# Incorporating Code Coverage in the Reliability Estimation for Fault-Tolerant Software

Mei-Hwa Chen
SUNY at Albany
Albany, NY 12222
mhc@cs.albany.edu

Michael R. Lyu
Lucent Bell Labs.
Murray Hill, NJ 07974
lyu@research.bell-labs.com

W. Eric Wong
Bellcore
Morristown, NJ 07960
ewong@bellcore.com

## Abstract

*We present a technique that uses coverage measures in reliability estimation for fault tolerant programs, particularly N-version software. This technique exploits both coverage and time measures collected during testing phases for the individual program versions and the N-version software system for reliability prediction. The application of this technique on the single-version software was presented in our previous research. In this paper we extend this technique and apply it on the N-version programs. The results obtained from the experiment conducted on an industrial project demonstrate that our technique significantly reduces the hazard of reliability overestimation for both single-version and multi-version fault tolerant software systems.*

## 1 Introduction

Reliability is an important metric for evaluating system dependability and quality. Traditional work on reliability analysis for a distributed system or for a network system often emphasizes hardware reliability. Due to the increasing complexity of computer systems, software has become a critical system component to provide flexible yet powerful applications of computers. Therefore, it is necessary to formulate a methodology that can be used to improve software reliability, which directly impacts the overall system reliability.

Fault tolerant software approaches to improving software reliability have been thoroughly studied in [1, 24] for the recovery block approach, in [2] for the $N$-version programming approach, and in [15, 16] for the $N$ self-checking programming approach. The dependability analyses of fault-tolerant software systems are conducted by either adopting a Markov approach [9, 16] or using stochastic reward nets [27]. Reliability models considering correlated software faults are discussed in [10, 12]. A comprehensive performance and reliability analysis of fault-tolerant software is provided in [26].

Parallel to these studies, a number of analytic models have been proposed to estimate software reliability [23]. The class of models which capture reliability growth during software testing, generally called software reliability growth models (SRGMs), have been widely researched. These models make use of the failure history of a program during testing to predict the field behavior of the program under the assumption that testing is performed in accordance with a given operational profile [17, 20].

The above reliability and dependability models, however, do not take the coverage of software into consideration. Coverage criteria that capture important aspects of a program's behavior during execution are proposed in [8, 14, 25], and empirical evidence of coverage measures to represent test completeness is illustrated in [13]. Observations, from both empirical and analytical studies, indicate that the reliability predictions made by the SRGMs tend to be too optimistic. [4, 7]. Consequently, many researchers postulate that coverage information should be used instead of testing time alone to overcome the difficulty of obtaining an accurate operational profile. Vouk [28] investigated the relation between test coverage and fault detection rate. Piwowarski et al. [22] observed the coverage measures on large projects during a function test and derived a coverage-based reliability growth model which is isomorphic to the Goel-Okumoto NHPP model and the Musa execution time model. Malayia et al. [19] modeled the relation among test effort, coverage and reliability and proposed a coverage based logarithmic model that relates a test coverage measure with fault detection.

Up to now the method of software reliability measurement made by SRGMs does not distinguish between fault-tolerant and non-fault-tolerant software. Few studies (except, for example, [29]) discuss the effectiveness of testing on fault-tolerant software. Moreover, the correlation between code coverage and the reliability of fault-tolerant software has never been explored. Existing reliability and dependability models on fault tolerant software are based on approaches where software components are treated as black boxes. No coverage measure or structural information was considered.

In this paper we present a model that uses coverage measures in reliability estimation for fault tolerant programs, particularly $N$-version software. This model employs both coverage and time measures collected during testing phases to predict operational reliability. The failure rate used by the SRGMs is adjusted by the failure rate derived from our model which considers both testing time and test coverage, and the redundant test effort (i.e., extra testing time without increasing program coverage) is reduced. Using this coverage enhanced pre-processing technique, we applied the extracted test data to the Goel-Okumoto NHPP model [11] and the Musa-Okumoto Logarithmic model [21] and observed the improvement of the estimation made by both models. The technique was applied to an industrial project on $N$-version programming to demonstrate its strength in real world usage. The empirical results show that the reliability of the $N$-version software increases when the code coverage of each individual program version and the coverage of the $N$-version system increase. We also show that, compared with traditional software reliability models, our model gives a better reliability estimation since software coverage measures are used in the estimation process.

The notion of coverage in this paper specifically refers to the coverage of a program with respect to the coverage of a program's blocks, not the coverage defined for fault tolerance. However, the coverage of testing is shown to be closely related to the detection of software faults, which in turn helps to improve the coverage of fault tolerance with respect to a generic class of faults.

## 2 Coverage-Reliability Relationship

Existing SRGMs make use of the failure history of a program to estimate the reliability of the program. Parameters used in such models are estimated using failure data collected during testing. As testing proceeds, it is more likely for a test case to increase coverage during an early testing phase than during a later phase. Thus, it becomes increasingly more difficult to construct a test case that will cause the program to execute the uncovered portion and detect faults in the program, and the time between failures increases as testing time increases, upon which reliability growth phenomena will be observed. However, the reliability of the program will increase only if the number of residual faults in the program is reduced. Redundant test cases do not explore new execution (either with new paths or new data) of the code and do not contribute to testing effectiveness, although they do increase failure-free testing time, resulting in overestimates in reliability growth. The more redundant test efforts are used, the more overestimates there will be.

To reduce the overestimates, we need to determine which test cases are redundant and how much test effort should be taken into account. We have proposed a coverage based technique to enhance reliability estimation on single-version programs [6]. In this work we have defined the notion of *effective* testing effort. A testing effort is effective if and only if it increases some type of coverage or it reveals some faults. A coverage enhanced technique utilizes coverage measures to determine the effectiveness of a test effort.

### 2.1 The Coverage Enhanced Technique

The test effort mentioned in the previous section is referred to as the time required by executing a test case. If a test case executes some uncovered portion of the program and/or the test case causes some failures to be triggered, then the test effort required by this test case is considered effective. Otherwise the effort is considered non-effective.

The coverage enhanced technique is used to pre-process test data obtained from the testing process before the data are applied to the reliability estimation. The pre-processing involves filtering all the non-effective test efforts and the factor used in filtering is the coverage increasing rate measured prior to the non-effective test case. A detailed mathematical description of this technique is in illustration in Appendix.

We extend this technique and apply it to the $N$-version software where the concept of super-program is introduced. The super-program is composed of $N$ versions of the program, and the coverage is measured against the super-program. Therefore, an effective test effort is considered in the $N$-version software under the following conditions:

- the majority of the software versions agree with the output,
- at least one of the versions exercises some uncovered codes.

The experiments conducted using this technique are presented in the next section.

## 3 Description of the Experiments

In order to demonstrate our technique for real world applications, we selected an industrial project which was developed and programmed by multiple programming teams reported in [18]. The application program was an automatic flight control function for the landing of commercial airliners that had been implemented by the avionics industry.

We randomly selected five of the program versions in this project and a super program consisting of these five versions was used in our experiments. The reliability measurement for a single-version configuration is obtained by randomly selecting exactly one of the five programs at a time for execution.

### 3.1 Testing and Debugging

The testing and debugging sequence is explained below.

Step 1: generate a test pool and initialize TEST_REPEAT to be *false*,

Step 2: `if` (TEST_REPEAT equals *true*)
`then` (a) use test case $t$ saved in Step (4)(d)
`else` (b) select a test case $t$ from the pool according to a uniform distribution profile,

Step 3: test the super program against $t$ by executing one of the five versions

(say $\mathcal{P}$) selected according to a uniform distribution profile,

Step 4: `if` ($\mathcal{P}$ fails on $t$)
`then` (a) find the fault(s) which is (are) responsible for the failure
(b) remove the fault(s) detected above
(c) set TEST_REPEAT to be *true*
(d) save $t$ for later use
(e) goto Step (2)
`else` (f) set TEST_REPEAT to be *false*
(g) goto Step (2).

The same super program is also applied for a multi-version configuration. For the purpose of illustration, a 3-version configuration was selected in our experiments as an example of a multi-version approach.

The reliability measurement for a 3-version configuration is similar to that for a single-version configuration except that three different versions, instead of just one, were selected each time for test execution. The testing and debugging sequence is given below.

Step 1: same as before,

Step 2: same as before,

Step 3: (a) construct a 3-version configuration (say $\mathcal{Q}$) by selecting three of the five versions from the super program
(b) test $\mathcal{Q}$ against $t$,

Step 4: same as before
However, the failure of $\mathcal{Q}$ is defined as the majority (two or three versions) of $\mathcal{Q}$ fail, *regardless* of the failure symptoms.

One important characteristic of the above testing and debugging process is that $\mathcal{P}$ (or $\mathcal{Q}$) may be executed against the same test case $t$ more than one time. For example, suppose the first execution of $\mathcal{P}$ (or $\mathcal{Q}$) on $t$ fails at time equals to $\alpha$. Since the simulation is not yet completed, $\mathcal{P}$ (or $\mathcal{Q}$) should be re-executed on $t$ after the responsible fault(s) is (are) removed. Suppose this time, $\mathcal{P}$ (or $\mathcal{Q}$) experiences another failure at time equals to $\beta$, with $\beta > \alpha$. Then, the debugging and testing process just described should be repeated with respect to the same test case $t$. Such a process continues until $\mathcal{P}$ (or $\mathcal{Q}$) succeeds on $t$. Hereafter, we refer to the super program as the program.

### 3.2 Program Coverage and Fault Exposure

Before performing reliability analysis of the non-fault-tolerant and fault-tolerant software configurations, we need to examine the effectiveness of testing coverage in revealing faults, and to evaluate the validity of the injected faults compared with the original faults. Table 1 shows the detection of the faults with respect to testing coverage changes among the five program versions. The first column represents the id of a version. The second to fifth columns identify the number of faults whose detection is related to changes of blocks, decisions, c-uses, and p-uses, respectively. For example, 6/8 for the $\epsilon$ version under the "Blocks" column means six out of the eight $\epsilon$ faults were detected when block coverage was changed during testing. The last column, "Any," counts changes in *any* of the four coverage measures. Measurements of the overall coverage changes among the five programs are provided in the last row. It can be seen from the table that all the coverage measures make a significant contribution to the detection of faults, where 70% of faults are detected when block coverage increases, and 80% of faults are detected when any one of the coverage measure changes. In addition, C-uses is more effective than the other three measures. We also note that every version except $\gamma$ has all but one fault whose detection is related to some of the four coverage measures.

Table 2 compares the "detectability" of injected faults versus original faults by listing the average time units the faults remain in the software before they are detected. The first column represents the id of a version. The following three columns show the average latency for a class of faults in each version, where the second column is for the original faults, the third column is for the injected faults, and the last column is for all the faults. The overall comparison is listed in the last row, from which we can see that it takes, on average, longer time to detect the injected faults. This implies that the faults we injected in these programs are more difficult to detect than the original faults.

| Version Id | Blocks | Decisions | C-Uses | P-Uses | Any |
|---|---|---|---|---|---|
| $\epsilon$ | 6/8 | 6/8 | 7/8 | 6/8 | 7/8 (87.5%) |
| $\gamma$ | 7/14 | 8/14 | 9/14 | 8/14 | 10/14 (71.4%) |
| $\kappa$ | 5/7 | 5/7 | 5/7 | 5/7 | 6/7 (85.7%) |
| $\lambda$ | 8/9 | 8/9 | 8/9 | 8/9 | 8/9 (88.9%) |
| $\nu$ | 6/7 | 6/7 | 6/7 | 6/7 | 6/7 (85.7%) |
| Overall | 32/46 (69.6%) | 33/46 (71.7%) | 35/46 (76.1%) | 33/46 (71.7%) | 37/46 (80.4%) |

**Table 1. Fault detection related to improvement of test coverage**

| Version Id | Original | Injected | Total |
|---|---|---|---|
| $\epsilon$ | 764.4 | 1026.2 | 862.5 |
| $\gamma$ | 1082.8 | 788.4 | 872.5 |
| $\kappa$ | 560.4 | 3063.8 | 1633.3 |
| $\lambda$ | 1488.9 | 704.2 | 1096.5 |
| $\nu$ | 216.9 | 972.8 | 756.8 |
| Overall | 918.7 | 1097.7 | 1017.6 |

**Table 2. Time to detect original faults versus injected faults**

## 3.3 Reliability Estimation

### 3.3.1 Single-Version Configuration

Figure 1 shows the reliability estimates obtained by applying the original data of the single-version configuration, collected from the testing process described in Sec. 3.1, and the data, processed by using the coverage enhanced technique, to the G-O model and the M-O model. The exposure time used in the estimation process was the maximum flight time which was 264 time units (5280 program iterations).

Reliabilities measured as the ratio of the number of failures to the number of executions were computed at *testing time* equals 18080.6, 19660.1, 20713.9, 23884.2, 34453.4 and 46341.6 time units, respectively. These six points were selected because they corresponded to the time when the last six fault correction activities occurred. While computing the reliability, the program was executed against inputs generated based on the same operational profile as used in the testing process. Such execution continued until the reliability converged to a 95% confidence interval.

The overestimates made by the G-O model and the M-O model are shown in Figure 2. For the G-O model the differences between the estimates and the reliability ranged from 0.0516 to 0.003 (5.4% to 0.3%). If the coverage enhanced technique was applied, they ranged from 0.029 to 0.00056 (3.0% to 0.06%). For the M-O model, the differences ranged from 0.046 to 0.0004 (4.9% to 0.04%) and the results obtained after applying the technique were from 0.027 to -0.00036 (2.9% to -0.036%).

The results obtained from this case study show that the coverage enhanced technique improves the reliability estimation made by the SRGMs and brings the estimates much closer to the reliability in the single-version configuration.

### 3.3.2 3-Version Configuration

The results observed from the experiment on the 3-Version configuration are depicted in Figure 3 and Figure 4. In this experiment, the reliability estimation time and the reliability computation time are 20335.1, 28253.2, 36969.7, 50174.8 and 71044.0 time units, which are the times of the last five fault correction activities, respectively.

The overestimates made by the G-O model ranged from 0.06 to 0.001 (6.3% to 0.1%) and those made by the M-O model ranged from 0.057 to 0.0002 (6.0% to 0.02%). If coverage information was used to enhance the reliability estimation, the overestimates ranged from 0.018 to 0.001 (1.9% to 0.1%) and from 0.0156 to 0.00084(1.6% to 0.084%) for the G-O model and the M-O model.

The results obtained from this case study also show that our technique improves the reliability estimation for the 3-version configuration.

## 3.4 Analysis and Discussion

A common argument regarding the N-version approach is the possible occurrences of correlated failures. In our application, the correlated failures can be considered in two situations: one is when more than one version produces identical incorrect output or fails at the same time frame; the other, when a majority of the programs fail during the execution of

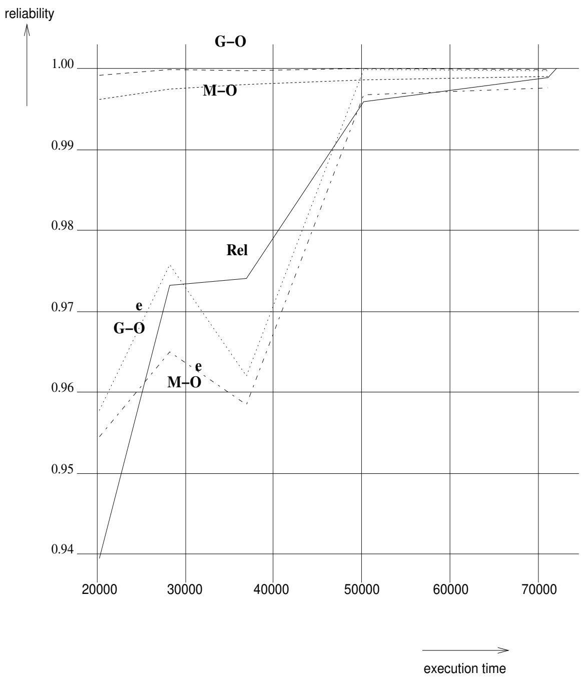


**Figure 1. Reliability estimates obtained from the models and the reliability for single-version software.**

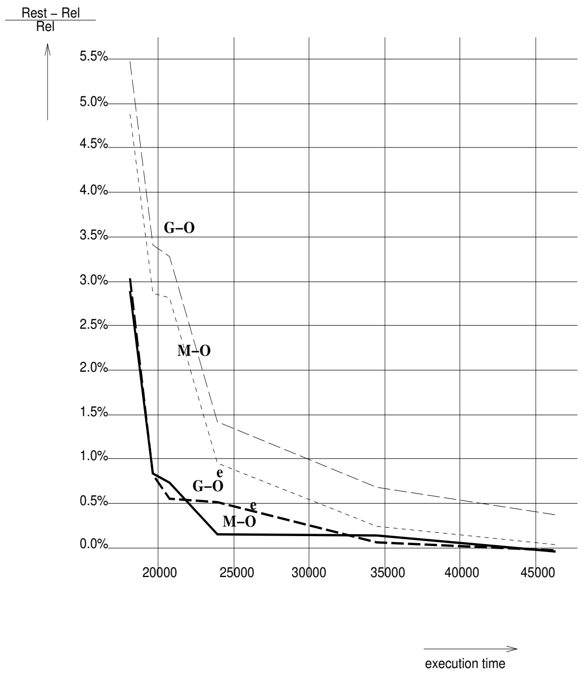


**Figure 2. Reliability overestimates made by the models for single-version software.**

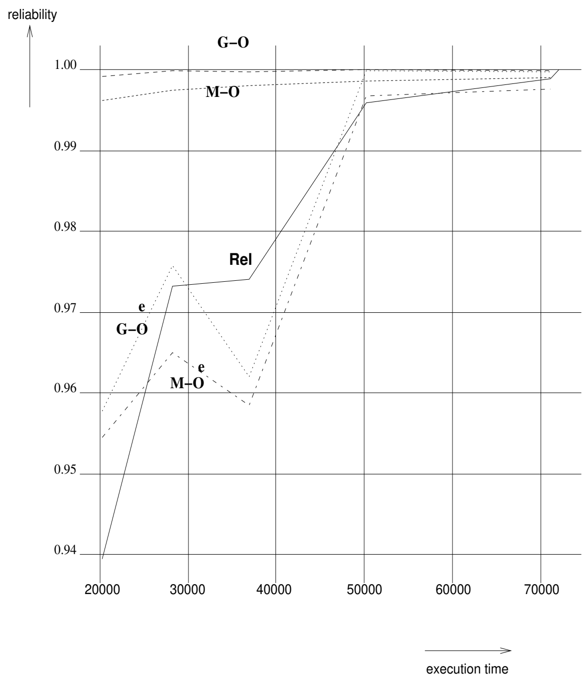


**Figure 3. Reliability estimates obtained from the models and the reliability for the 3-version software.**

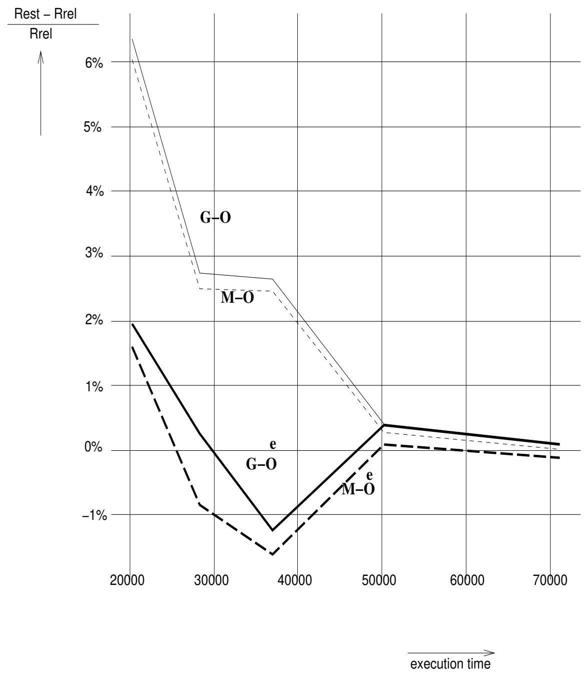


**Figure 4. Reliability overestimates made by the models for the 3-version software.**

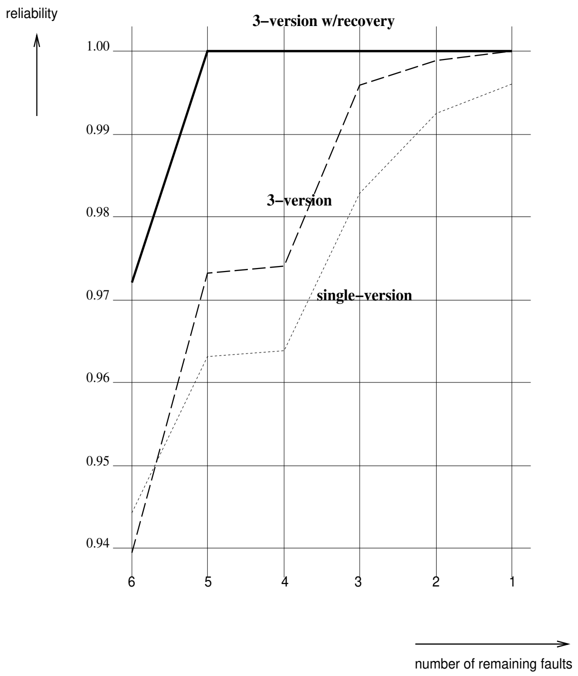


**Figure 5. Reliability comparison for the 3-version and single-version configurations**

an input instance regardless of time and output values. Our experiments described in the previous section take a conservative view of the second situation.

Figure 5 plots the reliabilities of both single-version and 3-version configurations. The reliabilities were compared when the number of remaining faults decreased from six to one. The 3-version configuration is represented by two curves. The first curve (dashed line) takes the conservative view that when two or three versions fail at the same test case (a complete flight), they are considered as coincident failures regardless of whether or not they fail simultaneously and identically. This would be the case if no recovery routines were performed on the failed versions. The second curve (solid line) assumes a recovery mechanism is available to recover faulty program versions as long as the majority results are correct. Among the six remaining faults, ($\gamma$.12, $\nu$.3) are the only identical faults, producing incorrect majority results which are unrecoverable.

The results show that the 3-version configuration (with or without recovery) is generally more reliable than the single-version. However, that 3-version without recovery is less reliable than the single version when there are six faults remainingin the system. This suggests that the N-version approach without recovery mechanisms may be effective if individual versions are not reliable. If recovery mechanisms are available, the N-version approach can improve the reliability of the system significantly even if the individual versions are not considerably reliable. Note that one of the identical faults, $\nu$.3, was first detected and removed, leaving the remaining five independent faults whose erroneous results were all recoverable.

From the data presented in this empirical study, one may argue that in general the N-version approach is applied when ultra reliability is required, for instance, in the case that the threshold of the failure rate is less than $10^{-7}$, which does not seem to be the case shown in our experiment. We note that the reliability measurement is a function of exposure time. The shorter period of time the system is exposed to, the less chances failure will occur, i.e., the higher reliability can be reached. Since this pitch control function as a critical application is only a small portion of a complete flight control system, its exposure time is considerably shorter than what we have exercised. The objective of our experiment is to show the relative differences of the reliabilities; therefore, we did not intend to select an exposure time which can lead to an absolute ultra high reliability.

On the other hand, our results show that when ultra high reliability of the system is required and if its estimation is feasible [3], it is necessary to examine the coverage information collected during testing before applying any software reliability growth model to obtain reliability estimates. Since the existing reliability models tend to be too optimistic, they often overestimate the reliability of the software, and any over-estimation of the reliability can lead to severe risks and hazards in accepting the software. Our model resolves this problem by providing the answer for the question: "Does the reliability of the software satisfy the requirement and does the software need to be tested more, when the estimated reliability given by the model satisfies the reliability requirement of the software?"

## 4 Conclusions

We have introduced a technique that incorporates code coverage measurement in the estimation of software reliability for fault-tolerant software as well as non fault-tolerant software. Although in this paper we only use block coverage to extract effective test data, other coverage measures such as decision coverage, c-use, p-use can also be applied. This technique improves the applicability and performance of software reliability growth models, which gives the users a better understanding of the software quality and helps the developers conduct a more effective testing scheme. It indicates to the testers when a testing technique becomes ineffective and should be switched to another one, and when to stop testing without overestimating the achieved reliability.

Our experiments, which were conducted by a real world $N$-version software project, confirm the advantage of this technique. To investigate the relation between the strength of the coverage criteria used and the improvement of the estimation made by this technique, we evaluated software reliability predictions made for both a single-version software configuration as well as a 3-version software configuration. Our experimental results show that in both cases, our technique can track software reliability closely, and it can avoid an overestimation of software reliability toward the end of software testing.

Even if an $N$-version software configuration is used in our experiment, the results are generally applicable to other fault-tolerant software architectures like recovery blocks and $N$ self-checking software. We note that while reliabilities measured by traditional software reliability growth models are somewhat insensitive to the fluctuations in the operational profiles, reliabilities are, and our technique can capture this phenomenon faithfully. In future efforts to establish an acceptance criterion for ultra high reliable software systems, we believe our technique will provide a vital mechanism in selecting trustworthy software versions, avoiding redundant test efforts, and making conservative yet accurate reliability predictions.

## References


[1] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding. Software fault tolerance: An evolution. *IEEE Transactions on Software Engineering*, SE-11(12):1502–1510, December 1985.

[2] A. Avižienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[3] R. Butler and G. Finelli. The infeasibility of experimental quantification of life-critical software reliability. In *ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 66–76, New Orleans, Louisana, December 1991.

[4] M. H. Chen, P. Garg, A. P. Mathur, and V. J. Rego. Investigating coverage-reliability relationship and sensitivity of reliability estimates to errors in the operational profile. *Computer Science and Informatics Journal - Special Issue on Software Engineering*, 1995.

[5] M. H. Chen, M. K. Jones, A. P. Mathur, and V. J. Rego. T$\mathcal{E}$RSE: A tool for evaluating software reliability estimation. In *Proceedings of fourth International Symposium on software reliability engineering*, 1993.

[6] M. H. Chen, M. R. Lyu, and W. E. Wong. An emperical study of the correlation between code coverage and reliability estimation. In *IEEE Third International Symposium on Software Metrics*, Berlin, Germany, March 1996.

[7] M. H. Chen, A. P. Mathur, and V. J. Rego. Effect of testing techniques on software reliability estimates obtained using time-domain models. *IEEE transactions on reliability*, 44(1), March 1995.

[8] R. DeMillo, W. McCracken, R. Martin, and J. Passafiume. *Software Testing and Evaluation*. The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1987.

[9] J. B. Dugan and M. R. Lyu. System-level reliability and sensitivity analyses for three fault-tolerant system architectures. In *4th International Working Conference on Dependable Computing for Critical Applications*, pages 295–307, San Diego, CA, Januray 1994.

[10] D. E. Eckhardt and L. D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[11] A. L. Goel and K. Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, R-28(3):206–211, 1979.

[12] A. Grnarov, J. Arlat, and A. Avižienis. On the performance of software fault tolerance strategies. In *Proceedings 10th Annual International Symposium on Fault-Toelrant Computing*, pages 251–253, Keyto, Japan, October 1980.

[13] J. Horgan, S. London, and M. Lyu. Achieving software quality with testing coverage measures. *IEEE Computer*, 27(9):6–69, September 1994.

[14] W. Howden. *Functional Program Testing and Analysis*. McGraw-Hill, New York, 1987.

[15] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and analysis of hardware-and-software fault-tolerant architectures. *IEEE Computer*, 23:39–51, July 1990.

[16] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Architectural issues in software fault tolerance. In M. Lyu, editor, *Software Fault Tolerance*, pages 47–80. John Wiley & Sons, February 1995.

[17] M. R. Lyu, editor. *Handbook of Software Reliability Engineering*. McGraw-Hill Publishing Company and IEEE Computer Society Press, New York, 1995.

[18] M. R. Lyu and Y. He. Improving the n-version programming process through the evolution of a design paradigm. *IEEE Transactions on Reliability*, 42(2):179–189, June 1993.

[19] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and R. Skibbe. The relationship between test coverage and reliability. In *Proceedings of fifth International Symposium on software reliability engineering*, 1994.

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

[21] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *Proceedings Seventh International Conferrence on Software Engineering*, pages 230–238, Orlando, 1984.

[22] P. Piwowarski, M. Ohba, and J. Caruso. Coverage measurement experience during function test. In *Proceedings of the fifteenth International conference on Software Engineering*, pages 287–300, 1993.

[23] C. V. Ramamoorthy and F. B. Bastani. Software reliability - status and perspectives. *IEEE Transactions on Software Engineering*, SE-8(4):354–371, 7 1982.

[24] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[25] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.

[26] A. T. Tai, J. F. Meyer, and A. Avižienis. Performability enhancement of fault-tolerant software. *IEEE Transactions on Reliability*, 42(2):227–237, 1993.

[27] L. Tomek and K. Trivedi. Analyses using stochastic reward nets. In M. Lyu, editor, *Software Fault Tolerance*, pages 139–165. John Wiley & Sons, February 1995.

[28] M. A. Vouk. Using reliability models during teseting with non-operational profile. In *Proceedings of the second Bellcore/Purdue Symposium on Issues in software reliability estimation*, pages 103–110, 1993.

[29] M. A. Vouk, D. F. McAllister, and K. C. Tai. An experimental evaluation of the effectiveness of random testing of fault-tolerant software. In *Proc. Workshop on Software Testing*, Banff, Canada, IEEE CS Press, July 1986.


## Appendix

In this appendix we describe the technique that uses coverage information to enhance the existing SRGMs by extracting the effective portion of test efforts, where the test effort of test case $T_i$ is defined as $t_i$ - $t_{i-1}$, and $t_i$ is the time when the test case $T_i$ is executed. The effective test effort is obtained by multiplying the test effort by a compression ratio, i.e., $\rho_i$ * ($t_i$ -$t_{i-1}$). The compression ratio is defined as follows:

$$\rho_i = \begin{cases} 1 & \text{if } T_i \text{ increases coverage} \\ & \text{or } T_i \text{ causes a failure} \\ 0 \leq \rho_i < 1 & \text{otherwise} \end{cases} \quad (1)$$

Let $T_1, T_2, ...T_n$ be the test cases used during the testing process and $d_1, d_2, ..., d_n$ be the data recorded upon completion of each test case. The $d_i$s are represented by ordered triples $(t_i, c_i, f_i)$, for $i$=1,...,$n$, where $t_i$ is the testing time spent by $T_j$, where j = 1,..., $i$; $c_i$ is the cumulative coverage obtained up to $T_i$ and $f_i$ denotes cumulative failure experienced up to $T_i$. A test case $T_j$ is considered to be non-effective if $c_j = c_{j-1}$ and $f_j = f_{j-1}$; in other words, $T_i$ is non-effective if it does not increase any coverage and it does not cause the execution of the program to fail. Two vectors $v_i^1$ and $v_i^2$ are formed at each point $d_i$, for $i$= 1, 2,..., $n$, as:

$$\begin{aligned} v_i^1 &= (t_i - t_{i-1},\, 0,\, f_i - f_{i-1}) \\ &= (\delta t_i,\, 0, \delta f_i) \\ \text{and} & \\ v_i^2 &= (0,\, c_i - c_{i-1},\, f_i - f_{i-1}) \\ &= (0,\, \delta c_i,\, \delta f_i) \end{aligned}$$

If test case $T_{i+1}$ is a candidate for a non-effective test case, then $d_{i+1}$ will be projected orthogonally onto a point $\tilde{d}_i$ which is on the plane formed by the point $(t_i, c_i, f_i)$ and the two vectors, $v_i^1$ and $v_i^2$. Figure 6 depicts the geometrical interpretation of this projection, where $(t_i, c_i, f_i)$ and

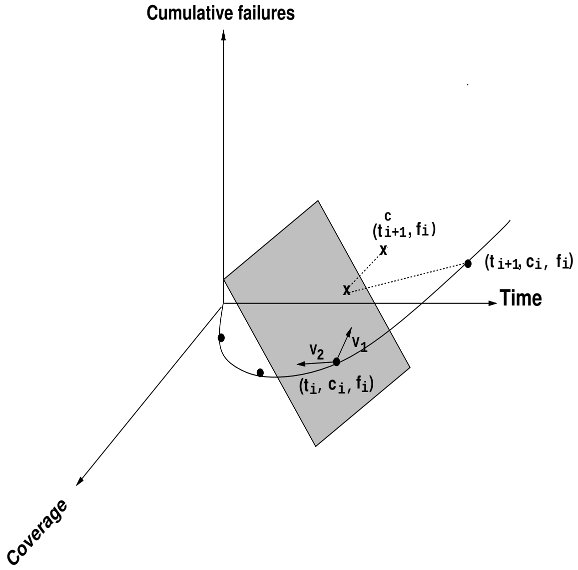


**Figure 6. Coverage enhanced data processing technique.**

$(t_{i+1}, c_i, f_i)$ are test data and $(t_{i+1}^c, f_i)$ is the projection of $(t_{i+1}, c_i, f_i)$ on the *Cumulative failure-Time* plane. The derived form of the new sequence $\tilde{d}_i$, for $i$= 1, 2,..., $n$, is given below.

$$\begin{aligned} (d_1, d_2, ..., d_n) \quad &\Longrightarrow \quad (\tilde{d}_1, \tilde{d}_2, ..., \tilde{d}_n) \\ \text{where} & \\ \tilde{d}_i \quad &= \quad \begin{cases} d_i = (t_i,\, c_i,\, f_i) & \text{if } T_i \text{ is effective} \\ (\rho_i * t_i,\, c_i,\, f_i) & \text{otherwise} \end{cases} \\ \text{and} & \\ \rho_i \quad &= \quad \frac{\alpha * \delta t_i^2 * \beta * \delta c_i^2 + \alpha t_i^2}{\alpha \delta t_i^2 + \beta * \delta c_i^2 + (\alpha * \delta t_i^2 * \beta * \delta c_i^2)} \end{aligned}$$

The $\rho_i$ is the *compression ratio* indicating the effective portion of the time interval $t_i$ and $\alpha$ and $\beta$ are two smoothing parameters which are program and model dependent and need to be adjusted for different data and models. To adjust these two parameters, we compare the difference between the reliability and its estimate at a given time instance. This instance can be any time during the the testing for small applications but it has to be after one half of the testing time for large applications. The time and the cumulative failures components of the new sequence $\tilde{d}_i$ are the data to be used by the SRGMs.