An Empirical Study of the Correlation between Code Coverage and Reliability Estimation*

Mei-Hwa Chen, Michael R. Lyu, and W. Eric Wong
{mhc@cs.albany.edu, lyu@research.att.com, ewong@bellcore.com}

Abstract

Existing time-domain models for software reliability often result in an overestimation of such reliability because they do not take the nature of testing techniques into account. Since every testing technique has a limit to its ability to reveal faults in a given system, as a technique approaches its saturation region fewer faults are discovered and reliability growth phenomena are predicted from the models. When the software is turned over to field operation, significant overestimates of reliability are observed. In this paper, we present a technique to solve this problem by addressing both time and coverage measures for the prediction of software failures. Our technique uses coverage information collected during testing to extract only effective data from a given operational profile. Execution time between test cases which neither increase coverage nor cause a failure is reduced by a parameterized factor. Experiments using this technique were conducted on a program created in a simulation environment with simulated faults and on an industrial automatic flight control project which contained several natural faults. Results from both experiments indicate that overestimation of reliability is reduced significantly using our technique. This new approach not only helps reliability growth models make more accurate predictions, but also reveals the efficiency of a testing profile so that more effective testing techniques can be conducted.

*Mei-Hwa Chen is an Assistant Professor in the Department of Computer Science, the State University of New York at Albany. Michael R. Lyu is a member of technical staff with AT&T Research Labs. Murray Hill, NJ 07974. W. Eric Wong is a research scientist with Bell Communications Research, Morristown, NJ 07960.

Keywords: time-domain models, software reliability, code coverage

1 Introduction

The reliability of a program is defined as the probability that the program does not fail in a given environment during a given exposure time interval [10]. It is an important metric of software quality. Since the late 1960s, a number of analytic models have been proposed to estimate software reliability [13]. Among them, the time-domain models, also called Software Reliability Growth models (SRGMs), have been the most popular and have been widely researched. These models make use of the failure history of a program obtained during testing and predict the field behavior of the program under the assumption that testing is performed in accordance with a given operational profile [5]. However, there are some fundamental problems with these models, such as the saturation effect existing in the testing process [2], and the difficulty in obtaining an actual operational profile.

Observations, both from empirical and analytical studies, show that the predictions made by the SRGMs tend to be too optimistic [1, 2, 4]. Many attempts have been made to improve the estimation made by the SRGMs. Some researchers have proposed that the test data must be pre-processed before they can be used by the SRGMs. Schneidewind's optimal selection method [14] excludes or gives little weight to the failure counts that are obtained from the early phase of the testing process. Li and Malayia [7] propose a data smoothing method by grouping data point with fixed group size or failure intensity lumps to filter out short term noise and present a weighted least square estimation to overcome the estimation bias. Chen *et al.* [2] define an effective test effort in terms of test coverage. In their model a testing effort is considered as effective only if it forces the program to exercise the uncovered portion or it causes the program to fail. The testing efforts that neither increase any test coverage nor detect any fault in the program are either discarded or reduced appropriately.

Another group of researchers postulate that coverage information should be used instead of testing time to overcome the difficulty of obtaining an operational profile of the software. They investigate the relationship between test coverage and reliability estimation and propose reliability models that are based on test coverage. Vouk [15] investigates the relation between test coverage and fault detection rate. He proposes that the fault detection rate with respect to coverage is proportional to the coverage and the effective number of residual faults. In his experiment, the strength of fault detectability with respect to different coverage criteria is compared. Piwowarski *et al.* [12] observe the coverage measurements on large projects during a function test and derive a coverage-based reliability growth model which is isomorphic to the Goel-Okumoto NHPP model and the Musa execution time model. Malayia *et al.* [9] model the relation among test effort, coverage and reliability and propose a coverage based logarithmic model that relates a test coverage measurement with fault coverage.

In this paper we present a technique that models the failure rate with respect to both test coverage and testing time. A justification to this technique is as follows. Suppose a software is tested successfully against a suite of test cases. Without any additional test cases being created, this software may be continuously tested using the same test suite and not result in any failure. If such a failure rate which is with respect to the testing time only is applied to the SRGMs, an obvious reliability overestimate will be observed. To overcome this problem, our technique uses test coverage to adjust the failure rate before it is applied to the SRGMs. In other words, the time intervals between failures are adjusted if any redundant testing effort such as repeating the same test cases is involved. By using this coverage enhanced pre-processing technique, we applied the extracted test data to the Goel-Okumoto NHPP model [6] and the Musa-Okumoto Logarithmic model [11] and observed an improvement of the estimation made by both models.

The details of this technique are described in the next section. In Section 3 we describe the experiment that we conducted with an industrial program and we conclude our observation and list possible future directions in Section 4.

2 Methodology

The relationship between coverage and software reliability has been studied by many researchers. Empirical studies have shown that fault detectability is correlated to test coverage; consequently, software reliability is correlated to test coverage [1, 16, 17]. This experimental evidence motivated and supported our belief that test coverage information should be used in reliability estimation. The nature of SRGMs is to give the estimation by using the time dependent failure data. As testing proceeds, the test cases generated in the latter phase are less likely to cause the program to execute the uncovered portion and detect faults in the program than in the earlier phase. Therefore, the time between failures increases as testing time increases and so do the reliability estimates made by the SRGMs.

However, the reliability of the software increases only in the case that the number of faults in the software is reduced. Therefore, we can expect that the more redundant test efforts that 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. In our model, the coverage information is used to determine the effectiveness of a test effort.

2.1 The model

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_is 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_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

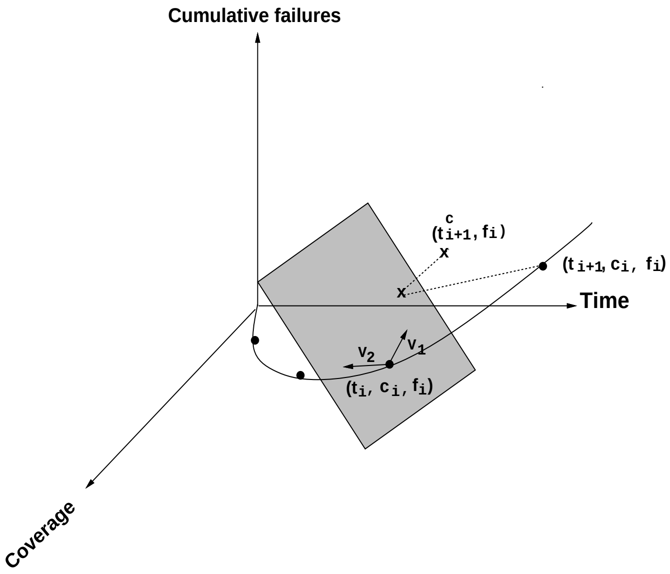

Figure 1: Coverage enhanced data processing technique.

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 1 depicts the geometrical interpretation of this projection, where (t_i, c_i, f_i) and (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.

$$(d_1, d_2, ..., d_n) \Longrightarrow (\tilde{d_1}, \tilde{d_2}, ..., \tilde{d_n})$$

where

$$\tilde{d_i} = \begin{cases} d_i = (t_i,\ c_i,\ f_i) & \text{if } T_i \text{ is effective} \\ (\rho_i * t_i,\ c_i,\ f_i) & \text{otherwise} \end{cases}$$

and

$$\rho_i = \frac{\alpha * \delta t_i^2 * \beta * \delta c_i^2 + \alpha t_i^2}{\alpha \delta t_i^2 + \beta * \delta c_i^2 + (\alpha * \delta t_i^2 * \beta * \delta c_i^2)}$$

The ρ_i is the *compression ratio* indicating the effective portion of the time interval t_i and α and β 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.

The rationale behind this approach is as follows. The v^1 vector describes the failure increasing pattern with respect to testing time, and the v^2 vector describes the failure increasing pattern with respect to coverage. Both time and coverage are crucial factors that affect the prediction of failures, so we incorporate the coverage information to extract the effective test efforts. Simulation results show that through the coverage enhanced process the Goel-Okumoto model and the Musa-Okumoto model lead to more accurate reliability estimates. These results are discussed in the next section.

2.2 Validation

In this section we describe the experiments which we have conducted under a simulation environment, T$\mathcal{E}$RSE [3]. Symbols M-O and G-O represent the Musa-Okumoto [10] and the Goel-Okumoto [6] models for reliability estimation, respectively. The procedures used in our study are listed below.

(1) Generate a program flow graph F with 1000 nodes.

(2) Annotate F with faults by assigning fault infection probability and fault propagation probability to each node.

(3) Test F by using the random testing technique with respect to a uniform profile. Faults that are responsible for the failure are removed during the debugging.

(4) Apply the failure data collected in Step (3) to SRGMs to obtain reliability estimates.

(5) Use a coverage enhanced technique to exclude non-effective testing efforts in Step (3) and then apply such extracted failure data to the models for reliability estimation.

(6) Compute reliability by simulating the execution of F with respect to the same profile used in Step (2).

The two smoothing factors, α and β, were computed at 240,000 units of testing time, and the two models, G-O model and M-O model were used to estimate the reliability of F. Figure 2 shows the reliability estimates obtained by applying the original data to the G-O model (labeled G-O) and the M-O model (labeled M-O) and by applying the extracted data to both models (labeled G-O^e and M-O^e). The estimates were compared with the reliability computed in Step (6) (labeled R).

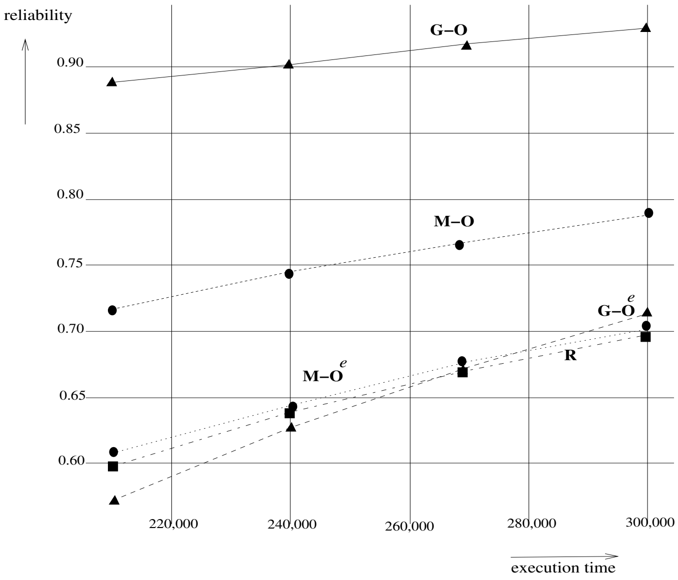

Figure 2: Reliability and its estimates obtained from the G-O model and the M-O model for a program flow graph with 1000 nodes.

The results show that at 270,000 units of testing time our technique reduces the overestimate made by the G-O model from 0.23 (33.7%) to 0.004 (0.55%), and that made by the M-O model from 0.073 (10.55%) to 0.00 (0%). Similarly, at testing time equal to 300,000 units, the overestimate is reduced from 0.205 (28.3%) to 0.0013 (0.18%) and 0.057 (7.8%) to -0.01 (-1.37%), respectively, for the G-O and M-O models. We conclude that the reliability overestimates made by the Goel-Okumoto model and the Musa-Okumoto model can be significantly reduced by considering only the effective testing efforts.

3 Application

3.1 The Autopilot Project

In order to demonstrate our technique for real world applications, we selected the *autopilot* project which was developed by multiple independent teams at the University of Iowa and the Rockwell/Collins Avionics Division [8]. The application program is an automatic flight control function for the landing of commercial airliners that has been implemented by the avionics industry. The specification can be used to develop the software of a flight control computer (namely, *autopilot*) for a real aircraft, given that it is adjusted to the performance parameters of a specific aircraft. All algorithms and control laws in the application are specified by diagrams which have been certified by the Federal Aviation Administration. The *pitch control* part of the automatic landing problem, i.e., the control of the vertical motion of the aircraft, is selected for the project. The major system functions of the pitch control and its data flow are shown in Figure 3.

In this application, the autopilot is engaged in the flight control beginning with the initialization of the system in the altitude hold mode, at a point approximately ten miles (52800 feet) from the airport. Initial altitude is about 1500 feet, initial speed 120 knots (200 feet per second). Pitch modes entered by the autopilot/airplane combination, during the landing process, are the following: altitude hold, glide slope capture, glide slope track, flare, and touchdown.

As shown in Figure 3, the autopilot application receives airplane sensor inputs (denoted by "I" in the figure) for its processing. These inputs include altitude, altitude rate, vertical acceleration, radio altitude, glide slope deviation, model valid flag, pitch attitude, pitch attitude rate, flight path, equalization, and signal display indicator. A subset of these inputs is processed by each of the eight autopilot major components: barometric altitude complementary filter, radio altitude complementary filter, glide slope deviation complementary filter, mode logic, altitude hold control, glide slope capture and track control, flare Control, command Monitor, and display.

The *complementary filters* preprocess the raw data from the aircraft's sensors. The *barometric altitude* and *radio altitude complementary filters* provide estimates of true altitude from various altitude-related signals, where the former provides the altitude reference for the altitude hold mode, and the latter provides the altitude reference for the flare mode. The *glide slope deviation complementary filter* provides estimates for beam error and radio altitude in the glide slope capture and track modes. Pitch mode entry and exit is determined by the mode logic equations, which use filtered airplane sensor data to switch the controlling equations at the correct point in the trajectory.

Each control law consists of two parts, the outer loop and the inner loop, where the inner loop is very similar for all three control laws. The altitude hold control law is responsible for maintaining the reference altitude, by responding to turbulence-induced errors in attitude and altitude with an automatic *elevator* command that controls the vertical motion. As soon as the edge of the glide slope beam is reached, the airplane enters the glide slope capture and track mode and begins a pitching motion to acquire and hold the beam center. A short time after capture, the track mode is engaged to reduce any static displacement towards zero. Controlled by the glide slope capture and track control law, the airplane maintains a constant speed along the glide slope beam. Flare logic equations determine the precise altitude (about 50 feet) at which the Flare mode is entered. In response to the flare control law, the vehicle is forced along a path which targets a vertical speed of two feet per second at touchdown.

Each program checks its final result (elevator command of each lane, or *land command*) against the results of the other programs. Any disagreement is indicated by the command monitor output, so that a supervisor program can take an appropriate action. The display continuously shows information about the autopilot on various panels. The current pitch mode is displayed for the information of the pilots (mode display), while the results of the command monitors (fault display) and any one of sixteen possible signals (signal display) are displayed for use by the flight engineer.

Upon entering the touchdown mode, the automatic portion of the landing is complete and the system is automatically disengaged. This completes the automatic landing flight phase. In summary, this application could be classified as a computation-intensive, real-time system.

3.2 Program Characteristics and Fault Description List

Five program versions developed in the autopilot project were used in our experiments. Table 1 shows a comparison of these five versions with respect to some common software metrics such as the: (1) number of lines excluding comments and blank lines (LOC); (2) number of executable statements (STMT); (3) number of programming modules (MOD); (4) mean number of statements per module (STM/M); (5) number of calls to modules (CALL); (6) number of global variables (GVAR); (7) number of local variables (LVAR); (8) number of blocks (BLOCK); and (9) number of decisions (DECI).

Table 2 shows the details of the faults found in these versions. The first column represents the id of a fault, which is composed of the program version (in Greek) and a sequence number. The second column indicates the testing phase in which the fault was detected including UT for unit testing, IT for integration testing, AT for acceptance testing, and FI for injected hypothetical faults. The next column identifies the source component where the fault is located, and the last column provides a short description of the fault.

Table 1: Software metrics for the five Program versions.

Metrics	γ	ϵ	κ	λ	ν
LOC	1229	895	1251	2520	1070
STMT	708	706	640	1366	810
MOD	11	6	17	17	24
STM/M	64	101	38	80	35
CALL	123	16	31	626	106
GVAR	55	101	7	0	423
LVAR	179	86	376	402	258
BLOCK	711	531	367	1132	473
DECI	250	320	286	357	237

Table 3: A Complete flight simulation scenario.

Flight Mode	Time (sec)	Distance (ft)	Altitude (ft)
Altitude hold	0.00	52800	1500
Glide slope capture	86.70	35460	1500
Glide slope track	96.65	33470	1475
Flare	258.95	1000	45
Touchdown	264.10	0	10

3.3 Testing and Debugging

Flight simulation testing of the autopilot application represents various execution scenarios in a feedback loop, where different flight modes will be entered and exercised. Table 3 shows the different flight modes that will be encountered in a complete flight (a duration of about 264 seconds). The second column shows the time when the flight mode is entered. The distance (with respect to the expected touchdown point at the airport) and the altitude of the airplane appear in the last two columns, respectively.

The initial conditions for each flight in our experiments are determined by the five variables listed in Table 4. Based on these variables, a test case composed of about 5280 autopilot program iterations can be constructed. Airplane sensor inputs for each iteration are generated from an airplane model to simulate the complete duration of a flight.

A super program which consists of the five versions, shown in Table 1, was used in our experiments to collect the failure data. The sequence of testing and debugging is given below.

Fault identifier	Detection phase	Source file	Brief Description
$\gamma.1$	IT	glideslp.c	Innerloop not initialized when first executed
$\gamma.2$	IT	bae_gscf.c	I8 not initialized when first entered
$\gamma.3$	AT	flare.c	Incorrect computation of a switch condition in flare outer loop
$\gamma.4$	AT	flare.c	RAE instead of H0 used when computing the value of THCI
$\gamma.5$	FI	flare.c	Flare flag is not checked when doing a computation
$\gamma.6$	FI	alt_hold.c	FPDC is subtracted instead of added in computing THCI
$\gamma.7$	FI	autoland.c	Wrong parameter when calling VOTEINNER
$\gamma.8$	FI	innerlp.c	FPEC is subtracted instead of added in computing SU3
$\gamma.9$	FI	innerlp.c	Wrong predicate condition (“>” instead of “<”)
$\gamma.10$	FI	innerlp.c	Wrong predicate condition (“>” instead of “<”)
$\gamma.11$	FI	innerlp.c	Wrong predicate condition (missing negation “!”)
$\gamma.12$	FI	mathutil.c	Wrong math operator (“-” instead of “+”)
$\gamma.13$	FI	racf.c	Wrong constant value
$\gamma.14$	FI	racf.c	Wrong math operator (“-” instead of “+”)
$\epsilon.1$	UT	modelogic.c	Incorrect initialization of filter F1
$\epsilon.2$	AT	modecontl.c	Incorrect initialization of Ho and Hdo
$\epsilon.3$	AT	modecontl.c	Incorrect initialization of I1 at flare
$\epsilon.4$	IT	filters.c	K0,K2,K3 values not set to static
$\epsilon.5$	AT	main.c	Wrong sequence voteinner during TOUCHDOWN
$\kappa.1$	IT	CLInner.c	Wrong initialization of variable from glide slope capture mode to glide slope track mode
$\kappa.2$	IT	GSCF.c	Wrong algorithm for different modes
$\kappa.3$	IT	GSCTOuter.c	Wrong initialization algorithm for variables for mode change
$\kappa.4$	AT	macros.h	Wrong algorithm in rate limiter LR1
$\lambda.1$	IT	autoland.c	Incorrect initialization sequence in the altitude hold mode
$\lambda.2$	IT	compFil.c	< instead of <= in comparison of frame counter
$\lambda.3$	AT	fclol.c	Incorrect computation of switch condition in flare outerloop
$\lambda.4$	UT	innerloop.c	Wrong computation sequence for SW2 in innerloop
$\lambda.5$	UT	utils.c	Incorrect logic branch
$\nu.1$	IT	GS_Outer.c	Incorrect computation of SW3 during mode transition
$\nu.2$	AT	Inner.c	Incorrect initialization of LR2 during mode transition

Table 2: Fault classification and characteristics for the program versions.

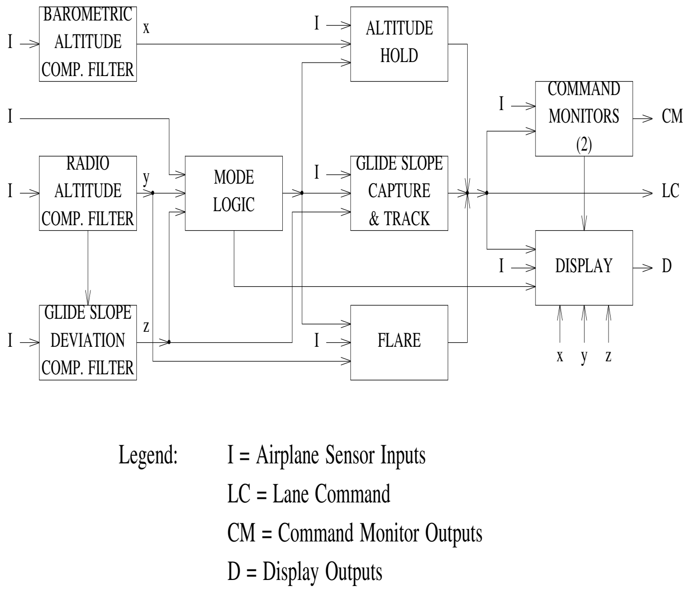

Figure 3: Pitch control system functions and data flow diagram.

Step 1: Generate a test pool $\mathcal{D}$

Step 2: Set the flag TEST_REPEAT to *false*

Step 3: **if** (TEST_REPEAT is *false*)
then (a) Select a test case t from $\mathcal{D}$ according to a uniform distribution profile
else (b) Re-use the test case t saved in Step (5)(d)

Step 4: Test the super program against t by executing one of the five versions (say $\mathcal{P}$) selected according to a uniform distribution profile

Step 5: **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 the flag TEST_REPEAT to *true*
(d) Save t for re-use in Step (3)(b)
else Set the flag TEST_REPEAT to *false*

Step 6: Goto Step (3)

One important characteristic of this testing and debugging process is that $\mathcal{P}$ may be executed against the same t more than one time. For example, suppose the first execution of $\mathcal{P}$ on t fails at time equals to p. After the faults responsible for this failure are removed, $\mathcal{P}$ is executed again on t. Suppose it fails at time equals to q, with $q > p$. Since the simulation is not yet completed, $\mathcal{P}$ should be re-executed against t after debugging. Such a process continues until $\mathcal{P}$ succeeds on t. Hereafter, we refer to the super program as the program.

Table 4: Initial flight conditions.

Variable	Range
Initial attitude	[-10^o, 10^o]
Initial pitch attitude	[-15^o, 15^o]
Initial height	1500 ft
Initial distance	52800 ft
Vertical wind turbulence	[-10 ft/sec, 10 ft/sec]

3.4 Reliability estimation

Figure 4 shows the reliability estimates obtained by applying the original data, collected from the testing and debugging process described in the previous section, 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: 265 seconds.

Reliabilities measured as the ratio of the number of failures to the number of executions were computed at testing time equal to 6274.7, 7857.2 and 8025.1 seconds, respectively. These three points were selected because it was at these that the last three fault correction activities occurred. While such reliability was computed, 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.

In applying the coverage enhanced technique, we used block coverage measurement. For the two smoothing constants, α was 1000 and 3000, β was 0.005 and 0.00046, for the G-O model and the M-O model, respectively. In the early phase of the testing, we observed that the estimation was not stable even using the data which had been processed by our technique. The number of data points was not large enough to support the estimation process, so we compared the reliability and its estimates only after 6000 seconds.

For the G-O model the differences between the reliability and its estimates ranged from -0.03 to 0.105

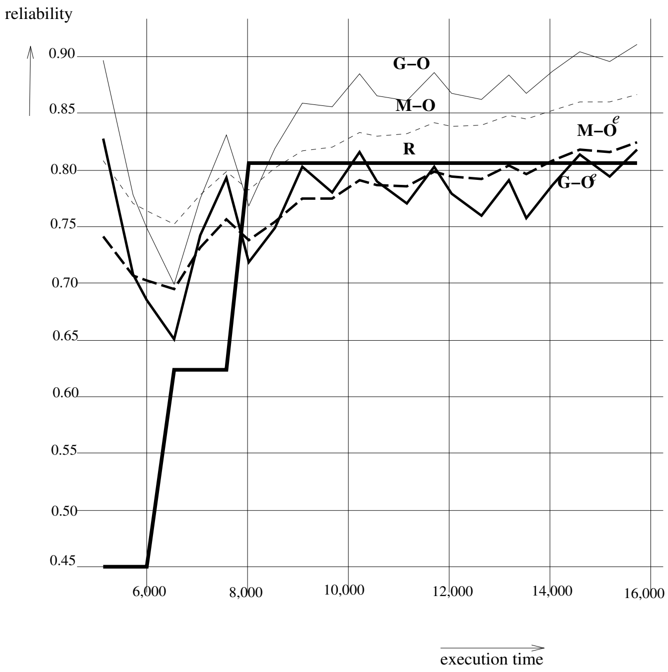

Figure 4: Reliability and its estimates obtained from the G-O model and the M-O model for the *autopilot* project.

(4% to 13%). If the coverage enhanced technique was applied, they ranged from -0.08 to 0.012 (-10.8% to 1.5%). For the M-O model, the differences ranged from -0.02 to 0.06 (-2.8% to 7.5%) and the results obtained after applying the technique were -0.06 to 0.018 (-8.4% to 2.3%). Comparing the differences at the end of the testing process, we observed that the overestimates made by the G-O model reduced from 13% to 1.5%, and those made by the M-O model reduced from 7.5% to 2.3%.

The results obtained from this study show that the coverage enhanced technique did improve the reliability estimation made by the SRGMs and brought the estimates much closer to the actual reliability.

4 Conclusions and Future Plan

We have introduced a technique that incorporates test coverage measurement in the estimation of software reliability. This technique improves the applicability and performance of software reliability growth models, which gives the user a better understanding of the software quality and helps the developer conduct a more effective testing scheme. It indicates to the tester 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 conducted by a simulated program and a real world project confirm the advantages 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 plan to apply stronger coverage measurements, like branch, all-uses and mutation coverage, for more empirical studies of this technique on industrial projects. Furthermore, while reliabilities measured by SRGMs are somewhat insensitive to the fluctuations in the operational profiles, the actual reliabilities are sensitive, and our technique can capture the phenomenon faithfully. We would like to quantitatively formulate this relationship by a sensitivity study. Finally, we hope to establish the criteria that will meet a reliability threshold specified for an ultra high reliable software system by applying our technique to avoid redundant test efforts.

5 Acknowledgments

We express our special thanks to Dr. J. R. Horgan at Bellcore and Professor A. P. Mathur at Purdue University for their guidance, advice and valuable comments on this work.

References

[1] 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.

[2] M. H. Chen, J. R. Horgan, A. P. Mathur, and V. J. Rego, "A time/structure based model for estimating software reliability," Technical Report SERC-TR-117-P, Purdue University, July 1992.

[3] 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.

[4] 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.

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

[6] 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.

[7] N. Li and Y. K. Malaiya, "Enhancing accuracy of software reliability prediction," in *Proceedings of fourth International Symposium on software reliability engineering*, 1993.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] N. F. Schneidewind, "Optimal selection of failure data for predicting failure counts," in *Proceedings of fourth International Symposium on software reliability engineering*, pages 142–149, 1993.

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

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

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