Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application*

W. Eric Wong, Joseph R. Horgan, Aditya P. Mathur and Alberto Pasquini

Abstract

An important question in software testing is whether it is reasonable to apply coverage based criteria as a filter to reduce the size of a test set. An empirical study was conducted using a test set minimization technique to explore the effect of reducing the size of a test set, while keeping block coverage constant, on the fault detection strength of the resulting minimized test set. Two types of test sets were examined. For those with respect to a fixed size, no test case screening was conducted during the generation, whereas for those with respect to a fixed coverage, each subsequent test case had to improve the overall coverage in order to be included. The study reveals that no matter how a test set is generated (with or without any test case screening) block minimized test sets have a size/effectiveness advantage, in terms of a significant reduction in test set size but with almost the same fault detection effectiveness, over the original non-minimized test sets.

Keywords: Block coverage, fault detection effectiveness, test set minimization, null hypothesis

*W. Eric Wong and and Joseph R. Horgan are Research Scientist and Research Director of the Software Environment Research Group at Bell Communications Research, Morristown, NJ 07960. Aditya P. Mathur is Professor of the Department of Computer Sciences and Director of the Software Engineering Research Center at Purdue University, W. Lafayette, IN 47907. Alberto Pasquini is with ENEA, Rome, Italy. All correspondence regarding this paper may be sent to W. Eric Wong (ewong@bellcore.com), 445 South Street, Bellcore, Morristown, NJ 07960.

Contents

List of Figures

List of Tables

1 Introduction

The utility of code coverage based testing criteria and how best to apply these is often of interest to a software tester. One question related to this utility is: "Whether or not it is reasonable to apply coverage-based criteria as a filter to reduce the size of a test set?" Results from an earlier study [14], which examined the correlation among code coverage, test set size and fault detection effectiveness, indicate that coverage obtained upon the execution of a program on a test set is a more significant determinant of the fault detection effectiveness than the size of the test set. This suggests that as the size of a test set is reduced, while the code coverage is kept constant, there may be only little or no reduction in the fault detection effectiveness of the new test sets so generated. This relationship was found to hold in a study [15] using ten Unix utility programs.

A test set may initially be large and contain redundant tests in that there are tests that, when generated, do not contribute to satisfying the testing criterion. For example, a test case generated in an attempt to cover all feasible blocks, may only cover blocks already covered. Such a test case is considered *ineffective* as it does not increase coverage beyond what has already been achieved. Eliminating all ineffective test cases might have a significant impact on the size of the test set. One possible advantage of including an ineffective test case is that it may possess some unknown "good quality" that the coverage criterion used ignores. For example, it may happen that two test cases t_1 and t_2 execute the same path, but that one of them, say t_1, detects certain faults while the other, t_2, does not. Removing t_1 from the test set will decrease the probability that a test set with a given coverage will detect these faults. Although the likelihood of generating such test cases exists, our earlier study shows that there is a small probability that they will occur. Most recent empirical studies in software testing, such as [7, 16], have eliminated the ineffective test cases before doing any analysis. For test sets so generated, minimization only pertains to the order in which test cases are executed and has a lesser impact on the test set size.

Test set minimization can thus be conceptually divided into two steps: (1) reduction of a "large" sized test set to a "moderate" sized test set by eliminating ineffective test cases and (2) from a moderate sized to a minimal test set using an exponential time algorithm or a heuristic. Although step 1 appears to be worthy of application, it is not clear whether step 2 should also be applied. We attempt to investigate this by determining the difference one should expect in the magnitude of reductions in test set size and fault detection effectiveness due to a method wherein only step 2 is applied compared with a method in which both steps are applied.

An application developed for the European Space Agency was selected for this study. The application is coded in the C language. The faults in this application were obtained from the error-

log maintained during its testing and integration into the remainder of the system. Two types of test sets were examined: one generated with respect to a fixed size and the other generated with respect to a fixed coverage. For those with respect to a fixed size, test cases were not screened during generation. For those with respect to a fixed coverage, each test case was required to improve the overall coverage in order to be included in the test set. Fixed-size test sets were generated so that we could study the effect of using both steps 1 and 2, as described above. Fixed-coverage test sets were generated so that we could study the effect of using step 2.

The remainder of this paper is organized as follows. The experimental methodology used is described in Section 2. A sample of data collected from the experiment appears in Section 3. Analysis of the results obtained is found in Section 4. The impact of the results reported herein on practices in software testing is discussed in Section 5.

2 Method

The following steps summarize the experimental methodology used.

1. *Program selection:* Select a subject program to be used in the experiments.
2. *Test set generation:* Generate test sets with respect to a fixed size or a fixed block coverage for the program selected in Step 1.
3. *Test set minimization:* Perform test set minimization with respect to the block coverage for the test sets generated in Step 2.
4. *Determination of the fault set and preparation of faulty versions:* Inject the faults that were recorded in the error-log back into the program. This results in a faulty version of the program.
5. *Size and effectiveness reduction:* Measure reduction in test set size and fault detection effectiveness between the test sets generated in Step 2 and their corresponding block minimized sets.
6. *Null hypothesis testing:* Perform hypothesis testing to determine whether or not the size/effectiveness advantages of block minimized test sets over their larger-sized non-minimized test sets were due to chance alone.

2.1 Program selection

An application developed for the European Space Agency in the C language was selected. This application provides a language-oriented user interface that allows the user to describe the configuration of an array of antennas using a high level language [1]. Its purpose is to prepare a data file

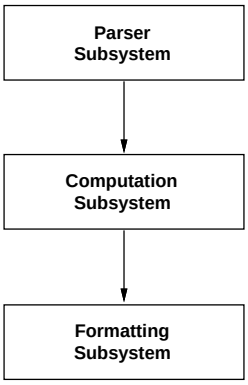

Figure 1: Program architecture

in accordance with a predefined format and characteristics from a user, given the array antenna configuration described in a language-like form. An appropriate Array Definition Language was defined and used within the program. This language allows the user to describe a certain antenna array by a few statements instead of having to write the complete list of elements, positions and excitations.

The application consists of about 10,000 lines of code (6,100 executable lines). It is divided into three subsystems as shown in Fig 1. The primary tasks of the Parser subsystem appear below.

- Perform syntactic analysis on input sentences.
- Set-up the internal data structure describing the array.

Main tasks of the Computation subsystems are given below.

- Assign or compute array parameters when omitted by the user.
- Execute consistency checks on user specified parameters.
- Compute position and excitation of each array element.

The primary task of the Formatting subsystem is to format the output data in the required format.

2.2 Test set generation

An operational profile, as formalized by Musa [8] and used in our experiment, is a set of the occurrence probabilities of various software functions. To obtain an operational profile for `space` we identified the possible functions of the program and generated a graph capturing the connectivity of these functions. Each node in the graph represented a function. Two nodes, A and B, were connected if control could flow from function A to function B. There was a unique start and end node representing functions at which execution began and terminated, respectively. A path through the graph from the start node to the end node represents one possible program execution. To estimate the occurrence probability of the software functions, each arc was assigned a transition

probability, i.e., the probability of control flowing between the nodes connected by the arc. For example, if node A is connected to nodes B, C and D, and the probabilities associated with arcs A-B, A-C, and A-D are, respectively, 0.3, 0.6 and 0.1, then after the execution of function A the program will perform functions B, C or D, respectively, with probability 0.3, 0.6, and 0.1. There was a total of 236 function nodes. Transition probabilities were determined by interviewing the program users.

Based on this profile, a test case pool with 1000 distinct test cases was created. Using this pool, we randomly generated multiple test sets with respect to a fixed size or a fixed block coverage. For fixed size, 30 distinct test sets of size 50, 100, 150 and 200 were generated, whereas 30 distinct test sets of 50%[1] 55%, 60%, 65%, 70% and 75% block coverage were generated for fixed coverage. Multiple test sets are necessary because a large number of test sets may satisfy a given size or a given block coverage for the program under test; therefore, selecting only one of these may possibly lead to false conclusions.

During the generation of test sets with fixed coverage, a test case was discarded if it could not cover at least one block which was not already covered. This guaranteed that each consequent test case was not redundant in terms of block coverage. However, it did not re-examine whether the previous test cases were still necessary, i.e., once a test case was included in a test set, it was not excluded due to the inclusion of any new test cases.

On the other hand, the only stopping criterion for generating a test set with a fixed size is the number of test cases in it. As long as its size is smaller than a pre-defined size, every test case randomly selected from the pool will be continuously added to the set without any screening. It is possible that such a test set contains different test cases that have executed the same path in the program.

2.3 Test set minimization

The purpose of minimizing a test set is to reduce the associated cost. This cost, computed as the sum of the costs of its test cases, can be measured in several ways. One measurement considers the computation time needed to execute each test case. Another measurement considers the tester time spent on constructing and analyzing these test cases. In this paper, the cost of a test set is the number of test cases in the set. Such a measure has also been used by other researchers [9, 11, 12, 13]. Based on this cost measure, a test set minimization procedure finds a minimal subset in terms of the number of test cases that preserve the coverage with respect to a certain coverage criterion of

[1] The actual block coverage of a test of 50% is in the range of [49.5%, 50.5%) where the notation '[' means inclusive and ')' means not. The same also applies to test sets of 55%, 60%, 65%, 70% and 75% block coverage.

the original test set. Hereafter, test set minimization with respect to the number of test cases, is referred to as *minimization*.

Test set size minimization is equivalent to the NP-complete "minimal set covering problem" [4]. A tool called `ATAC` [6] was used to find minimized test sets. `ATAC` uses an implicit enumeration algorithm with reductions to find optimal set covering. The technique used in `ATAC` finds exact solutions for minimizations of all tests examined. Although this technique is exponential on some "set covering problems" derived from Steiner triples [3], no problem derived from test set minimization has been appreciably more costly to solve than by inexact greedy heuristics.

Harrold *et al.* [5] present a heuristic for an approximate solution to the set covering problem in the context of finding a reduced size test set. A generalization of their algorithm is known as "greedy on bottlenecks" and is described in Zuev [17]. A simpler "greedy" algorithm for an approximate solution is described in Chvatal [2]. An upper bound on the cost of the solution is given as $\sum \frac{1}{n}$ for n from 1 to the size of the largest set times the cost of the optimal solution. `ATAC` provides options to select either of these "greedy" heuristics in the event that the exact solution is not obtained in reasonable time.

In Harrold's study, equal cost for each test is assumed. Such an assumption makes it impossible to include certain test cases in a minimized set. `ATAC`, however, allows different costs to be assigned to different test cases. If a test case has to be included in the minimized set, this can be done by assigning a zero cost to this test case so that it will certainly be selected because of the minimization. This feature is very important in practice as will be explained in Section 5.

In this study, test sets generated with respect to either a fixed size or a fixed block coverage, as explained in Section 2.2, are referred to as *non-minimized* test sets. Such non-minimized test sets were minimized with respect to the block coverage to obtain corresponding block minimized sets. These block minimized sets are minimal in terms of test cases in that it is impossible to remove any test case from them without reducing the overall block coverage. When test set minimization was conducted with respect to test sets with fixed coverage, it only pertained to the order in which test cases were executed. However, when test set minimization was conducted with respect to test sets with fixed size, a much larger size reduction was expected due to many redundant test cases in terms of coverage. Refer to Section 2.2 on how test sets were generated with respect to a fixed size or a fixed block coverage.

2.4 Fault set and erroneous program preparation

The fault set for the program used in this study was obtained from the error-log maintained during its testing and integration phase. For convenience, each fault has been numbered as **Fk** where the

integer **k** denotes the fault number. This number does not indicate the order in which faults were detected during experimentation. Eighteen faults, listed in the Appendix, were used. The line number and the file name where needed, together with the incorrect and correct strings, specify a fault uniquely. Both incorrect and correct strings may be empty, which means that some code was added or deleted to remove a fault. For example, a correction of fault F7 requires a statement `port_ptr→OMIT_POL = YES;` to be added at line 88 in file `fixselem.c`, whereas a correction of fault F9 requires an `if` statement to be deleted at line 25 in file `simpol.c`. All faults listed in the Appendix were injected one at a time, i.e., one erroneous program was created for each fault. Table 1 lists a brief classification, based on a scheme given elsewhere [10], of each fault and the number of test cases in the test case pool that detect it. Eight faults are in the "logic omitted or incorrect" category, seven faults belong to the type of "computational problems," and the remaining three faults have "data handling problems." Regarding the difficulty of each fault, five faults can be detected by less than 1% of the test cases, three faults by [1-2)%, one fault by [2-3)% and six faults by [3-4)%. Of the remaining three faults, one is in [4-5)%, another in [6-7)% and the last one (F15) is more than 32%. These percentages are computed based on the test case pool generated in Section 2.2 which has 1000 distinct test cases.

Table 1: Fault classification and the number of test cases to detect a fault

Fault No.	Fault classification		No. of detected test cases†
	Fault type	Subtype	
F1	Logic omitted or incorrect	Missing condition test	26
F2	Logic omitted or incorrect	Missing condition test	16
F3	Computational problems	Equation insufficient or incorrect	36
F4	Computational problems	Equation insufficient or incorrect	4
F5	Data handling problems	Data accessed or stored incorrectly	38
F6	Computational problems	Equation insufficient or incorrect	61
F7	Logic omitted or incorrect	Forgotten cases or steps	35
F8	Data handling problems	Data accessed or stored incorrectly	4
F9	Logic omitted or incorrect	Unnecessary function	18
F10	Computational problems	Equation insufficient or incorrect	32
F11	Computational problems	Equation insufficient or incorrect	32
F12	Logic omitted or incorrect	Missing condition test	6
F13	Logic omitted or incorrect	Checking wrong variable	46
F14	Logic omitted or incorrect	Missing condition test	37
F15	Logic omitted or incorrect	Checking wrong variable	320
F16	Computational problems	Equation insufficient or incorrect	6
F17	Data handling problems	Data accessed or stored incorrectly	14
F18	Computational problems	Equation insufficient or incorrect	2

†The test case pool has 1000 distinct test cases.

Two faults are considered to overlap when they share some code to be added when faults are removed. For example, faults F6 and F16 are overlapping because they occur in the same line even though they are two conceptually different faults. In the Appendix, the entry labelled (F16, not F6) lists the string at line 113 in file sgrphasr.c when fault F6 was removed while F16 was not. A similar explanation applies to entries (F6, not F16) and (not F16, not F6).

2.5 Size and effectiveness reduction

As explained in Section 2.4, for each fault, a single variant of the subject program was constructed. The program so constructed differs from the original program only due to the introduction of the fault. Since the program had been extensively used[2] without any new failures observed, it was assumed that it was fault-free and could serve as a base for fault detection. A test case is said to be able to detect a fault in one of the variant programs if the output of the variant program differs from the output of the original subject program. A test set detects the fault if one of its test cases does. The fault detection effectiveness of a test set for a program is the ratio of faults detected by the test set to all injected faults. Thus, the fault detection effectiveness of a test set depends on how well it distinguishes the behavior of faulty variant programs from the program under test. Hereafter, the fault detection effectiveness is referred to as *effectiveness*.

Reduction in fault detection effectiveness and test set size between a test set (T) and its corresponding block minimized set (T_m) is computed as follows:

$$(1 - \frac{\text{number of faults detected by } T_m}{\text{number of faults detected by } T}) * 100\% \quad (1)$$

$$(1 - \frac{\text{number of test cases in } T_m}{\text{number of test cases in } T}) * 100\% \quad (2)$$

2.6 Null hypothesis testing

Hypothesis testing was conducted to check whether or not the size/effectiveness advantages of block minimized test sets over their larger non-minimized test sets were due to chance. Ten distinct test sets, with the same size as the corresponding block minimized sets, were generated randomly from their respective non-minimized test sets. However, as some non-minimized test sets have the same size as the corresponding minimized test sets, it is not feasible to randomly generate any corresponding distinct test set. Furthermore, even if some of these randomly generated test sets can be generated from certain non-minimized test sets, they may introduce bias in favor of randomization if there are not enough different ways for such generation. Thus, for a fair comparison

[2]The number of times the program had been executed was at least on the order of 10^4.

between block minimized test sets and their randomly generated counterparts, it was decided that such a comparison would only be performed for programs that had a sufficient number of non-minimized test sets which allowed at least ten random test sets to be generated with the same size as the minimized test sets. Under this restriction, randomization was performed on test sets with 70% and 75% block coverage.

Nevertheless, this concern does not exist for block minimization with respect to test sets of size 50, 100, 150 and 200. This is because the size of each non-minimized test (say n) is significantly larger than the corresponding block minimized test set (say m) which guarantees substantially different ways for selecting m test cases from n test cases. More details can be found in Section 4.

3 Results

As discussed in Section 2.2, 30 distinct test sets of size 50, 100, 150 and 200 and of 50%, 55%, 60%, 65%, 70% and 75% block coverage were generated. Block minimization was performed on these test sets. Ten distinct random test sets with the same size as the corresponding block minimized sets were generated from their respective non-minimized test sets. The fault detection capability of each test set was measured. For the purpose of illustration, samples of the data set collected for test sets with 75% block coverage are shown in Table 2.[3]

4 Analysis

Table 3 lists the average number of faults detected by test sets with respect to a fixed size, or a fixed block coverage, their corresponding block minimized sets, and the average number of test cases in these sets. Reductions in fault detection effectiveness and test set size computed using Equations (1) and (2) in Section 2.5 are also included.

4.1 Effectiveness reduction

There is no effectiveness reduction for test sets of 50%, 55%, 60% and 65% block coverage. For test sets of low block coverage, there are not too many *redundant* test cases to be excluded because of the minimization. This is confirmed by examining the average test set size in Table 3. Therefore, it is not unusual to observe that block minimized test sets have the same fault detection effectiveness as that of the original non-minimized sets.

On the other hand, when there is a clear difference between the average size of non-minimized sets and that of the corresponding block minimized sets, the effectiveness reduction, in general,

[3] A complete data set may be obtained by sending electronic mail to W. Eric Wong at ewong@bellcore.com.

Table 2: Data collected for test sets with 75% block coverage

Test set number	Number of faults detected by		Number of test cases in	
	non-min sets	block min sets	non-min sets	block min sets
1	13	12	45	32
2	16	16	35	26
3	13	13	36	31
4	14	14	45	34
5	15	14	48	33
6	15	15	42	27
7	9	9	40	29
8	11	11	42	29
9	10	10	32	23
10	18	17	50	34
11	15	15	49	29
12	16	12	48	32
13	12	12	41	32
14	13	9	44	29
15	12	9	40	33
16	16	16	43	29
17	12	12	44	32
18	14	14	44	34
19	14	14	44	31
20	14	14	49	35
21	14	14	49	34
22	14	14	46	29
23	14	14	53	31
24	16	16	48	30
25	16	16	47	34
26	15	15	37	26
27	14	14	38	27
28	14	13	45	31
29	16	16	53	34
30	15	15	37	28

Test set number	Number of faults detected by										
	block min	random set-1	random set-2	random set-3	random set-4	random set-5	random set-6	random set-7	random set-8	random set-9	random set-10
1	12	10	13	12	12	13	12	10	10	12	13
2	16	14	7	14	13	12	16	16	9	14	11
3	13	13	13	13	13	13	12	12	13	13	13
4	14	12	14	11	14	12	8	10	8	14	12
5	14	14	15	15	14	14	15	13	10	13	15
6	15	15	13	11	13	14	12	15	15	13	15
7	9	9	3	9	9	7	9	9	9	9	7
8	11	11	11	6	9	10	11	9	11	10	11
9	10	10	10	9	10	10	10	9	10	10	7
10	17	16	17	18	14	14	17	16	18	17	14
11	15	11	14	11	9	13	13	11	12	10	15
12	12	16	11	10	13	15	13	16	16	14	12
13	12	10	12	12	11	12	12	11	12	8	9
14	9	13	12	8	13	13	7	8	12	8	11
15	9	11	12	9	12	12	9	12	9	12	12
16	16	14	9	11	16	12	12	8	14	12	9
17	12	11	11	7	11	12	11	11	11	11	11
18	14	12	10	13	13	13	14	13	13	14	13
19	14	10	14	12	13	14	12	12	13	13	14
20	14	13	14	13	13	14	14	14	14	13	13
21	14	12	14	10	12	12	12	10	14	13	12
22	14	10	14	12	11	14	13	12	14	11	9
23	14	13	13	9	13	12	13	14	14	12	13
24	16	11	9	15	14	15	13	13	11	14	13
25	16	14	9	9	16	12	12	16	10	14	16
26	15	12	10	12	11	14	10	11	9	8	13
27	14	12	14	14	14	14	12	14	10	14	11
28	13	11	13	9	12	14	13	9	11	13	10
29	16	16	15	13	13	13	13	13	14	14	14
30	15	15	14	13	15	15	14	13	15	15	12

† Each random set has the same size as the block minimized set.

Table 3: Effectiveness and size reductions due to block minimization

	Average number of faults detected by		Average effectiveness reduction (%)	Average number of test cases in		Avg. size reduction (%)
	non-min sets	block min sets		non-min sets	block min sets	
size-50	10.77	10.53	2.23	50.00	22.10	55.80
size-100	14.07	13.17	6.40	100.00	27.63	72.37
size-150	15.53	14.40	7.28	150.00	29.67	80.22
size-200	16.27	15.37	5.53	200.00	30.43	84.78
block-50	2.73	2.73	0.00	4.53	4.37	3.53
block-55	2.90	2.90	0.00	6.70	6.37	4.93
block-60	5.33	5.33	0.00	11.43	10.13	11.37
block-65	6.50	6.50	0.00	16.93	14.13	16.54
block-70	10.43	10.17	2.49	25.80	19.57	24.15
block-75	14.00	13.50	3.57	43.80	30.60	30.14

Table 4: Which faults are missed due to the block minimization†

Fault No.	Test sets with fixed size					Test sets with fixed block coverage		
	size-50	size-100	size-150	size-200	sum	block-70	block-75	sum
F1								
F2								
F3		2	2		4		3	3
F4								
F5								
F6								
F7	3	3		3	9	1		1
F8								
F9	3	8	4	5	20	3	5	8
F10		2	2		4		3	3
F11		2	2		4		3	3
F12								
F13		1			1			
F14								
F15								
F16	1	7	15	12	35	2		2
F17		2	7	5	14	2		2
F18			2	2	4		1	1
sum	7	27	34	27	95	8	15	23

† Each figure in this table represents the number of block minimized test sets that miss the fault while the original non-minimized sets do not. A blank entry indicates a figure 0.

increases as test set coverage or test set size increases. However, regardless of whether test sets are of fixed size or fixed coverage, the average effectiveness reduction due to block minimization is less than 7.28%. In fact, most of the reductions are less than 3.57%.

4.2 Missed faults

We now examine what causes the detection of a fault to be missed due to block minimization. With respect to each fault Table 4 lists the number of block minimized sets which miss it when the original non-minimized sets do not. Since test sets of 50%, 55%, 60% and 65% block coverage have no reduction in effectiveness, they are not included in this table. We observe that faults F7, F9, F16 and F17 are most likely to be missed. Among these faults, two belong to the category "logic omitted or incorrect," one has "computational problems" and the remaining one has "data handling problems." On the other hand, five of eight logic faults, two of seven computational faults, and two of three data handling faults are not missed by any block minimized set when the corresponding non-minimized set has also not missed it. Based on this information, it seems that the three fault types examined do not play a decisive role in determining whether or not a fault will be missed by the block minimized test sets. However, all of the likely candidates are difficult to detect faults. Three of them can be detected by no more than 2% of the test cases and the remaining one by no more than 3.5% of the test cases. The only easy fault (F15) that is detected by more than 32% of the test cases is not missed by any block minimized set if the corresponding non-minimized set detects it.

4.3 Size reduction

The size reductions for test sets with fixed size are much larger than those for test sets with fixed coverage. This is because of the way in which these test sets were generated. As explained in Section 2.2, no test case screening was conducted while generating test sets of size-50, 100, 150 and 200, whereas each subsequent test case had to cover at least one block which was not covered in order to be included in test sets of 50%, 55%, 60%, 65%, 70% or 75% block coverage. The result of this is that test sets with fixed size may contain many *ineffective* test cases from the coverage point of view because they execute the same path in the program. However, no matter how a test set is generated, with or without any test case screening, it appears that minimized test sets have a size advantage in terms of fewer test cases over the original non-minimized test sets.

For test sets with respect to different specified sizes, it is observed that the average size reduction for test sets of size 200 is larger than that for test sets of size 150. Test sets of a larger size have more test cases. Thus more of them are likely to be excluded due to minimization. Same observations

can also be made for test sets of size-100 and 50.

Similarly, since test sets with higher coverage normally have more test cases than those with lower coverage, it is not unusual to expect the former to have a larger size reduction than the latter. The experimental data support this expectation.

4.4 Randomization or minimization?

Hypothesis testing was performed only with respect to test sets of 70% and 75% block coverage as discussed in Section 2.6 and of size-50, 100, 150 and 200. Let e_1 be the effectiveness of a minimized set and e_2 be the effectiveness of a corresponding random set of equal size. The null hypothesis H_0, which is that $e_1 \leq e_2$, is tested against the alternative hypothesis H_1, which is that $e_1 > e_2$. The results indicate that H_0 can be rejected at the 0.0001 significance level in every case. They also provide statistical support to the argument that for test sets with high block coverage or big size, the block minimized test set has a size/effectiveness advantage over its random counterpart.

5 Discussion

Conceptually, test set minimization can be considered as a two-step procedure. In the first step one obtains a moderate-sized test set from a large-sized test set by eliminating *ineffective* test cases. In the second step one obtains a minimal test set from a moderate-sized test set using an exponential time algorithm or a heuristic. Test sets with respect to a fixed size or a fixed block coverage were used to simulate test set minimization using both steps or only the second step. Results obtained indicate that although fixed-size test sets may lead to a larger size reduction due to block minimization than fixed-coverage test sets because of the way they are generated, there is little or no reduction in fault detection effectiveness for both types of test sets.

The data obtained, and their analysis, strongly suggest that even if a test set is generated using screening to exclude any subsequently generated test case which does not increase the overall block coverage, its corresponding block minimized test set retains the size/effectiveness advantage over the corresponding non-minimized test set. This suggestion also holds for test sets generated without any test case screening.

As explained in Section 2.3, the minimization procedure used in `ATAC` guarantees that a test case is included in the resulting minimized test set by assigning it a cost of zero. In certain situations this feature of `ATAC` assumes special importance. For example, suppose that a test case t_1 is the only one that detects a fault F; then we definitely want to include t_1 in the minimized set if it appears in the original non-minimized set. Not doing so might cause the fault F to be missed due to

minimization because t_1 may or may not be in the minimized set. Assigning a zero cost to t_1 forces it to be included in the minimized set and avoids a reduction in effectiveness with respect to F.

Based on the results from this study, we suggest a procedure that can help lower the cost of regression testing. After one release of a software system, testers can apply test set minimization either to the entire regression test suite or only to regression tests which have been selected according to certain selective retest techniques. Those which are in the resulting minimized subsets should have a higher priority to be reused for validating the functionalities inherited from a previous version than those which are not. In this way, instead of reexecuting every regression test, which is usually not affordable in practice, testers can still conduct an efficient and effective regression testing by using a small set of those selected on the basis of minimization.

References

[1] A. Cancellieri and A. Giorgi, "Array preprocessor user manual," Technical Report IDS-RT94/052, 1994.

[2] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of Operations Research*, 4(3), August 1979.

[3] D. R. Fulkerson, G. L. Nemhauser, and L. E. Trotter Jr, "Two computational difficult set covering problems that arise in computing the 1-width of incidence matrices of steiner triple systems," *Mathematical Programming Study*, 2:72-81, 1974.

[4] M. R. Gary and D. S. Johnson, *"Computers and Intractability,"* Freeman, New York, 1979.

[5] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Transactions on Software Engineering and Methodology*, 2(3):270-285, July 1993.

[6] J. R. Horgan and S. A. London, "ATAC: A data flow coverage testing tool for C," in *Proceedings of Symposium on Assessment of Quality Software Development Tools*, pp 2-10, New Orleans, LA, May 1992.

[7] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the Sixteenth International Conference on Software Engineering*, pp 191-200, Sorrento, Italy, May 1993.

[8] J. D. Musa, "Operational profiles in software reliability engineering," *IEEE Software*, 10(2):14-32, March 1993.

[9] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. on Software Engineering*, 14(6):868-874, June 1988.

[10] IEEE Computer Society, "IEEE 1044 - standard classification for software errors, faults and failures," *IEEE Computer Society*, 1994.

[11] K. C. Tai, "Program testing complexity and test criteria," *IEEE Trans. on Software Engineering*, SE-6(6):531-538, November 1980.

[12] E. J. Weyuker, "The cost of data flow testing: An empirical study," *IEEE Trans. on Software Engineering*, 16(2):121-127, February 1990.

[13] E. J. Weyuker, "More experience with data flow testing," *IEEE Trans. on Software Engineering*, 19(9):912-919, September 1993.

[14] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set size and block coverage on fault detection effectiveness," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pp 230-238, Monterey, CA, November 1994.

[15] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th IEEE International Conference on Software Engineering*, pp 41-50, Seattle, WA, April 1995.

[16] W. E. Wong and A. P. Mathur, "Fault detection effectiveness of mutation and data flow testing," *Software Quality Journal*, 4(1):69-83, March 1995.

[17] Y. A. Zuev, "A set-covering problem: The combinatorial-local approach and the branch and bound method," *U.S.S.R Computational Mathematics and Mathematical Physics*, 19(6):217-226, June 1979.

Appendix: Faults considered in this study

Fault	File	Line	Incorrect string	Correct string

F1	sgramp2n.c	122	a = ((p1_et + p2_et - 2 * e)/(2 * x1 * x1));
			c = ((a * x1 * x1 + e - p1_et)/x1);
			b = ((q1_et + q2_et - 2 * e)/(2 * y1 * y1));
			d = ((b * y1 * y1 + e - q1_et)/y1);
				if (x1 == 0)
				{a = c = 0;}
				else {a = ((p1_et + p2_et - 2 * e) / (2 * x1 * x1));
				c = ((a * x1 * x1 + e - p1_et) / x1);
				};
				if (y1 == 0)
				{b = d = 0;}
				else {b = ((q1_et + q2_et - 2 * e) / (2 * y1 * y1));
				d = ((b * y1 * y1 + e - q1_et) / y1);
				};

F2	sgrpha2n.c	123	a = ((p1_ep + p2_ep - 2 * e) / (2 * x1 * x1));
			c = ((a * x1 * x1 + e - p1_ep) / x1) ;
			b = ((q1_ep + q2_ep - 2 * e) / (2 * y1 * y1));
			d = ((b * y1 * y1 + e - q1_ep) / y1) ;
				if (x1 == 0)
				{a = c = 0;}
				else {a = ((p1_ep + p2_ep - 2 * e) / (2 * x1 * x1));
				c = ((a * x1 * x1 + e - p1_ep) / x1);
				};
				if (y1 == 0)
				{b = d = 0;}
				else {b = ((q1_ep + q2_ep - 2 * e) / (2 * y1 * y1));
				d = ((b * y1 * y1 + e - q1_ep) / y1);
				};

F3	mkshex.c	84	x = P[i] * pstep + Q[i] * qstep * cos(angle); y = Q[i] * qstep * sin(angle);	x = P[i] * pstep + Q[i] * qstep * dcos(angle); y = Q[i] * qstep * dsin(angle);
F4	sgrrot.c	42	XE = ((XD - XC) * cos(phi)) - ((YD - YC) * sin(phi)) + XC ; YE = ((XD - XC) * sin(phi)) + ((YD - YC) * cos(phi)) + YC ;	XE = ((XD - XC) * dcos(phi)) - ((YD - YC) * dsin(phi)) + XC ; YE = ((XD - XC) * dsin(phi)) + ((YD - YC) * dcos(phi)) + YC ;
F5	unifamp.c	40	GetKeyword(Keywords[88], curr_ptr);	error = (GetKeyword(Keywords[88], curr_ptr));
F6†	gnodevis.c	29	(gnode_ptr->GEOMPORT_PTR)->PPA += phase;	(gnode_ptr->GEOMPORT_PTR)->PPA = phase;
F7	fixselem.c	88	Missing code	port_ptr->OMIT_POL = YES;
F8	sgrrot.c	54	app_ptr->PSEA += phi;	app_ptr->PHEA += phi;
F9	simpol.c	25	if ((group_ptr->ELEM_PTR)->POLARIZATION != LIN_POL) return 0;	
F10	seqrothg.c	49	can = angle_step;	can += angle_step;
F11	seqrothg.c	42	cph = phase_step;	cph += phase_step;

F12	seqrotrg.c	52	Missing code	if ((pmin == pmax) && (qmin == qmax)) {
				gnodevis(pmin, qmin, cph, can, g);
				cph += phase_step;
				can += angle_step;
				cont++;
				if (cont == nodes_num)
				endvisit = 1;
				}
				else {

		121	Missing code	};
F13	pqlimits.c	28	while (app_ptr->NEXT != NULL) {	while (app_ptr != NULL) {

F14	fixsgrel.c	88	Missing code	if ((group_ptr->GRPHAEXC_PTR)->TYPE ==
				ROTATION_SEQUENTIAL_LAW) {
				grid->PSTEP = elem->RADIUS;
				grid->QSTEP = elem->RADIUS;
				}
				else {

		103	Missing code	}

†See Section 2.4 for a discussion of faults F6 and F16.

Fault	File	Line	Incorrect string	Correct string
F14 (cont'd)	fixsgrel.c	112	Missing code	if ((group_ptr->GRPHAEXC_PTR)->TYPE == ROTATION_SEQUENTIAL_LAW) { if (grid->PSTEP < elem->RADIUS) printf("n%s%s", MOSErrors[12], group_ptr->NAME); } else {
		125	Missing code	}
		139	Missing code	if ((group_ptr->GRPHAEXC_PTR)->TYPE == ROTATION_SEQUENTIAL_LAW) { grid->PSTEP = elem->RADIUS; grid->QSTEP = elem->RADIUS; } else {
		152	Missing code	}
		164	Missing code	if ((group_ptr->GRPHAEXC_PTR)->TYPE == ROTATION_SEQUENTIAL_LAW) { if (grid->PSTEP < elem->RADIUS) printf("n%s%s", MOSErrors[12], group_ptr->NAME); } else {
		177	Missing code	}
F15	mksblock.c	68	for (q = q1; q <= q2; q++) { for (p = p1; p <= p2; p++) {	for (q = intmin(q1,q2); q <=intmax(q1,q2); q++) { for (p = intmin(p1,p2); p <=intmax(p1,p2); p++) {
F16, not F6	sgrphasr.c	121		(group_ptr->ELEM_PTR)->PORT_PTR->PSH = (geomnode_app_ptr->GEOMPORT_PTR)->PSH; (group_ptr->ELEM_PTR)->PORT_PTR->PSC = (geomnode_app_ptr->GEOMPORT_PTR)->PSC; (group_ptr->ELEM_PTR)->PORT_PTR->PPA += (geomnode_app_ptr->GEOMPORT_PTR)->PPA;
F6, not F16	sgrphasr.c	130		(geomnode_app_ptr->GEOMPORT_PTR)->PSH = (group_ptr->ELEM_PTR)->PORT_PTR->PSH; (geomnode_app_ptr->GEOMPORT_PTR)->PSC = (group_ptr->ELEM_PTR)->PORT_PTR->PSC; (geomnode_app_ptr->GEOMPORT_PTR)->PPA = (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
not F16, not F6	sgrphasr.c	138		(geomnode_app_ptr->GEOMPORT_PTR)->PSH = (group_ptr->ELEM_PTR)->PORT_PTR->PSH; (geomnode_app_ptr->GEOMPORT_PTR)->PSC = (group_ptr->ELEM_PTR)->PORT_PTR->PSC; (geomnode_app_ptr->GEOMPORT_PTR)->PPA += (group_ptr->ELEM_PTR)->PORT_PTR->PPA;
F17	kwdsinit.c	101	strcpy(Keywords [84],"P1_ET"); strcpy(Keywords [85],"P2_ET"); strcpy(Keywords [86],"Q1_ET"); strcpy(Keywords [87],"Q2_ET");	strcpy(Keywords [84],"P1_VAL"); strcpy(Keywords [85],"P2_VAL"); strcpy(Keywords [86],"Q1_VAL"); strcpy(Keywords [87],"Q2_VAL");
		115	strcpy(Keywords [89],"P1_EP"); strcpy(Keywords [90],"P2_EP"); strcpy(Keywords [91],"Q1_EP"); strcpy(Keywords [92],"Q2_EP");	strcpy(Keywords [89],"P1_VAL"); strcpy(Keywords [90],"P2_VAL"); strcpy(Keywords [91],"Q1_VAL"); strcpy(Keywords [92],"Q2_VAL");
F18	gnodevis.c	38	gnode_ptr->PHEA = angle;	gnode_ptr->PHEA += angle;

†See Section 2.4 for a discussion of faults F6 and F16.