Effect of Test Set Size and Block Coverage on the Fault Detection Effectiveness *

W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur

April 27, 1994

Abstract

Size and code coverage are two important attributes that characterize a set of tests. When a program P is executed on elements of a test set T, we can observe the fault detecting capacity of T for P. We can also observe the degree to which T induces code coverage on P according to some coverage criterion. We would like to know whether it is the size of T or the coverage of T on P which determines the fault revealing effectiveness of T for P. In an earlier study, we found that there is little or no reduction in the fault detection effectiveness of a test set when its size is reduced while keeping the all-uses coverage fixed. These data suggest, indirectly, that coverage is more correlated than the size with the fault detection effectiveness. To further investigate this suggestion, we report here an empirical study to compare the statistical correlation between (1) fault detection effectiveness and coverage, and (2) fault detection effectiveness and the size. Results from our experiments indicate that the correlation between effectiveness and block coverage is higher than that between effectiveness and size.

Keywords: Block coverage, fault detection effectiveness, correlation coefficient, test set size

* W. Eric Wong is with Hughes Network Systems, Germantown, MD 20876. Joseph R. Horgan and Saul London are with Bell Communications Research, Morristown, NJ 07962. Aditya P. Mathur is with the Software Engineering Research Center, Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907.

Contents

List of Figures

List of Tables

1 Introduction

Random testing is a long standing testing technique and many researchers have studied its fault detection effectiveness [6, 7, 12, 17, 18]. The results of these studies are diverse. Some researchers [4, 6, 7, 13] conclude that random testing can be used to replace coverage based testing such as data flow and mutation testing. They make such conclusions based on the advantages of random testing such as reduced cost, high coverage in branch testing, and high coverage in mutation testing. Other researchers [14] tend to reject random testing for its poor fault detection capability with respect to certain types of faults such as the boundary value and loop termination conditions.

In random testing, one generates test cases randomly in accordance with some input distribution. In coverage based testing, one generates test cases to increase some form of coverage. The stopping criteria for random testing are based on statistical principles [6, 7, 12, 13]. For coverage based testing, the coverage criterion provides a stopping rule. A test set generated using random testing is likely to contain test cases that do not improve the coverage of interest. Depending on the stopping rule used, the size of a randomly generated test set might also be larger than that generated using coverage based testing. In both cases, a tester is interested in generating a test set which reveals hidden faults in the program. An interesting question arises when we consider the size and coverage of a test set as two of its attributes. How are these two attributes related to the fault detection capability of a test set?

An answer to this question is indicative of the relative importance of these two attributes. If test sets generated by random testing are assumed to be larger than those generated using coverage based testing for the same program, then an answer to the above question directs our confidence to one of these testing methods. In this paper we report experiments designed to investigate the above question.

Another motivation for the experiments reported here came from our results [19] that showed little or no reduction in the fault detection effectiveness of a test set when its size is reduced while keeping the all-uses coverage fixed. These data suggest, indirectly, that coverage is more correlated than the number of test cases with the fault detection effectiveness. To further investigate this suggestion, a study to compare the statistical correlation between (1) fault detection effectiveness and coverage, and (2) fault detection effectiveness and the number of test cases is necessary. Hereafter, we refer to the number of test cases in a test set as its *size*.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the block coverage, fault detection effectiveness, and various correlation coefficients used in our experiments. Our experimental methodology is described in detail in Section 3. Data collected from experiments and resulting analyses appear in Section 4. Section 5 explains how a practicing tester can benefit from our study. Conclusions and on-going work are presented in Section 6.

2 Basic concepts and terminology

In this section we review the notions of block coverage, fault detection effectiveness, and the Spearman, Kendall, and Pearson correlation coefficients required for an understanding of the rest of the paper. Let P denote a program under test with D as its input domain. A test case t is a sequence of values input to P during one execution of P. A test set $T \subseteq D$ consists of one or more test cases on which P is executed during testing.

2.1 Block coverage

A block is a sequence of consecutive statements or expressions containing no branches except at the end, so that if one element of it is executed all are. A block is feasible if there exists a test case $t \in D$ such that t running on P executes this block. A block of dead code, for example, is an infeasible block. A test set T may be evaluated against the block coverage criterion by computing the ratio of the number of blocks covered to the total number of blocks. A ratio of unity implies that T is fully adequate with respect to this criterion. Full adequacy is rare in practice because of the presence of infeasible blocks. Determining whether a block is infeasible is in general undecidable. More details of the block criterion may be found in [3, 15].

2.2 Fault detection effectiveness

In the experiments reported below, we consider $E = \{ e_i \mid 1 \leq i \leq n \}$ as a set of possible faults to be injected into P. For each $e_i \in E$, a P_i' is constructed by injecting e_i into P. A test set T is said to be able to detect e_i in P_i' if there exists a test case $t \in T$ such that P_i' behaves differently from P when executed against t. We define the fault detection effectiveness of T in terms of P and E as:

$$\Psi_{P,E}(T) = \frac{\text{number of faults in } E \text{ detected by } T \text{ when injected into } P}{\text{total number of faults in } E} * 100\% \qquad (1)$$

Clearly, the fault detection effectiveness of T depends on how well it distinguishes the behavior of P and P_i' for the faults in E. In general, it is impossible to determine the fault detection effectiveness for all programs with an arbitrary set of faults. Hereafter, we refer to fault detection effectiveness as *effectiveness*.

2.3 Correlation Coefficient

We used different correlation coefficients to measure the correlation between two variables. Such correlation coefficients are useful because they are designed to indicate how closely two variables move together. Below we present an overview of different coefficients used in our analysis with only the necessary details. More of these coefficients can be found in [10, 16, 20].

1. Spearman Rank Correlation Coefficient: γ

   This coefficient was the earliest to be developed and is perhaps the most well studied among statistics based on rank. It requires both variables to be measured on an ordinal scale so that every subject can be ranked in two ordered series. The coefficient γ is computed using

   $$\frac{\Sigma\ (\mathrm{x_i} - \overline{\mathrm{x}})\ (\mathrm{y_i} - \overline{\mathrm{y}})}{\sqrt{\Sigma\ (\mathrm{x_i} - \overline{\mathrm{x}})^2\ \Sigma\ (\mathrm{y_i} - \overline{\mathrm{y}})^2}} \tag{2}$$

   where x_i and y_i are the ranks of the ith x and y values, respectively, and $\overline{x}$ and $\overline{y}$ are the means of the x_i and y_i values, respectively. In case of ties, averaged ranks are used. The index i varies from 1 to n, n being the number of subjects.

2. Kendall Rank Correlation Coefficient: τ

   This coefficient requires the same type of data as γ does. One may regard τ as a function of the minimum number of interchanges required between neighbors to transform one rank into another. The coefficient τ is measured as:

   $$\frac{\Sigma_{i<j}\ \mathrm{sgn}(\mathrm{x_i} - \mathrm{x_j})\ \mathrm{sgn}(\mathrm{y_i} - \mathrm{y_j})}{\sqrt{(\mathrm{T_0} - \mathrm{T_x})}\ \ \sqrt{(\mathrm{T_0} - \mathrm{T_y})}} \tag{3}$$

   where $T_0 = \frac{n(n-1)}{2}$, n being the number of subjects; $T_x = \Sigma\frac{t_i(t_i-1)}{2}$ and $T_y = \Sigma\frac{u_i(u_i-1)}{2}$, t_i and u_i being the number of tied x and y values in the ith group of tied x and y values, respectively. The function $sgn(z) = 1$ if z is greater than 0, 0 if z is equal to 0, and -1 if z is less than 0.

   Although τ and γ have different underlying scales and numerically are not directly comparable to each other, both use the same amount of information for a given set of data and reject the null hypothesis[1] at the same level of significance.

[1] Null hypothesis: two variables under study are independent.

3. Partial Rank Correlation Coefficient: $\tau_{xy.z}$ and $\gamma_{xy.z}$

   When correlation is measured between two variables, it is possible that this correlation is due to the correlation between each of these two variables and a third variable. For example, the correlation between the effectiveness and the block coverage of a given test set in our study may not reflect the real correlation between these two variables. Instead, it may be the result of two other pairs of correlation: (1) the effectiveness and the number of test cases, and (2) the block coverage and the number of test cases. One way to overcome this problem is to measure the partial rank correlation coefficients. In such a measure, the effects of varying a third variable on the correlation between two given variables are eliminated by keeping the third variable constant while measuring the coefficient between the two given variables. For example, the partial rank correlation coefficient between the effectiveness and the block coverage of a given test set is measured by keeping the number of test cases constant. The partial correlation coefficients $\tau_{xy.z}$ and $\gamma_{xy.z}$ are measured as:

$$\frac{\zeta_{xy} - \zeta_{xz}\ \zeta_{yz}}{\sqrt{(1-\zeta^2_{\mathrm{xz}})(1-\zeta^2_{\mathrm{yz}})}} \tag{4}$$

   where ζ_{xy}, ζ_{xz}, and ζ_{yz} are the appropriate Spearman or Kendall correlation coefficients.

4. Pearson Correlation Coefficient: ρ

   Given two variables, the Pearson correlation measures the extent of a linear relationship between them. If there exists a perfect positive linear relation between these two variables, ρ has the maximal value of $+1$. If the linear relation is perfect negative, then ρ has the minimal value of -1. Since ρ is a measure of linearity only, a zero value for ρ does not necessarily mean these two variables are independent. It only means that there is no linear relation between them. The following equation shows the formula to compute ρ.

$$\frac{\mathrm{cov(x,\ y)}}{\sqrt{\mathrm{var(x)\ var(y)}}} \tag{5}$$

   where $\mathrm{var}(x)$ and $\mathrm{var}(y)$ are the variance of x and y variables, respectively, and $\mathrm{cov}(x, y)$ is the covariance of x and y.

One distinction between these coefficients is that Pearson uses the *values* of the variables while others use the *ranks* of the variables.

SAS procedure `CORR`

We used the SAS procedure `CORR` [10] to compute the above correlation coefficients between (1) fault detection effectiveness and coverage, and (2) fault detection effectiveness and the number of test cases for each subject program.

3 Experimental methodology

We used the tool `ATAC` [8, 9] in our experiments. `ATAC` is a data flow coverage measurement tool for C programs. Given a program and a test set, `ATAC` can compute the block coverage. The sequence of steps used in our experiments is given below; details follow in subsequent sections. It is important to note that the injection of faults in subject programs was completely independent of the generation of test sets for the programs.

Step 1: Prepare subject programs.
Step 2: Construct test case pools.
Step 3: Generate test sets of fixed size.
Step 5: Inject faults in subject programs.
Step 6: Compute block coverage and fault detection effectiveness of test sets .
Step 7: Compute correlation coefficients.
Step 8: Analyze data.

3.1 Program selection and preparation

A suite of ten `C` programs described in Table 1 was selected. Together, these ten programs represent 2310 lines of C code. One virtue of these programs in experimentation is that since they have been so thoroughly used, they serve as reliable oracles in evaluating the behavior of fault injected programs derived from them. Moreover, they are unlikely to have naturally occurring faults; thus the failure during execution of a derived fault injected program may be attributed to the injected fault with great confidence.

3.2 Test set generation

For each program, a test case pool of 1000 test cases was generated quasi-randomly in conformance with the specifications of the program. The technique was to construct a generator from the Unix specifications of the program. Where input data were required, as for the `Sort` program, they were both generated and gathered from existing files. Then random strings meeting the input signature of the program were generated from the data and the specifications.

Table 1: Characteristics of subject programs

Program	Objective†
`Cal`	Print a calendar for a specified year or month
`Checkeq`	Report missing or unbalanced delimiters and .EQ/.EN pairs
`Col`	Filter reverse paper motions from nroff output for display on a terminal
`Comm`	Select or reject lines common to two sorted files
`Crypt`	Encrypt and decrypt a file using a user supplied password
`Look`	Find words in the system dictionary or lines in a sorted list
`Sort`	Sort and merge files
`Spline`	Interpolate smooth curve based on given data
`Tr`	Translate characters
`Uniq`	Report or remove adjacent duplicate lines

†Details can be found in the `Unix` manual.

If test cases t_1 and t_2 executed the same path, only one of these was selected randomly for inclusion in the pool. Table 2 lists the size and maximal cumulative data flow coverage for each test case pool. However, full coverage was seldom achieved because the test cases were generally not manually tuned to achieve high coverage. Furthermore, there was no effort (beyond the heuristics in `ATAC`) to eliminate infeasible blocks, decisions, or data flow objects.

Based on these pools, multiple distinct test sets of size n, $2 \leq n \leq 10$, were generated for each program. Duplicate test cases were removed from each test set. Table 3 lists the number of distinct test sets generated for each program. Although 60 test sets of size n, $2 \leq n \leq 10$, were constructed for each program, some of them are duplicates. An example of this occurs in `Cal` for which we found three duplicate test set pairs. Since only one test set from each duplicate pair was selected, there were 57, instead of 60, distinct test sets of size 2 for `Cal`. Figure 1 shows the sequence of steps used for test set generation. A few characteristics pertinent to our test set generation are listed below.

(1) All test cases were generated quasi-randomly.

(2) All test cases were generated *before* any fault detection experiment was conducted. This was done to avoid test cases aimed specifically at a certain type of fault.

(3) Since a large number of test sets may satisfy a given size for a given program, selecting only one of these may possibly lead to false conclusions. To assure the validity of our experiments we attempted to generate multiple test sets for each size.

Table 2: Characteristics of test case pool

Program	number of test cases	Maximal cumulative data flow coverage				
		block %	decision %	c-use %	p-use %	all-uses %
`Cal`	65	100.00	100.00	93.62	91.86	92.78
`Checkeq`	61	100.00	86.96	81.48	75.28	78.24
`Col`	100	87.66	87.50	80.53	80.98	80.75
`Comm`	482	100.00	86.96	98.15	82.14	90.00
`Crypt`	77	89.86	71.79	94.34	83.33	88.50
`Look`	100	96.59	84.62	96.15	89.66	92.73
`Sort`	862	94.09	83.25	79.95	74.74	77.53
`Spline`	230	98.96	94.21	87.89	85.88	87.00
`Tr`	247	95.05	84.15	80.43	64.74	70.56
`Uniq`	325	98.81	94.83	95.31	98.33	96.77

Table 3: Number of distinct test sets of various size

Function	Size 2	Size 3	Size 4	Size 5	Size 6	Size 7	Size 8	Size 9	Size 10	Σ
`Cal`	57	60	60	60	60	60	60	60	60	537
`Checkeq`	58	60	60	60	60	60	60	60	60	538
`Col`	59	60	60	60	60	60	60	60	60	539
`Comm`	60	60	60	60	60	60	60	60	60	540
`Crypt`	59	60	60	60	60	60	60	60	60	539
`Look`	59	60	60	60	60	60	60	60	60	539
`Sort`	60	60	60	60	60	60	60	60	60	540
`Spline`	60	60	60	60	60	60	60	60	60	540
`Tr`	60	60	60	60	60	60	60	60	60	540
`Uniq`	60	60	60	60	60	60	60	60	60	540
Σ	592	600	600	600	600	600	600	600	600	5392

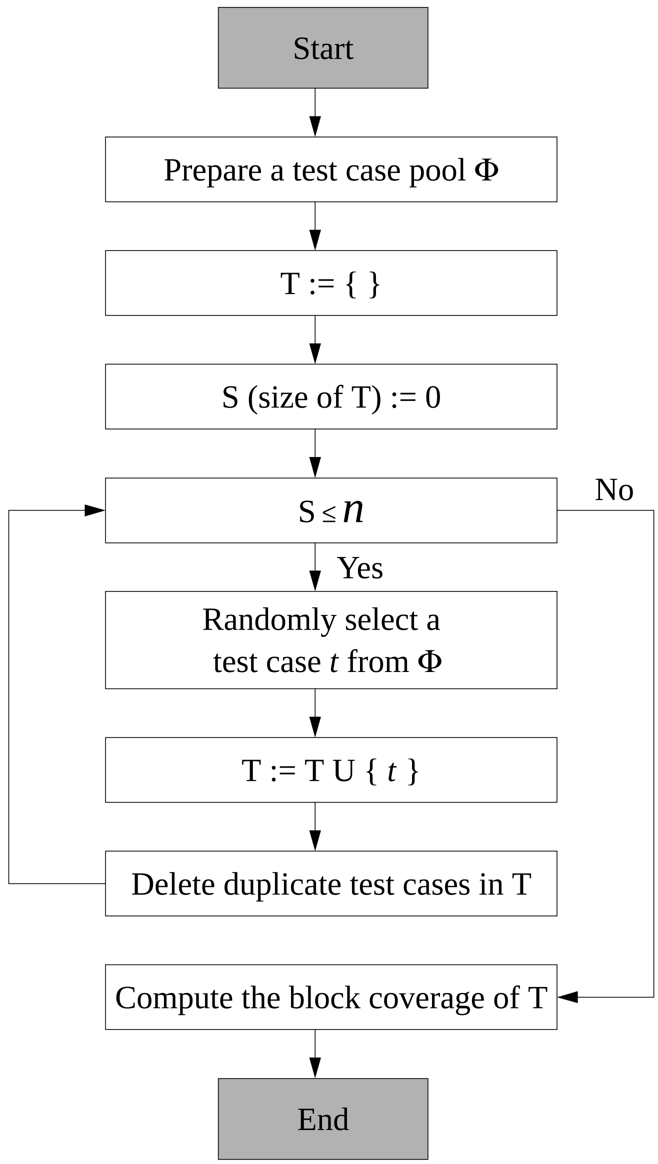

Figure 1: Procedure for generating test sets of size n, $2 \leq n \leq 10$.

3.3 Fault injection and detection

Common fault types described in [1, 2, 5, 11] served as the basis for our experiments. For each of the ten programs in our suite, a graduate student[2] injected the fault so as to produce syntactically correct programs. Each student was instructed to use experience and judgment to inject one or more faults of each type listed in Table 4. As discussed in Section 2.2, one erroneous program was created for each fault. The number of erroneous programs created for a given program equals the number of faults selected for injection into that program. To make the injection process more objective, students were required to work independently. Faults that could not be detected by any test case from its corresponding test case pool were excluded from the study. The number of faults examined in each program appears in Table 6. Listings of these faults can be found in [19].

By excluding faults that were not detectable by the test case pools, we have probably eliminated extremely difficult faults from our investigation. These difficult faults may represent the faults that persist in field deployed software. However, these were not the faults we were concerned with. The kinds of faults that we hoped to characterize by our fault injection method were those that the programmer might encounter during thorough unit or multi-unit testing. Such faults are likely to be more readily excited than the secretive faults only detectable in field use.

Although the faults we have selected may be representative of faults found in unit testing, the single fault seeding method is artificial. One might expect a fault density of between four and 40 faults in one thousand lines of code before a program is unit tested. Therefore, if we were to model the natural unit testing process for the 842 line `Sort` program, all 25 faults (see Table 6) should be seeded in a single erroneous program. The reasons we did not follow this path are practical. Single fault programs are easier to run and control than are multiple fault programs. What is more, if a test case fails on a multiply fault seeded program, it is extremely difficult to determine which of the faults produced the failure, and, therefore, it is difficult to determine which faults are detected. Finally, we feel that the testing-failure-debugging cycle is fairly represented by the single fault seeded programs. If test case t_i can detect multiple faults in a multiply fault seeded program, it can do so only as debugging eliminates and testing reveals faults one at a time. Thus the testing of singly seeded faults is a fair representation of the test-failure-debug cycle.

Since all ten programs in our suite had been extensively used, we assumed that these

[2]All participating students were from the Department of Computer Science, Purdue University and had at least three years of programming experience in C.

Table 4: Description of fault types

missing path faults	
incorrect predicate faults	relational operator replacement
	logical operator replacement
incorrect computation statement	incorrect initialization
	incorrect constant
	incorrect precedence
	incorrect array element reference
	incorrect pointer operation
	same type variable replacement
	arithmetic operator replacement
	miscellaneous
missing computation statement	delete a complete statement
	delete a part of a statement
incorrect number of loop iterations	
missing clause in predicates	

programs were fault-free and could serve as oracles for fault detection. The set of faults detected by a test set is the union of the sets of faults detected by its member test cases. Thus, to compute the set of faults detected by a test set we only needed to determine faults detected by each test case. For example: given a test set T with three test cases, t_1, t_2, and t_3 which detect faults $\{e_1, e_2\}$, $\{e_2, e_3\}$, and $\{e_3, e_4\}$, respectively, T is said to be able to detect faults $\{e_1, e_2, e_3, e_4\}$. After the faults detected by a test set were determined, its fault detection effectiveness was computed using Equation (1).

4 Experimental Results and Analysis

Tables 5 and 6 list the number of lines, blocks, decisions, and faults examined in each of the ten programs studied. These metrics serve as indicators of the relative complexity of programs considered in our experiments. Among these programs, `Sort` and `Crypt` are, respectively, the largest and smallest programs, with 500 and 69 blocks of code.

4.1 Comparing correlation coefficients

Various correlation coefficients computed for each program listed in Table 1 are presented in Table 7. From our experimental data and the summary in this table, we make the following observations:

Table 5: Program size metrics[†]

Program	LOC	# of blocks	# of decisions
`Cal`	163	96	50
`Checkeq`	90	74	69
`Col`	274	154	104
`Comm`	144	100	69
`Crypt`	121	69	39
`Look`	135	88	52
`Sort`	842	508	394
`Spline`	289	193	121
`Tr`	127	101	82
`Uniq`	125	84	58
Average	231	146.7	103.8

[†]LOC (lines of code) excluding comment and declaration lines. All other metrics were computed by ATAC.

Table 6: Number of faults examined in each program

Program	# of faults
`Cal`	20
`Checkeq`	20
`Col`	29
`Comm`	15
`Crypt`	17
`Look`	13
`Sort`	25
`Spline`	14
`Tr`	12
`Uniq`	18
Σ	183

- For all ten programs, the Spearman and Pearson coefficients between effectiveness and block coverage are higher than that between effectiveness and size.

- In nine out of ten programs, the Kendall, Spearman partial, and Kendall partial coefficients between effectiveness and block coverage are higher than that between effectiveness and size.

- Program `Comm` is the only program whose Kendall, Spearman partial, and Kendall partial coefficients between effectiveness and block coverage are lower than that between effectiveness and size. However, in all these cases, effectiveness and block coverage is only slightly[3] less correlated than effectiveness and size.

- In six out of ten programs, `Cal`, `Col`, `Crypt`, `Sort`, `Spline`, and `Uniq`, the Pearson coefficient between effectiveness and block coverage is greater than 0.83 which suggests some kind of linear relationship between these two variables. An example of this appears in Figure 2. (See Appendix A for other figures.) From this figure, we observe that effectiveness is linear in block coverage from 70% to 90%. On the other hand, the same correlation coefficient between effectiveness and size is less than 0.68 for all ten programs.

The above observations indicate that effectiveness and block coverage are more correlated than effectiveness and size.

4.2 Why Are Some Partial Correlation Coefficients Negative?

From Table 7, we find that the Spearman partial rank correlation coefficients between the size and the effectiveness are negative for programs `Sort` and `Spline`. Since the effectiveness increases with size, negative coefficients appear to be logically invalid. A careful examination of Equation (4) indicates that such negative values arise because the product of ζ_{xz} and ζ_{yz} is greater than ζ_{xy} with x, y, and z being size, effectiveness, and coverage, respectively. Hence, rather than violate our intuition, such negative values strongly support our claim that coverage is more correlated to effectiveness than size.

5 Practical Implications

In this section we answer how the results reported here can be used in practice. Since our results are from a single case study, we caution that more experiments are necessary to further strengthen the following conclusions.

[3]The difference between these two coefficients is ≤ 0.02.

Table 7: Correlation between size, effectiveness, and coverage§

Program	Correlation coefficient			Partial correlation coefficient	
	Spearman	Kendall	Pearson	Spearman	Kendall
`Cal`	(0.59 : 0.73)	(0.49 : 0.63)	(0.57 : 0.90)	(0.21 : 0.56)	(0.24 : 0.51)
`Checkeq`	(0.60 : 0.63)	(0.49 : 0.52)	(0.55 : 0.77)	(0.34 : 0.42)	(0.32 : 0.38)
`Col`	(0.63 : 0.84)	(0.51 : 0.73)	(0.63 : 0.85)	(0.41 : 0.77)	(0.34 : 0.67)
`Comm`	(0.65 : 0.65)	(0.53 : 0.52)	(0.64 : 0.71)	(0.31 : 0.29)	(0.30 : 0.29)
`Crypt`	(0.54 : 0.99)	(0.46 : 0.93)	(0.52 : 0.83)	(0.09 : 0.98)	(0.13 : 0.91)
`Look`	(0.33 : 0.46)	(0.26 : 0.36)	(0.32 : 0.47)	(0.00 : 0.35)	(0.07 : 0.28)
`Sort`	(0.59 : 0.87)	(0.45 : 0.71)	(0.59 : 0.85)	(-0.20 : 0.81) †	(0.07 : 0.62)
`Spline`	(0.44 : 0.77)	(0.33 : 0.62)	(0.44 : 0.92)	(-0.17 : 0.72) †	(0.02 : 0.56)
`Tr`	(0.57 : 0.68)	(0.44 : 0.56)	(0.58 : 0.67)	(0.24 : 0.50)	(0.21 : 0.43)
`Uniq`	(0.67 : 0.84)	(0.53 : 0.69)	(0.68 : 0.83)	(0.22 : 0.69)	(0.25 : 0.57)

§In entry $(a : b)$, a is the coefficient between size and effectiveness and b is the coefficient between coverage and effectiveness.

†See Section 4.2 for explanations.

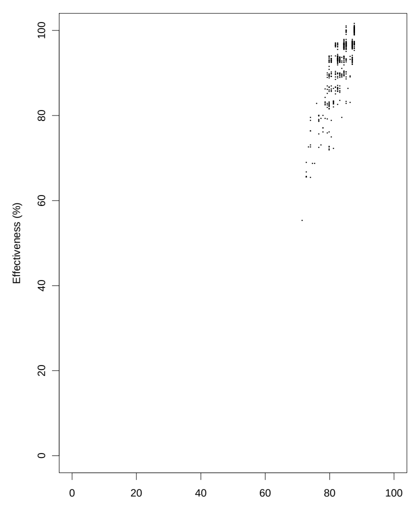

Figure 2: A scatter plot of effectiveness versus block coverage for the 539 test sets of `Col`.

Table 8: Average effectiveness (%) of test sets with fixed size

Program	Size-2	Size-3	Size-4	Size-5	Size-6	Size-7	Size-8	Size-9	Size-10
`Cal`	72.11	80.00	83.92	86.25	89.08	90.08	90.08	91.50	90.75
`Checkeq`	82.93	87.83	91.25	92.08	93.00	95.92	95.75	95.83	96.33
`Col`	79.37	89.31	90.29	92.41	94.83	95.57	96.21	96.78	96.84
`Comm`	45.11	53.33	60.22	64.89	66.67	70.00	68.67	69.33	71.56
`Crypt`	86.64	90.69	93.73	97.65	99.12	99.12	99.61	100.00	100.00
`Look`	42.37	43.72	49.10	51.67	72.44	67.18	68.21	71.16	70.00
`Sort`	23.60	33.33	36.13	39.47	42.27	46.93	50.60	53.47	56.33
`Spline`	29.88	44.05	48.09	52.97	70.59	69.88	68.81	72.26	74.28
`Tr`	32.08	42.22	49.44	55.83	59.31	63.75	68.19	68.33	71.81
`Uniq`	52.59	66.94	72.78	78.33	80.92	87.22	89.35	90.74	91.39

Effectiveness and size

Intuitively, the effectiveness of a test set increases as its size increases. Our experimental data in Table 8 support such an intuition. However, as shown in Figure 3, we observe a wide variation in the effectiveness for test sets with a small size. (See Appendix B for other figures.) Such variation becomes less significant for test sets with larger sizes. For example, the effectiveness variation among test sets of size 2 for `Cal` is more than that among tests sets of size 10. In addition, we find that although the effectiveness tends to increase as the size increases, this does not necessarily mean that a test set with a smaller size must be less effective in detecting faults than a test set with a larger size. For example, for `Cal`, some test sets of size 6 are more effective than some test sets of size 10.

Block coverage and size

Intuitively, the block coverage of a test set increases with size. Our experimental data in Table 9 support such an intuition. We also observe a wide variation in the block coverage for test sets with the same size. As shown in Figure 4, although block coverage tends to increase with size, this does not necessarily mean that a test set with a smaller size must have a lower block coverage than a test set with a larger size. (See Appendix C for other figures.) For example, some test sets of size 2 have higher block coverage than some test sets of size 5.

Effectiveness and block coverage

Intuitively, the effectiveness of a test set increases as its block coverage increases. Our experimental data in Table 10 support such an intuition. However, as shown in Figure 5, there exists

Figure 3: Effectiveness (%) of test sets of size 2, 6, and 10 for `Cal`.

Table 9: Average block coverage (%) of test sets with fixed size

Program	Size-2	Size-3	Size-4	Size-5	Size-6	Size-7	Size-8	Size-9	Size-10
`Cal`	66.87	76.95	80.09	82.02	85.44	86.96	86.53	88.91	88.49
`Checkeq`	86.56	90.27	92.30	92.52	94.21	95.47	95.23	96.76	96.17
`Col`	78.63	82.01	83.31	84.11	84.76	85.09	85.35	85.50	85.71
`Comm`	74.52	81.13	83.85	86.05	87.35	88.32	89.12	89.18	89.98
`Crypt`	84.87	86.67	87.78	88.87	89.40	89.45	89.72	89.86	89.86
`Look`	78.56	79.96	82.42	85.25	86.29	86.40	86.93	87.48	87.59
`Sort`	46.81	54.87	59.86	62.49	64.10	68.63	70.54	72.02	73.11
`Spline`	66.28	75.45	78.90	81.06	88.07	89.23	88.33	90.99	91.41
`Tr`	81.97	86.47	89.52	90.30	92.11	93.83	93.45	94.13	94.01
`Uniq`	73.71	79.54	81.53	85.04	86.85	88.24	90.32	90.44	92.26

Figure 4: A scatter plot of block coverage versus size for the 540 test sets of `Sort`.

Table 10: Average effectiveness (%) of test sets with fixed block coverage

Program	(50 – 55)	(60 – 65)	(70 – 75)	(80 – 85)	(90 – 95)
`Cal`	57.86	64.50	73.79	87.41	91.67
`Checkeq`	35.00	46.00	63.33	76.75	91.38
`Col`	NA	44.83	66.96	87.40	NA
`Comm`	23.53	37.78	37.93	54.44	71.78
`Crypt`	NA	NA	50.80	81.57	NA
`Look`	7.69	14.20	20.74	38.98	82.57
`Sort`	24.00	36.40	54.14	71.31	NA
`Spline`	7.14	7.14	11.26	60.24	75.00
`Tr`	NA	8.33	16.11	33.06	50.83
`Uniq`	22.22	40.12	55.37	67.22	92.04

NA: *not available.*

Figure 5: Effectiveness (%) of test sets with (50-55), (60-65), (70-75), (80-85), and (90-95) block coverage (%) on `Cal`.

6 Conclusions and future work

Data collected during experimentation have shown that the correlation between effectiveness and block coverage is higher than that between effectiveness and size. In another study [19], we showed that when the size of a test set is reduced while the coverage is kept fixed, there is little or no reduction in test fault detection effectiveness. These two results lead us to believe that test cases, unless with some special characteristics, that do not add coverage to a test set are likely to be ineffective in detecting faults. Thus, a randomly generated test set which is "boiled down" to preserve its coverage is likely to be as effective as originally at less cost.

We believe the results of this study support the thesis that coverage based testing has a cost/benefit advantage over random testing, although our conclusions may need to be tempered by the relatively narrow scope of our experiment. A similar study on naturally occurring faults in large and varied programs is on-going which will provide more confident conclusions.

Acknowledgments

We express our thanks to Hiralal Agrawal, Rich DeMillo, Stuart Feldman, Elaine Keramidas, Edward Krauser, Tom Kuczek, Vernon Rego, and Teresa Ruesgen for discussions on data analysis, and Frank Oreovicz for reviewing an earlier draft of this paper.

References

[1] T. A. Budd, *"Mutation Analysis of Program Test Data,"* PhD thesis, Yale University, New Haven, CT, 1980.

[2] T. A. Budd, "Mutation analysis: Ideas, examples, problems and prospect," in *Computer Program Testing*, B. Chandrasekaran and S. Radicchi, Eds. Amsterdam, North Holland, July 1981.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A formal evaluation of data flow path selection criteria," *IEEE Trans. on Software Engineering*, 15(11):1318–1332, November 1989.

[4] P. Currit, M. Dyer, and H. Mills, "Certifying the reliability of software," *IEEE Trans. on Software Engineering*, SE-12(1):3–11, January 1986.

[5] 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," in *Thirteenth Minnowbrook Workshop on Software Engineering*, July 1990.

[6] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. on Software Engineering*, SE-10(7):438–444, July 1984.

[7] R. G. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. on Software Engineering*, 16(12):1402–1411, December 1990.

[8] J. R. Horgan and S. A. London, "Data flow coverage and the C language," in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991.

[9] 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*, pages 2–10, New Orelean, LA, May 1992.

[10] *"SAS procedures guide, Version 6, Third Edition,"* SAS Institute Inc., Cary, NC, 1990.

[11] A. Koenig, *"C Traps and Pitfalls,"* Addison-Wesley, 1988.

[12] H. Mills, M. Dyer, and R. C. Linger, "Cleanroom software engineering," *IEEE Software*, pages 19–25, September 1987.

[13] C. U. Munoz, "An approach to software product testing," *IEEE Trans. on Software Engineering*, 14(11):1589–1596, November 1988.

[14] Glenford J. Myers, *"The art of Software Testing,"* John Wiley & Sons, Inc., New York, 1979.

[15] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. on Software Engineering*, SE-11(4):367–375, April 1985.

[16] S. Siegel, *"Nonparametric Statistics for The Behavioral Science,"* McGraw-Hill, 1988.

[17] M. A. Vouk, D. F. McAllister, and K. C. Tai, "An experimental evaluation of the effectiveness of random testing of fault-tolerant software," in *Proceedings of Workshop on software testing*, pages 74–81, Banff, Alberta, Canada, July 1986. IEEE Computer Society Press.

[18] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. on Software Engineering*, 17(7):703–711, July 1991.

[19] W. E. Wong, *"On Mutation and Data Flow,"* PhD thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, December 1993.

[20] T. H. Wonnacott and R. J. Wonnacott, *"Introductory Statistics,"* John Wiley & Sons, 1990.

Appendix A: Scatter plots of effectiveness versus size

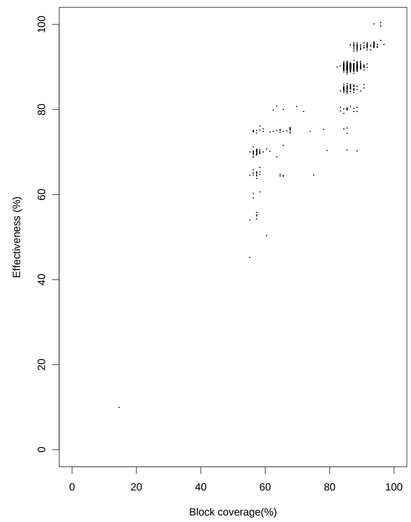

Figure 6: A scatter plot of effectiveness versus block coverage for the 537 test sets of `Cal`.

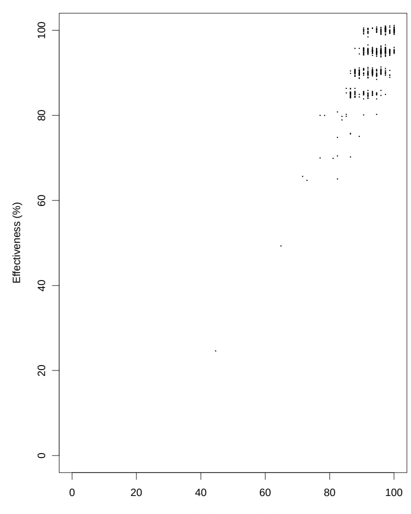

Figure 7: A scatter plot of effectiveness versus block coverage for the 538 test sets of `Checkeq`.

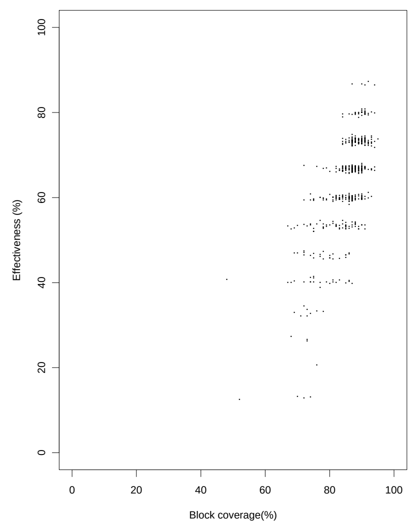

Figure 8: A scatter plot of effectiveness versus block coverage for the 540 test sets of `Comm`.

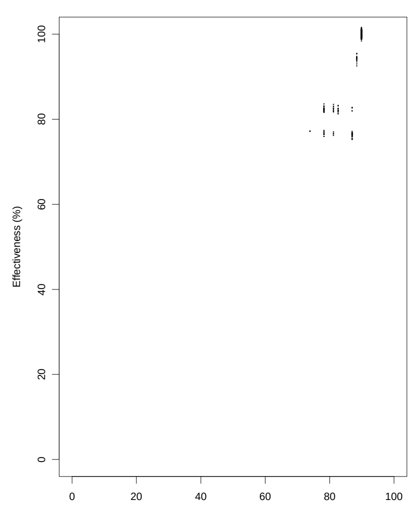

Figure 9: A scatter plot of effectiveness versus block coverage for the 539 test sets of `Crypt`.

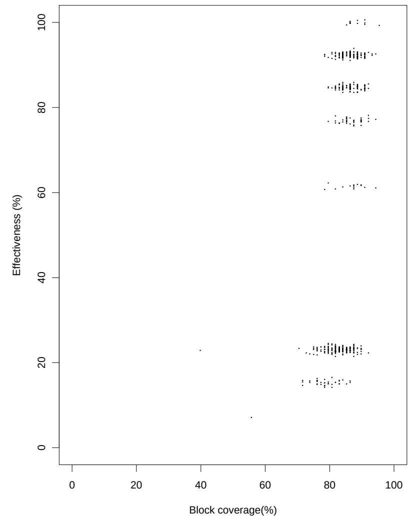

Figure 10: A scatter plot of effectiveness versus block coverage for the 539 test sets of `Look`.

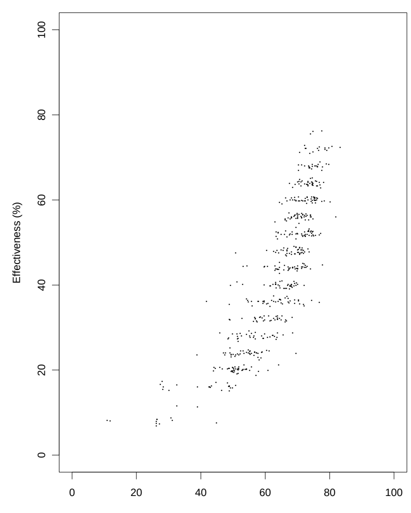

Figure 11: A scatter plot of effectiveness versus block coverage for the 540 test sets of `Sort`.

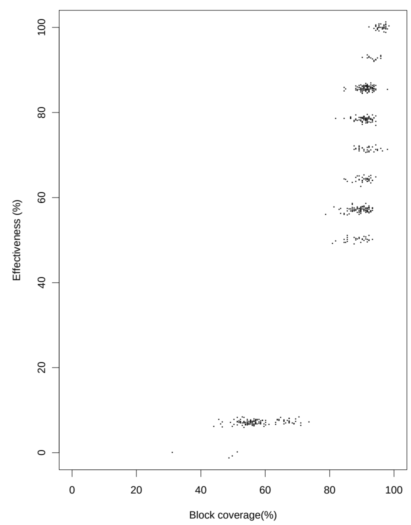

Figure 12: A scatter plot of effectiveness versus block coverage for the 540 test sets of `Spline`.

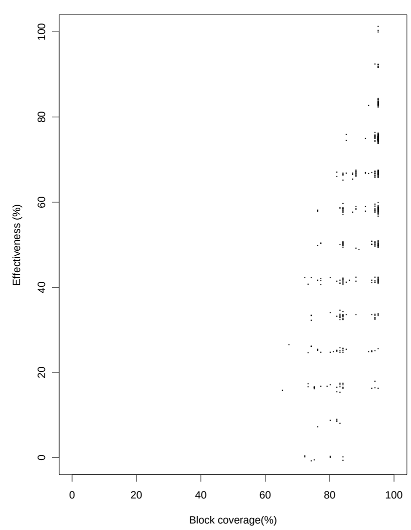

Figure 13: A scatter plot of effectiveness versus block coverage for the 540 test sets of `Tr`.

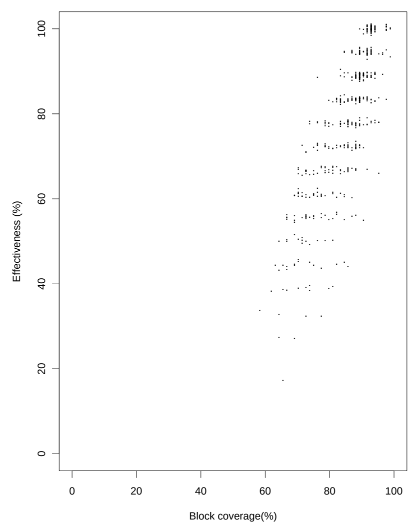

Figure 14: A scatter plot of effectiveness versus block coverage for the 540 test sets of `Uniq`.

Figure 16: Effectiveness (%) of test sets of size 2, 6, and 10 for `Col`.

Figure 18: Effectiveness (%) of test sets of size 2, 6, and 10 for `Crypt`.

Figure 20: Effectiveness (%) of test sets of size 2, 6, and 10 for `Sort`.

Figure 22: Effectiveness (%) of test sets of size 2, 6, and 10 for `Tr`.

Figure 23: Effectiveness (%) of test sets of size 2, 6, and 10 for `Uniq`.

Figure 25: A scatter plot of block coverage versus size for the 538 test sets of `Checkeq`.

Figure 27: A scatter plot of block coverage versus size for the 540 test sets of `Comm`.

Figure 29: A scatter plot of block coverage versus size for the 539 test sets of `Look`.

Figure 31: A scatter plot of block coverage versus size for the 540 test sets of `Tr`.

Figure 32: A scatter plot of block coverage versus size for the 540 test sets of `Uniq`.

Figure 34: Effectiveness (%) of test sets with (60-65), (70-75), and (80-85) block coverage (%) on `Col`.

Figure 36: Effectiveness (%) of test sets with (70-75), and (80-85) block coverage (%) on `Crypt`.

Figure 38: Effectiveness (%) of test sets with (50-55), (60-65), (70-75), and (80-85) block coverage (%) on `Sort`.

Figure 40: Effectiveness (%) of test sets with (60-65), (70-75), (80-85), and (90-95) block coverage (%) on `Tr`.

Figure 41: Effectiveness (%) of test sets with (50-55), (60-55), (70-55), (80-85), and (90-95) block coverage (%) on `Uniq`.