ON THE CORRELATION BETWEEN CODE COVERAGE AND SOFTWARE RELIABILITY*

Fabio Del Frate
University of Rome "Tor Vergata"
Via della Ricerca Scientifica
00133 Roma, Italy
Praerit Garg
Microsoft Corporation, One Microsoft Way
Redmond WA 98052-639
Email : praeritg@microsoft.com
Aditya P. Mathur
Software Engineering Research Center
1398 Department of Computer Sciences
Purdue University, W. Lafaioyette, IN 47907, USA
Alberto Pasquini
ENEA, Rome, Italy

Abstract

We report experiments conducted to investigate the correlation between code coverage and software reliability. Block-, decision-, and all-use- coverage measures were used. Reliability was estimated to be the probability of no failure over the given input domain defined by an operational profile. Four of the five programs were selected from a set of Unix utilities. These utilities range in size from 121 to 8857 lines of code. Artificial faults were seeded manually using a fault seeding algorithm discussed elsewhere. Test data was generated randomly using a variety of operational profiles for each program. One program was selected from a suite of outer space applications. Faults seeded into this program were obtained from the faults discovered during the integration testing phase of the application. Test cases were generated randomly using the operational profile for the space application.

Data obtained was graphed and analyzed to observe the relationship between code coverage and reliability. In all cases it was observed that an increase in reliability is accompanied by an increase in at least one code coverage measure. It was also observed that a decrease in reliability is accompanied by a decrease in at least one code coverage measure. Statistical correlations between coverage and reliability were found to vary between -0.1 and 0.91 for the shortest two of the five programs considered; for the remaining three programs the correlations varied from 0.89 to 0.99.

*This research was supported in part by an award from the Center for Advanced Studies, IBM Toronto Laboratories and NSF award CCR-9102331. All correspondence regarding this paper may be sent to Aditya P. Mathur (apm@cs.purdue.edu), Data not reported here may also be obtained by communicating at the above listed address.

1 Introduction

We report experiments conducted to investigate the relationship, if any, between code coverage and software reliability, hereafter referred to as reliability. Chen [1] has investigated this correlation using experimentation with randomly generated flowgraphs. Our experiments are different from Chen's as summarized in [2]. Veevers and Marshall [10] have also investigated this relationship in an analytic setting; our investigation is empirical.

The use of code coverage in reliability estimation rests on the assumption that there is a strong correlation between code coverage and reliability. Researchers have proposed models for reliability estimation that account for code coverage [4, 5]. These models are considered as

1071-9458/95 $4.00 ©1995 IEEE

an alternative to the traditional "black-box" models that do not account for the structure of the program whose reliability is to be estimated [7]. A critique of the traditional models may be found in [1, 4].

The code coverage measures considered in our experiments are block, decision, and all-uses. For definition and examples of these measures see [3]. Briefly, a block is a sequence of statements in a program in which control enters at the first statement and exits at the last statement of the sequence. A decision is a condition in a program which when evaluated may assume any one of two boolean values, true and false. All-uses comprise of def-use pairs. A def-use pair consists of two statements in a program the first of which contains an assignment of a value to some program variable, say `x`, and the second statement in the pair references the value of `x`. While testing program `P` we say that a block is covered if during some execution of `P` this block is executed at least once; a decision is covered if the corresponding condition evaluates at least once to true and at least once to false; a def-use pair for some variable `x` is covered when during program execution (a) control reaches the first statement in the pair and, (b) during the same execution, control reaches the second statement in the pair without reaching any statement that assigns a value to `x`. In the literature def-use pairs are further classified as p-use and c-use pairs. In the data reported below we have used def-use pairs only; a discussion on p-use and c-use pairs with reference to our experiments is in Section 4. Below we use the term "coverage" to refer to the three coverage measures described above.

The remainder of this paper is organized as follows. The experimental setup is described in Section 2. Results from the experiments and the analyses appear in Section 3. Finally in Section 4 we discuss our results from the point of view of a practitioner in the area of software testing and reliability.

2 Method

The following steps summarize our experimental methodology.

1. *Program selection*: Select programs to be used in the experiments.
2. *Operational profile generation*: Generate operational profiles for each program selected in Step 1.
3. *Fault set and erroneous program preparation*: Select and seed faults into each program.
4. *Coverage/reliability data generation*: For each program generate data to observe any correlation between coverage and reliability.

2.1 Program selection

A total of five programs namely, `Uniq`, `Crypt`, `Sort`, and `Grep`, were selected. The following criteria were employed to select these programs: (i) variety of application domain, (ii) variation in program size, (iii) ease of automatic generation of test data, (iv) programming language used, (v) availability of an operational profile, and (vi) availability of a fault set.

Criteria (i) and (ii) were set up to investigate if at all the application domain and program size have any effects on the correlation under study. Criterion (iii) was necessary to enable the random generation of tests from an operational profile. Criterion (iv) was necessary to enable the use of the code coverage measurement tool ATAC [3]. Criteria (v) and (vi) were used in selecting one program from a set of space application programs. `Space` was the only program that satisfied criteria (v) and (vi). Table 1 summarizes some measurable attributes of programs in the two sets. For example, program `Grep` has 4765 blocks, 3745 decisions, 28708 def-use pairs, and 8857 lines of code. The lines of code were measured to exclude any comment and blank lines.

2.2 Operational profile generation

An operational profile, denoted as O, specifies a mapping from disjoint subsets of the input domain to probability values. More formally, let $\mathcal{D}$ be the input domain of a program, 2^D the set of all subsets of the input domain, $D' \in 2^D$ a finite set of n disjoint subdomains, and $\mathcal{R}$ the set of real numbers. We assume that elements of D' can be indexed. Thus D'_1 denotes the first element of D', D'_2, the second element and so on. The operational profile is $O : D' \rightarrow R$ such that $\Sigma_{i=1}^{n} P(D'_i) = 1$, where $P(D'_i)$ denotes the probability associated with the i^{th} element of set D'.

We constructed t various operational profiles for `Crypt`, `Uniq`, `Sort`, and `Grep` in an arbitrary manner.

Table 1: Attributes of programs used.

Attributes	Crypt	Uniq	Sort	Grep	Space
Blocks	69	84	508	4765	2970
Decisions	39	58	394	3745	1179
All Uses	113	124	1624	28708	5359
Lines of code	121	125	841	8857	6107

The only guideline observed in constructing these profiles was the one to obtain variety. For `Space` we had an operational profile available. The operational profiles for all programs may be found in [2].

2.3 Fault set and erroneous program preparation

For `Uniq`, `Crypt`, and `Sort` the fault set was borrowed from an earlier study [6]. The rationale for selecting these faults is also described in [6]. A fault set was constructed for `Grep`. The fault set for `Space` was obtained from the error-log maintained during its testing and integration phase. The entire fault set for all programs may be found in [2].

We have numbered each fault in a program as `Fk` where the integer `k` denotes the fault number. This number does not indicate the order in which faults were detected during experimentation. Thus, for example, fault `F4` in `Crypt` was not necessarily the fourth fault detected. Where needed we have specified the program file in which the fault was injected. A total of 10, 10, 10, 19, and 22 faults were injected respectively, into `Crypt`, `Uniq`, `Sort`, `Grep`, and `Space`. The line number, and the file name where needed, together with the incorrect and correct strings specify a fault uniquely. The incorrect string may be an empty string which means that some code was changed to remove a fault. For example, a correction of fault F17 in `Grep`[2] requires the string `strcpy(n, '''');` to be added at line 270 in file `search.c`.

All faults from a fault set were injected into the corresponding program. Thus exactly one erroneous program was generated from its correct version. Below we use the term P_k to denote a program with k faults removed. P_N is the correct program.

2.4 Coverage/reliability data generation

To generate coverage and reliability data, the following sequence of steps was executed for each program in the two sets. Recall that P_i is the program under test with i faults removed. The algorithm below is a high level description; each step in this algorithm is detailed subsequently.

High level algorithm

1. Let OP_k denote the kth operational profile for the program under test. Repeat Step 2 for each OP_k for k varying from 1 to the total number of operational profiles (NOP).
2. Let N denote the number of faults in the program under test. Repeat Steps 3 and 4 for i, the number of faults removed, varying from 0 to (N-1).
3. Repeat Steps 3a and 3b until the block coverage converges according to the criterion described below.
   (a) Compute the reliability of P_i.
   (b) Execute P_i on test data generated randomly from the selected operational profile until a failure occurs. Record various code coverage measures.
4. Identify and remove the fault responsible for the largest number of failures in Step 3b. This gives us P_{i+1}.

Note that for each P_i we compute average values of reliability and code coverages. Thus, for example, a block coverage value of 0.3 for P_2 for OP_3 is to be interpreted as: "It requires an average of 30% of program blocks to be covered to remove the 2 faults from P_2 when tested according to OP_3." A similar interpretation holds for other coverage measures and the reliability.

The convergence criterion used in Step 3 was based on the 99% confidence bounds computed on the block coverage [9]. Thus, the loop terminated when the block coverage converged as per this criterion. This criterion was not applied for **Space** as we used a known operational profile in this case. We discuss the convergence aspects of coverage and reliability values in Section 4.

Reliability computation

In Step 3a the reliability of P_i is computed for each operational profile OP_k by the following algorithm. The inputs to this algorithm are P_i, OP_k, and δ. The output is the reliability of P_i for OP_k.

1. Set `total-executions` and `failures-observed` to 0.
2. Randomly select a test case t from OP_k.
3. Execute P_i on t.
4. (a) Increment `total_executions` by 1. If the program does not fail then repeat from Step 2.

   (b) If the program fails increment `failures_observed` by 1 and follow the next step.
5. (a) If `total_executions` is less than 1000 then repeat from Step 2.

   (b) If `total_executions` is 1000 then compute `fp_old` as: `failures_observed/total_executions` and repeat from Step 2.

   (c) If `total_executions` is greater than 1000 then compute `fP_new` as:

   `failures_observed/total_executions`.

   (d) If $|(\texttt{fp\_new} - \texttt{fp\_old})| > \delta$ then the failure probability has not yet converged, set `fp_old` = `ft_new` and repeat execution from Step 2 otherwise follow the next step.
6. One estimate of the reliability of P_i for operational profile OP_k is $(1 - \texttt{fp\_new})$.

Coverage computation

This step takes P_i and OP_k as inputs and computes the coverage values at which a fault is detected. The following algorithm is used.

1. Select a test case t randomly from OP_k
2. Execute P_i on t.
3. (a) If the program does not fail and at least one coverage measure has changed in the previous 50 executions then repeat execution from Step 1.

   (b) If the program fails then record block, decision, p-use, c-use, and all-uses coverage using ATAC.

3 Results and analysis

Below we examine the data obtained from the experiments. In our examination we look for answers to the following questions:

1. What is the relationship between code coverage and faults detected?
2. What is the relationship between reliability and faults detected?
3. What is the relationship between code coverage and reliability?
4. How does the answer to each of the questions above depend on program complexity?

For each program in the two sets coverage values and reliability were measured initially when the first fault was discovered subsequently after each fault was removed. Figure 1 show the plots of coverage and reliability values for the faulty program after each fault found was removed. Note that all coverage and reliability values plotted here are averages as explained in Section 2. The horizontal axes in these graphs labelled **faults** is the count of faults detected; it is not the fault number, e.g. F7, found in the fault tables in the Appendix. The coverage (reliability) value in these graphs is the value after a fault is removed. For example, in Figure 1, the coverage at the **fault removed** value of 4 is the coverage measured when four faults were removed and the fifth was detected. In the discussion below we use the term "coverage" to refer jointly to the three coverage measures used in our experiments. Where necessary we prefix "coverage" with "block", "decision", or "all-uses".

Before plotting the coverage values in Figures 1 were transformed using $(C_i - C_{min})/factor$ where C_i is the coverage computed immediately after i faults have been

Table 2: Terminology used in the description of experiments.

Term	Meaning
P_i	Program under test after the removal of i faults.
OP, OP_i	Operational Profile or the ith operational profile.
NOP	Number of operational profiles used for a program.
N	Total faults seeded in a program.
R_i	Reliability of a program with i faults removed.
$FP(P_i)$	Failure probability of P_i; $R_i = 1 - FP(P_i)$
δ	Used to detect convergence of R_i

detected and $(i-1)$ faults removed, C_{min} is the minimum coverage value observed for program and an operational program, and *factor* denotes a scale factor. In the absence of this transformation the variation in coverage values was not visible as the reliability values are plotted on the same vertical axes; hence the need for transformation. The scale factors and minimum coverage values are listed in Table 3. F, Min. and Max. denote, respectively, the scale factor used, minimum and maximum coverage values. B, D, and A denote, respectively, the block, decision, and all-uses coverage. All coverages are percent values. Minimum and maximum values are computed over all executions of a program for an operational profile.

3.1 Code coverage and faults detected

We find that a coverage measure computed after removing a fault may increase, decrease, or remain unchanged. For example, for program `Crypt` in Figure 1, operational profile 1, the decision coverage reduces and increases when three and five faults, respectively, are removed. For operational profile 1 of `Sort` the decision coverage measure remains unchanged after the second fault has been detected. An increase in coverage causes previously unexecuted code to be executed. If the fault lies in the code that is executed the first time there is chance that it will be revealed.

There are two reasons why code coverage may decrease when a fault is removed. The first reason concerns the notion of *fault masking*. For a given program P a fault f_1 is said to mask another fault f_2 if (a) there is no test case in the input domain of P that will reveal f_2 and (b) there is at least one test case in the input domain of P that will reveal f_2 after f_1 has been removed. Due to fault masking a test set T_1 may provide a coverage value of C prior to the removal of f_1. However, once f_1 is removed a lesser amount of code is executed resulting in a coverage lower than C. For example, in `Uniq`, after the second fault is removed, the third is encountered at a lower coverage. This happens because the second fault at line 130 masks the fault at line 127 in the program causing the test cases to execute much more code earlier than later when the second fault was removed [2]. Similarly, in `Crypt`, the coverage dips when the fourth fault is encountered. Here too the third fault at line 132 masks the fault at line 57.

The second reason has to do with the nature of correction resulting from the removal of a fault. To correct a fault f_1 in program P it may be necessary to add code to P. When P is executed against T, the test set which revealed the fault, the added code may not be executed resulting in a lower value of coverage. This, however, did not happen in any of our programs. We conjecture that coverage reduction due to this reason is more likely to occur when the amount of code added is significant relative to the size of P. In all our programs only a few additional statements were required to be added in order to correct a "missing code" fault.

Code coverage does not decrease or remain unchanged in `Grep` and `Space`. In these programs whenever a fault was discovered code coverage did increase beyond its initial value or the value prevailing when the previous fault was removed.

The graphs discussed above indicate that coverage values may change in various ways when a fault is detected. The change in coverage after the removal of fault i until the detection of fault $(i+1)$ is discussed in [2].

Table 3: Minimum and Maximum coverage values and scale factors used in transforming coverage data prior to graphing it.

Program	OP 1			
	F	B	D	A
Crypt	20			
Min.		73.44	60.47	61.52
Max.		81.16	76.92	74.33
Grep	45			
Min.		1.07	0.75	0.57
Max.		48.33	38.57	21.10
Space	1			
Min.		34.0	20.0	25.0
Max.		78.00	66.00	69.52

3.2 Reliability and faults detected

As seen from the graphs in Figure 1 the reliability may remain increase, decrease, or remain unchanged after the removal of a fault. For example, in operational profile 1 of `Uniq` (for a graph see [2]), the reliability remains unchanged before and after the second fault is removed at 0.0087 and also before and after fifth fault is removed at 0.0204. It decreases from 0.0087 to 0.0081 when the third fault is detected and removed. Also, it increases from 0.0204 to 0.37 when the sixth fault is removed and again from 0.37 to 0.89 when the seventh fault is removed.

A decrease in reliability occurs due to fault masking. For example, for operational profile 1 of `Uniq`, the third fault is masking the fourth fault [2]. When the third fault is removed, the fourth is exposed causing the program to fail more frequently. Similarly, for operational profile 3 of `Sort` the reliability decreases from 0.932 to 0.931 when the third fault is removed causing the program to fail more frequently on the fourth fault (for a graph see [2]).

3.3 Statistical correlations

Statistical correlations [9] were computed for each program and operational profile. The correlation between coverage and reliability values appear in Figure 2. These correlations were computed using pairs of coverage and reliability values plotted in Figures 1. These values for `Crypt` and `Uniq` vary from $0.16 - 0.91$ and $-0.10 - 0.72$, respectively. There was much less variation in correlation values for `Sort`, `Grep`, and `Space` as indicated by the values $0.93 - 0.99$, $0.89 - 0.91$, and $0.98 - 0.99$, respectively, for the three programs.

4 Discussion

The observations made in the previous section are summarized below. Recall that in the experiments reported here (a) random testing was used with respect to an operational profile and (b) once detected, a fault was removed before testing resumed.

1. Code coverage measures and program reliability may increase, decrease, or remain unchanged as faults are found and removed.
2. An increase in code coverage is always accompanied by an increased or unchanged reliability.
3. The above two observations are independent of program complexity measures. However, variations in coverage and reliability when measured against the number of faults removed is smoother for larger programs.
4. Statistical correlations between coverage and reliability vary widely for relatively smaller programs; for larger programs these correlations are high with less variation. This statement holds for correlation between change in coverage and change in reliability.

In addition to the above observations, the plots of coverage-reliability versus number of faults removed also

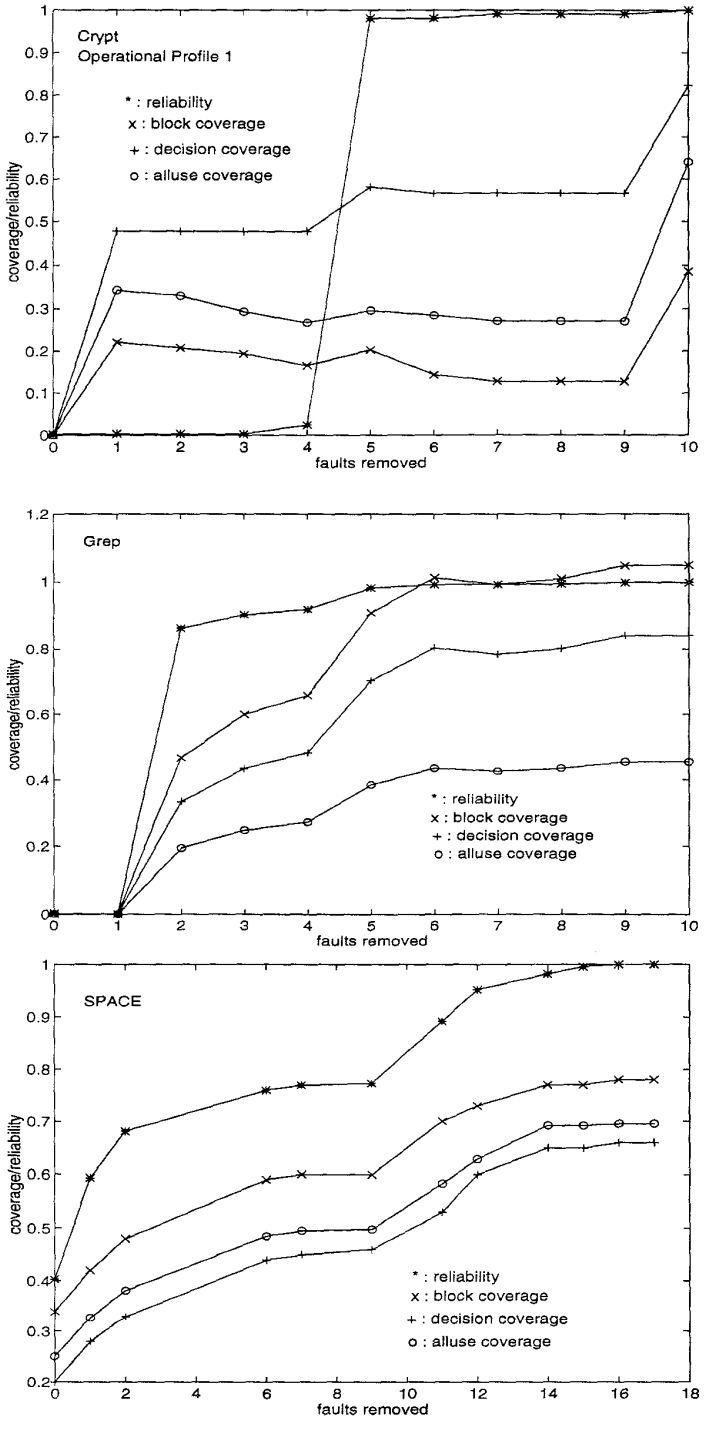

Figure 1: Coverage and reliability versus faults detected for `Crypt`, `Grep`, and `Space`.

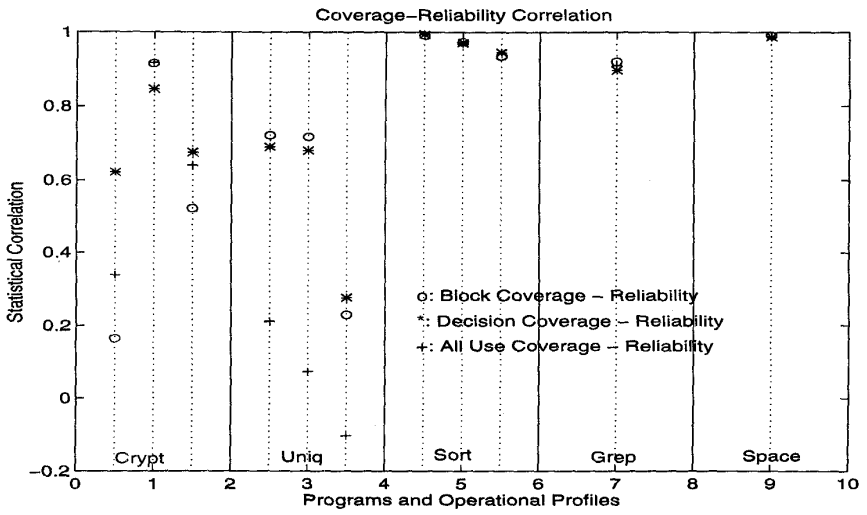

Figure 2: Statistical correlation between code coverage and reliability.

indicate that in `Crypt`, `Uniq`, and `Sort` all faults seeded were found before any coverage measure attained its maximum value of 1. In the case of `Grep` 10 faults could be removed after approximately two months of program execution on a Sun Sparc 5 machine. The number of test cases generated was not measured. Despite intensive program execution none of the code coverage measures reached their maximum values. The reliability reached a high of 0.999999. For `Space` 18 out of a total of 22 faults could be removed and no coverage measure attained its maximum value despite program execution on over 50000 test cases. The measurable attributes were found to change as a result of fault detection and removal; the change, however, in any attribute of any program was less than 3% [2].

As explained earlier, the all-uses in a program consist of p- and c-uses. In our experiments p- and c-use coverage was measured and analyzed. The data and analysis is not reported here as the relationship between p- and c-use coverages, faults detected, and reliability, followed the same pattern as that observed with all-uses coverage. The p- and c- use coverage data may be obtained from the authors.

The wide variation in correlations between coverage and reliability for `Crypt` and `Uniq` may be attributed to the erratic behavior of coverage and reliability values in the initial stages of testing (see Figure 1. This behavior is not observed in the remaining three programs.

The application domain itself does not appear to have any effect on the coverage-reliability relationship. Our experiments did not have enough variety of applications with similar complexity to be able to make any justifiable conclusions regarding the effect of application domain on the coverage-reliability relationship. However, this relationship is more likely to be dependent on code complexity than on the application domain.

For the benefit of a practitioner the following point of view is derived from the above discussion and results presented.

1. Increase in code coverage is likely to increase reliability. As coverage increases, dips in reliability due to fault masking are possible though unlikely.

2. The block coverage measure is powerful enough to reveal a large fraction of the faults in the program, all faults in some cases, before the coverage measure itself peaks at 100%.

3. Random testing may require an impractical number of test cases to increase the block coverage to beyond 75%. Recalling that other coverage measures used are more difficult to satisfy, this conclusion is true for other measures too.

4. If expense is measured in terms of the number of test cases to obtain a given level of reliability, random testing is several orders of magnitude more ex-

pensive than coverage based testing. We assume that coverage based testing directs the development of new test cases to improve one or more coverage measures. This is not true of random testing where test cases are generated randomly. Thus a large sequence of test cases may be generated without resulting in any increase in code coverage. This is particularly true in the case of larger programs, e.g. `Grep` and `Space`.

5 Acknowledgement

We express thanks to the Center for Advanced Studies, IBM, Toronto for providing partial support for this research. Thanks to the European Space Agency and the company IDS for providing us the SPACE program used for the experiment, Dott. A. Cancellieri for helping us in inserting the original faults in this program and the student P. Matrella for developing some of the tools used in the experiment. Thanks to the following people who contributed to the discussions that led to the design of the reported experiments: Richard DeMillo, Bob Horgan, Yashwant Malaiya, John Musa, Michael Lyu, Vernon Rego, Nozer Singpurwalla, and Marty Shooman.

References

[1] M. H. Chen, A. P. Mathur, and V. J. Rego, " Effect Of Testing Techniques On Software Reliability Estimates Obtained Using Time Domain Models," *Proceedings of the 10th Annual Software Reliability Symposium*, IEEE Reliability Society, Denver, Colorado, pp. 116-123, June 25-26, 1992.

[2] F. D. Frate, P. Garg, A. P. Mathur, and A. Pasquini, "Experiments to Investigate the Correlation Between Code Coverage and Software Reliability," SERC-TR-P-162, April 1995, Software Engineering Research Center, Purdue University, W. Lafayette, IN 47907.

[3] J. R. Horgan and A. P. Mathur, "Assessing Tools In Research And Education," *IEEE Software*, May 1992, pp. 61-69.

[4] J. R. Horgan, A. P. Mathur, A. Pasquini, and V. J. Rego, "Perils of Software Reliability Modelling," Technical Report, SERC-TR-160-P, February 1995, Software Engineering Research Center, Purdue University, W. Lafayette, IN 47907.

[5] Y. K.Malaiya, N. Li, J. Bieman, R. Karcich, and R. Skibbe, "The Relationship Between Test Coverage And Reliability," *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, Monterey, CA, November 6-9, pp 186-195, 1994.

[6] A. P. Mathur and W. E. Wong, "Fault Detection Effectiveness of Mutation and Data Flow Testing," *Software Quality Journal*, Vol. 4, pp 69-83, 1995.

[7] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.

[8] A. Pasquini, "Measuring Software Reliability in a Space Application", *Proceedings of the European Safety and Reliability Conference*, Elsevier Applied Science, June 1992.

[9] K. S. Trivedi, "Probability & Statistics with Reliability, Queuing and Computer Science Applications", 1982, Prentice-Hall Inc, Englewood Cliffs, N.J., USA.

[10] A. Veevers and A. Marshall, "A Relationship Between Software Coverage Metrics and Reliability," Software Testing, Verification and Reliability, Vol. 4, pp. 3-8, 1994.