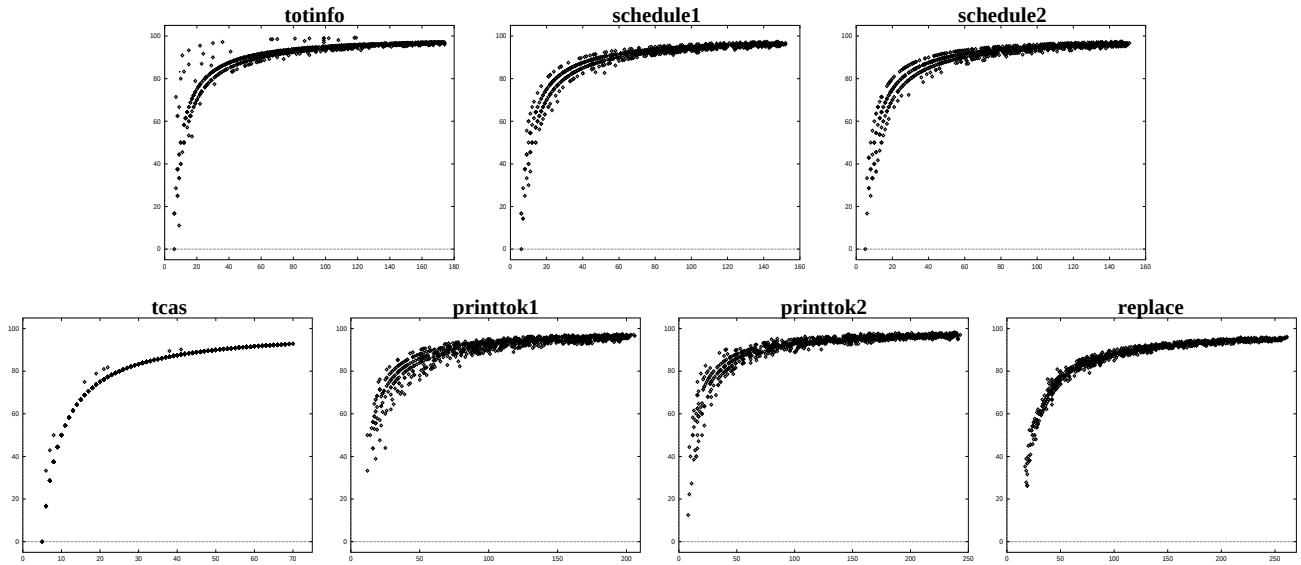

Figure 4. Percentage reduction in test suite size as a result of minimization, versus sizes of initial test suites. Horizontal axes denote the size of the initial test suites, and vertical axes denote the percentage reduction in test suite size.

program	regression equation	r^2
totinfo	$y = 13.82Ln(x) + 30.31$	0.75
schedule1	$y = 15.38Ln(x) + 24.20$	0.80
schedule2	$y = 15.20Ln(x) + 25.19$	0.80
tcas	$y = 24.31Ln(x) - 3.95$	0.88
printtok1	$y = 12.76Ln(x) + 31.69$	0.80
printtok2	$y = 12.11Ln(x) + 34.22$	0.78
replace	$y = 16.73Ln(x) + 7.07$	0.85

Table 2. Correlation between test suite size reduction and size of initial test suite.

Our experimental results support our hypothesis (H1) that test suite minimization can produce savings in test suite size on coverage-adequate, coverage-redundant test suites. The results also support our hypothesis (H2) that as test suite size increases, the savings produced by test suite minimization increase. Analysis of the data suggests that the increase in savings is logarithmic in the increase in test suite size. The savings increase at a rapid rate across (approximately) the first quartile of test suite sizes — a range that corresponds in our study to test suite sizes of up to one-eighth the number of statements in the program. Significantly, these results are relatively consistent across the seven subject programs, despite the differences in size, structure, and functionality among those programs.

4.2 Fault detection effectiveness reduction

Figure 5 depicts the cost (reduction in fault detection effectiveness) incurred by minimization, in terms of the formula discussed in Section 3.2.2, for each of the seven subject programs. The data for each program P is represented by a scatterplot containing a point for each of the 1000 test suites utilized for P; each point shows the percentage reduction in fault detection effectiveness observed for a test suite versus the size of that test suite prior to minimization.

The scatterplots show that the fault detection effectiveness of test suites can be severely compromised by minimization. For example, on `replace`, the largest of the subject programs, minimization reduces fault-detection effectiveness by over 50% for more than half of the test suites considered. Also, although in a small percentage of cases minimization does not reduce fault-detection effectiveness (e.g., on `printtok1`), there are many cases in which minimization reduces the fault-detection effectiveness of test suites by 100% (e.g., on `schedule2`).

Visual inspection of the plots suggests that reduction in fault detection effectiveness slightly increases as test suite size increases. Test suites in the smallest size ranges do produce effectiveness losses of less than 50% more frequently than they produce losses in excess of 50%, a situation not true of the larger test suites. Even the smallest test cases, however, exhibit effectiveness reductions in most cases: for example, on `replace`, test suites containing fewer than 50 test cases exhibit an average effectiveness reduction of nearly 40%, and few such suites do not lose effectiveness.

These results support our hypothesis (H3) that test suite minimization can compromise the fault-detection effectiveness of coverage-adequate, coverage-redundant test suites. However, the results only weakly support our hypothesis (H4) that as test suite size increases, the reduction in

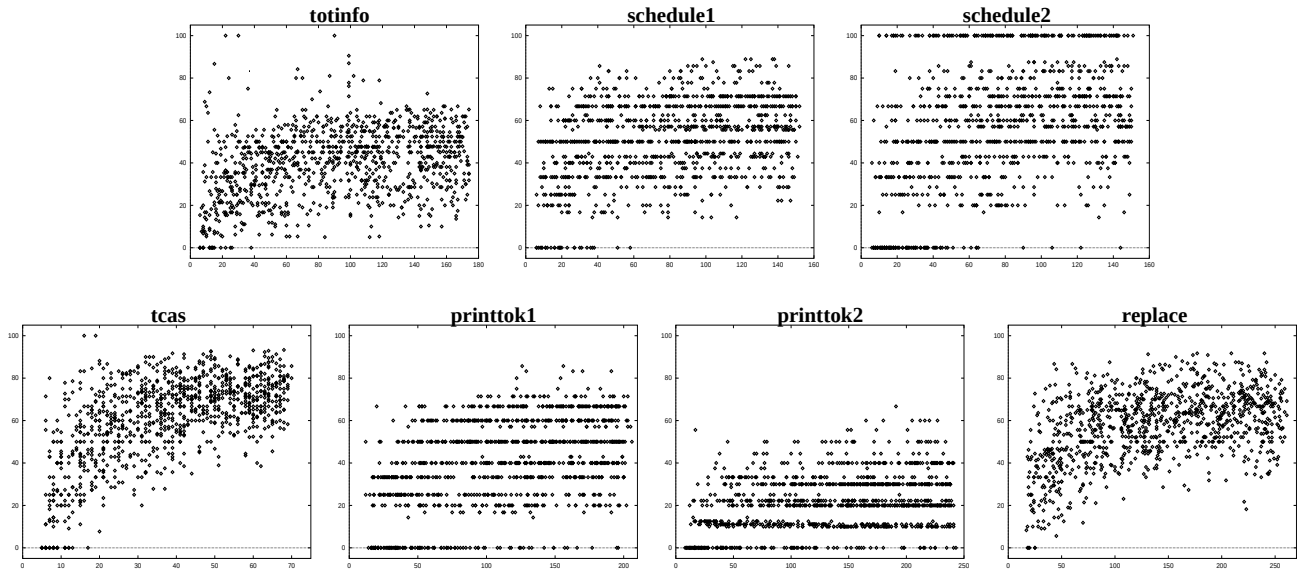

Figure 5. Percentage reduction in fault-detection effectiveness as a result of minimization, versus sizes of initial test suites. Horizontal axes denote the size of the initial test suites, and vertical axes denote the percentage reduction in fault-detection effectiveness.

the fault-detection effectiveness of those test suites will increase.

In contrast to the scatterplots showing size reduction effectiveness, the scatterplots showing reduction in fault detection effectiveness do not give a strong impression of closely fitting any curve or line: the data is much more scattered than the data for test suite size reduction. Our attempts to fit linear, logarithmic, and quadratic regression curves to the data validate this impression: the data in Table 3 reveals little linear, logarithmic, or quadratic correlation between reduction in fault detection effectiveness and initial test suite size.

One additional feature of the scatterplots of Figure 5 bears mentioning: on several of the graphs, there are markedly visible "horizontal lines" of points. In the graph for `printtok1`, for example, there are particularly strong horizontal lines at 0%, 20%, 25%, 33%, 40%, 50%, 60%, and 67%. Such lines indicate a tendency for minimization to exclude particular percentages of faults for the programs on which they occur.

This tendency is partially explained by our use of a discrete number of faults in each subject program. Given a test suite that exposes k faults, minimization can exclude test cases that detect between 0 and k of these faults, yielding discrete percentages of reductions in fault-detection effectiveness. For `printtok1`, for example, there are seven faults, of which the unminimized test suites may reveal between zero and seven. When minimization is applied to the test suites for `printtok1`, only 19 distinct percentages of fault detection effectiveness reduction can occur: 100%, 86%, 83%, 80%, 75%, 71%, 67%, 60%, 57%, 50%, 43%, 40%, 33%, 29%, 25%, 20%, 17%, 14%, and 0%. Each of these percentages except 29% and 100% is evident in the scatterplot for `printtok1`. With all points occurring on these 16 percentages, the appearance of lines in the graph is unsurprising.

It follows that as the number of faults utilized for a program increases, the presence of horizontal lines should decrease; this is easily verified by inspecting the graphs, considering in turn `printtok1` with 7 faults, `schedule1` with 9, `schedule2` with 10, `printtok2` with 10, `totinfo` with 23, `replace` with 32, and `tcas` with 41.

This explanation, however, is only partial: if it were complete, we would expect points to lie more equally among the various reduction percentages (with allowances for the fact that there may be multiple ways to achieve particular reduction percentages). The fact that the occurrences of reduction percentages are not thus distributed reflects, we believe, variance in fault locations across the programs, coupled with variance in test coverage patterns of faulty statements.

4.3 Comparison to previous empirical results

Both the WHLM study and our study support our hypotheses H1 and H2, that test suite minimization can produce savings (in test suite size reduction), and that these savings increase with test suite size. Both studies also support, to some degree, hypothesis H4, that reductions in fault-detection effectiveness increase as test suite size (or

program	regression line 1	r^2	regression line 2	r^2	regression line 3	r^2
totinfo	$y = 0.11x + 29.66$	0.11	$y = 8.68Ln(x) + 2.70$	0.16	$y = -0.002x^2 + 0.45x + 19.03$	0.16
schedule1	$y = 0.15x + 38.92$	0.12	$y = 10.03Ln(x) + 9.25$	0.15	$y = -0.002x^2 + 0.47x + 29.80$	0.15
schedule2	$y = 0.28x + 34.86$	0.16	$y = 17.70Ln(x) + 17.12$	0.20	$y = -0.004x^2 + 0.89x + 17.07$	0.21
tcas	$y = 0.68x + 34.89$	0.38	$y = 22.18Ln(x) - 16.23$	0.47	$y = -0.020x^2 + 2.18x + 13.41$	0.46
printtok1	$y = 0.16x + 22.48$	0.18	$y = 14.68Ln(x) + 26.34$	0.20	$y = -0.001x^2 + 0.44x + 10.94$	0.20
printtok2	$y = 0.07x + 12.44$	0.11	$y = 6.91Ln(x) + 11.13$	0.14	$y = -0.001x^2 + 0.19x + 6.84$	0.13
replace	$y = 0.11x + 42.67$	0.20	$y = 13.07Ln(x) + 4.82$	0.26	$y = -0.001x^2 + 0.41x + 26.79$	0.28

Table 3. Correlation between reduction in fault detection effectiveness and size of initial test suite.

in the case of the WHLM study, as percentage of block coverage achieved by test suites) increases.

The two studies differ substantially, however, with respect to hypothesis H3. The authors of the WHLM study conclude that, for the programs and test cases they studied: (1) test suites that do not add coverage are not likely to detect additional faults, and (2) effectiveness reduction is insignificant even for test suites that have high block coverage. This conclusion contrasts markedly with our results, and we would like to know the causes of this difference.

A precise answer to this question is outside of the scope of this work, as it requires a controlled comparative study. We suggest several potential causes.

The subject programs utilized by the two studies differed, and all but one of our programs were larger than all but one of the WHLM programs. However, all subject programs contained fewer than 1000 lines of code.

The WHLM study used ATAC for minimization, whereas our study utilized the algorithm of Reference [7]. Reference [14] reports that the ATAC approach achieved minimal test selection on the cases studied; we have not yet determined whether our algorithm was equally successful. However, if our algorithm is less successful than the algorithm used in the WHLM study, we would expect this to cause us to understate possible reductions in fault detection effectiveness.

Although both studies utilized seeded faults that may be accurately described as "mutation-like", all of the faults utilized in our study were Quartile I faults, whereas only 41% of the faults used in the WHLM study were Quartile I faults. Easily detected faults are less likely to go unrecognized in minimized test suites than faults that are more difficult to detect; thus, we would expect our results overall to show greater reductions in fault-detection effectiveness than the WHLM study. However, the authors of the WHLM study did separately report results for Quartile I faults, and in their study, minimized test suites missed few of these faults.

A more likely cause of differences stems from the fact that in the WHLM study, test suites were minimized for all-uses coverage, whereas our study minimized for all-edges coverage. All-uses coverage requires more test cases and greater code coverage overall than all-edges coverage. The authors of the WHLM study suggest that, had they minimized with respect to block coverage, "both the size and effectiveness reductions of the minimized test sets would have been substantially greater than for minimization with respect to all-uses coverage" [14, page 43].

In our opinion, the factor likely to be most responsible for differences in results in the two studies involves the types of test suites utilized. The WHLM study used test suites that were not coverage-adequate, and that were relatively small compared to our test suites. Overall, 933 of the 1022 test suites utilized in the WHLM study belonged to groups of test cases whose average sizes did not exceed 7 test cases. Small test suite size reduces opportunities both for minimization, and for reduction in fault-detection effectiveness. Further differences in test suites stem from the fact that the test pools used in the WHLM study as sources for test suites did not contain trace-redundant test cases, and did not necessarily contain any minimum number of test cases per covered item. These differences may contribute to reduced redundancy in test coverage within test suites, and reduce the likelihood that minimization will exclude fault-revealing test cases.

Further understanding of the factors that caused the differences in the results of the two studies would be useful; we leave this for future work.

5 Summary and Conclusions

As we discussed earlier, this experiment, like any other, has several limits to its validity. In particular, these limits include several threats to external validity that limit our ability to generalize our results; these threats can be addressed only by additional experimentation with a wider variety of programs, faults, and test suites. Keeping this in mind, we draw the following conclusions.

Our results strongly support our first three hypotheses: given test suites that are coverage-adequate and coverage-redundant, test suite minimization can provide significant savings in test suite size. These savings can increase as the size of the original test suites increases, and these savings

are relatively highly correlated (logarithmically) with test suite size. However, minimization can significantly compromise the fault-detection abilities of such test suites, even when the test suites are not particularly large.

Our results only marginally support our fourth hypothesis: degree of reduction in fault detection effectiveness does not markedly increase as test suite size increases. More importantly, we did not observe a close correlation between fault detection effectiveness and test suite size. Conceivably, the presence of discrete numbers of faults in our experimental subjects may have influenced this result: given an effectively infinite (or at least significantly larger) number of faults our results might have differed. Nevertheless, the results support an hypothesis that some factors other than test suite size play a significant role in loss of fault detection effectiveness.

A comparison of our results with those of the WHLM study reveals similarities in all but one important respect: the two studies differ substantially in terms of the costs in fault-detection effectiveness that they attribute to test suite minimization. We have suggested several possible reasons for this difference, and we are pursuing further studies to investigate the issue in greater depth. In particular, we will soon repeat our study using minimization for all-uses coverage.

Finally, of particular significance to testing practitioners is the fact that our results demonstrate no clear cost-benefit tradeoffs associated with the use of test suite minimization. In our study, as test suite size increased, the savings provided by minimization increased, but regardless of the level of savings provided, potential losses in fault-detection effectiveness varied widely. This is unfortunate, because as testers, we prefer situations in which the risks inherent in our testing processes relate measurably to the effort we put into those processes; in such cases we can balance those risks against the costs of our efforts. Thus, we would prefer that the risks inherent in reducing test suite size be related to to the potential for savings resulting from reducing test suite size. The results of this study suggest that such a relationship may not hold.

Given this fact, and the suggestion that some factor other than test suite size influences the reduction in fault-detection effectiveness that attends minimization, a possible conclusion is that we may not want to minimize test suites strictly in terms of code coverage. Alternative minimization strategies, whose cost-benefit tradeoffs are more clear, could be beneficial.

6 Acknowledgements

This work was supported in part by grants from Microsoft, Inc. to Ohio State University and Oregon State University, by National Science Foundation Faculty Early Career Development Award CCR-9703108 to Oregon State University, by National Science Foundation Award CCR-9707792 to Ohio State University and Oregon State University, and by National Science Foundation National Young Investigator Award CCR-9696157 to Ohio State University. Siemens Corporate Research supplied the subject programs. Jim Jones and Karen Rothermel assisted with the statistical analysis.

References

[1] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proc. of the 3rd Symp. on Softw. Testing, Analysis, and Verification*, pages 210–218, Dec. 1989.

[2] T. Chen and M. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, Mar. 1996.

[3] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Trans. on Softw. Eng.*, 19(8):774–787, Aug. 1993.

[4] M. Garey and D. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.

[5] T. Graves, M. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *The 20th Int'l. Conf. on Softw. Eng.*, Apr. 1998.

[6] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, Mar 1997.

[7] 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.

[8] J. Horgan and S. London. ATAC: A data flow coverage testing tool for C. In *Proc. of the Symp. on Assessment of Quality Softw. Dev. Tools*, pages 2–10, May 1992.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int'l. Conf. on Softw. Eng.*, pages 191–200, May 1994.

[10] R. Johnson. *Elementary Statistics*. Duxbury Press, Belmont, CA, sixth edition, 1992.

[11] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Comm. of the ACM*, 31(6), June 1988.

[12] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *IEEE Trans. on Softw. Eng.*, 22(8):529–551, Aug. 1996.

[13] 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. In *Proc. of the Fifth Intl. Symp. on Softw. Rel. Engr.*, pages 230–238, Nov. 1994.

[14] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th Int'l. Conf. on Softw. Eng.*, pages 41–50, Apr. 1995.