

Applying Testability to Reliability Estimation

Mark C. K. Yang
Department of Statistics
University of Florida
yang@stat.ufl.edu

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

Alberto Pasquini
ENEA
Rome, Italy
pasquini@casaccia.enea.it

Abstract

The purpose of this report is to implement the idea of using testability to estimate software reliability. The basic steps involve estimating testability, evaluating how well software was written, and assessing the relationship between testing and usage. Results from these steps along with operational profiles are used to estimate software reliability. This paper describes an application of this method to evaluate the reliability of a real software system of about 6000 lines of executable code and discusses the results of such an estimation. The results are also compared with those obtained by using two reliability growth models.

Keywords: *software testing, software reliability, testability, fault detection*

1 Introduction

Many recent studies have considered software reliability from the point of view of testability. This is quite different from the approach taken by conventional software reliability growth models. These models do not take into account important information that can be derived from the relationship between testing and software execution, such as the coverage (structural or functional) obtained during testing. Recent research [26] has shown a strong correlation between reliability and coverage criteria, although it is very difficult to quantify this relation. Dalal et al. [6] examined this relationship between unit-test statement coverage and system-test faults later attributed to those units. Piwowarsky et al. [23] observed that fault removal rate and code coverage are closely related by a somewhat linear function. Malaiya et al. confirmed this relationship [15] and proposed a new model to relate test coverage to software reliability [17]. Del Frate et al. [10] investigated the relationship between coverage and reliability in an experimental context, using several real programs seeded with artificial and

real faults. In their extensive experiments a clear positive, almost linear relation, between coverage and reliability was found in all the programs considered, independent of their size, the fault distribution and the coverage criterion used (statements, blocks, branches, c-uses and p-uses). Chen, Lyu and Wong [3, 4] also introduced a technique that incorporates test coverage measurement into the estimation of software reliability.

On the other hand, most conventional reliability growth models are based on the assumption that software is tested with test cases selected randomly according to its future operative usage. Unfortunately, this assumption is often not realistic [30]. Test engineers may prefer completely different testing strategies to random testing. This is because random test selection may miss certain special test cases which are effective in detecting faults. Information obtained from these different testing strategies is not used by reliability growth models.

Another point worth noting is that even though a few studies [9, 11] reported that random testing can be as efficient as other testing strategies, this is still debatable.

To verify whether a non-random test set is indeed better, we need to measure its fault-detection efficiency. Voas et al. [28] and Bertolino and Strigini [1] defined testability as the probability that a faulty program is detected by taking inputs from a certain input distribution. While Howden and Huang [13] defined the detectability as the probability that the testing method will detect a fault in a randomly selected program, given that the program contains faults. There are also many less formal evaluations of testability (e.g. in [18, 22]). Regardless how testability is defined, one can easily sense its usefulness. If a testing strategy has a high fault detectability, it is a good strategy; and software tested by using this strategy with good standing is more likely to be reliable.

Although the studies cited above have envisioned this great potential, it has not been applied to real world software. How can the testability be estimated in practice? How can it be used in reliability estimation if the test set is not a

random sample from an operational profile? The purpose of our study was to apply the concept of testability to white-box testing and reliability estimation. A few gaps between theory and practice were filled by using reasonable assumptions. Results of an application of our method are discussed and compared with those obtained by using two reliability growth models.

The remainder of this paper is organized as follows. Section 2 explains our methodology. Section 3 gives a description of how parameters can be estimated followed by a numerical example. Section 4 presents a case study to show the merits of using our method. Our conclusion and the direction of future research appear in Section 5.

2 General Concepts and Definitions

The *testability* of a program can be viewed as the likelihood of a program failing if it contains at least one fault. Bertolino and Strigini [1] define testability as

$$Testab_{ABS} \equiv P\{\text{reject}|\text{prob.distribution of inputs, oracle, faulty}\} \quad (1)$$

with the assumption that the oracle is not perfect in that it may reject a correct execution or accept an incorrect execution. Voas et al. [28], on the other hand, define testability without taking oracle into account as

$$Testab_{HV} \equiv P\{\text{reject}|\text{prob.distribution of inputs, faulty}\} \quad (2)$$

If the oracle is a perfect I/O oracle, then $Testab_{ABS}$ is equal to $Testab_{HV}$ for a given distribution of the inputs. We follow Voas's assumption that the program under study is memory-less and its test outcome can be precisely decided by comparing its outputs against the specification. In this paper, the "prob.distribution of inputs" is taken according to either the testing strategy during the testing phase or the operational profile during the usage phase. The former is used to compute the *testability* and the latter for the *revealability*.

In this section and the next section (Section 3: Estimation), we explain our method by measuring testability at the *basic block* level rather than the entire program. However, our method can be applied at other units such as a single function, a group of functions, a file, or a group of files, and not only at the block level (i.e. although the derivations of these equations listed below is done with respect to blocks, they also work at different granularity level). Our derivation is based on the concept that each unit contains at most one fault, or at most one fault of each kind when the faults are classified into categories. Thus, the number of units should be made as small as possible so long as the tester think that

the at most one fault per unit assumption is valid. Too many units makes the computation messy as we will see from the estimation formulae. An example of this is in our case study in Section 4 where each function is chosen as a basic reliability unit.

Let the program be partitioned into N disjoint blocks. Our definition of Equation (1) is

$$p_i \equiv Pr\{\text{reject}|\text{block } i \text{ is executed based on a testing strategy, faulty}\} \quad (3)$$

where p_i is the *testability* for the i th block. The statement that block i is rejected is equivalent to that the fault in this block is detected. The testability p_i is contingent upon the execution of block i because if block i in (3) is not executed, its faulty status cannot be determined. However, even if the block is faulty and executed in testing, there is only a probability that the fault can be discovered. Obviously, p_i depends on the block contents and the testing strategy. In theory, if each block is tested long enough, we should be able to estimate all the p_i 's, but in practice, with a limited amount of testing, we have to group blocks into a few types and assume that the same types of blocks share the same p_i . For the purpose of explanation, we use a single p to represent the testability for all the blocks. Although the extension from a single p to multiple p 's is straightforward, the complexity in notation detracts from the discussion of the basic ideas. Multiple p_i appears in Section 3.5.

With an approach similar to the one proposed in [1], if block i has been tested n_i times without failure, then the probability that it is faulty is

$$\alpha_i(i) = \frac{\alpha_0(i)(1-p)^{n_i}}{1-\alpha_0(i)+\alpha_0(i)(1-p)^{n_i}} \quad (4)$$

where $\alpha_0(i)$ is the prior probability before testing that the i th block is faulty. A proof is given in the appendix A. Equation (4) can also be used for a block that needs to be debugged. Although we do not assume that such a block will be perfect after debugging, we assume that it is at least as good as those which have been tested without failure. Though $p < 1$ is an undesirable property in testing, it occurs in practice because we can never expect that all faults will be detected after all the blocks are executed once. We define the *revealability* of block i as

$$q_i \equiv Pr\{\text{reject}|\text{block } i \text{ is executed based on the users' operational profile, faulty}\} \quad (5)$$

where block i is rejected if it produces an incorrect output. Although testing may not require an operational profile, the reliability of a program must depend on the operational profile of a particular application. Let the operational profile be denoted by $(X, \phi(x))$ where X contains all the inputs and

$\phi(x)$ is the distribution of how frequently $x \in X$ is used. Suppose q_i and $\alpha_t(i)$ have been estimated. Based on this, the probability that input x will produce a correct output is

$$\pi_t(x) = \prod_{i \in S(x)} [1 - \alpha_t(i)q_i] \quad (6)$$

where $S(x)$ contains all the blocks executed by input x . Equation (6) assumes that all blocks act independently from the point of view of reliability. Counter examples against the independence assumption can be constructed, but it can be considered the first degree approximation. Malfunctioning among components has also been assumed independent in most hardware assembly and software data flow analysis [5, 16, 24]. For an operational profile $(X, \phi(x))$, the reliability expressed as the probability of having no failure in the next execution, is

$$R_t = \int_X \pi_t(x)\phi(x)dx. \quad (7)$$

Equation (7) should be interpreted as the Lebesgue integral, I.e., if the input domain contains only finite number of possible inputs, one can replace the integral in Equation (7) by summation. To estimate R_t , we need to know p , $\alpha_0(i)$, q_i , $S(x)$ and $\phi(x)$. Details of this method are described in the next section.

3 Estimation

In this section, we explain how p , $\alpha_0(i)$ and R_t for a given $(X, \phi(x))$ can be estimated. A numerical example is provided to elucidate these estimations. Once again a single p is used in the beginning. Since we work with white-box testing, we assume that the tester has a tool such as ATAC [12] which can record all the executed blocks in each test.

3.1 Estimation of p

To estimate p , we follow the suggestion in [27] using “seeded” faults (i.e. mutants). Common fault types described in [2, 8, 14] were injected into blocks. Earlier studies [7, 21] indicated that there is a great consistency between errors generated by these first-order mutation-like faults and by real faults. In addition, these simple faults can create erroneous behaviors as complex as those identified for the real faults. Nevertheless, we are also aware of some critiques of using this approach as discussed in [1]. To respond to some of the concerns, a case study using a space application with real faults recorded in the error-log is presented in Section 4 to show that, at least in this example, our method is not limited by the constraints of mutation testing.

As in mutation testing, the correct output of the program is not needed. A comparison of the outputs from the non-mutant and the mutant programs is sufficient to determine

whether a mutant can be killed (detected). If the outputs disagree, then the mutant is killed. Otherwise, it is still alive. It is possible that some mutants are still alive after a long time of testing, but time and/or budget constraints do not allow more testing.

The average number of tests required to detect these faults (i.e. kill these mutants) can be estimated. Suppose, on the average, it takes ν passes to discover a faulty block, then p , defined in Equation (3), can be roughly estimated as $\hat{p} = 1/\nu$.¹ More rigorously, let m_f be the total number of faults injected, that is, the total number of mutants to be killed. Of these, r faults are detected during the testing and $m_f - r$ remain undetected. Let ν_j be the average number of test cases used to detect the j th fault, $j = 1, 2, \dots, r$, and $\nu_{r+1}^+, \dots, \nu_{m_f}^+$, be the number of test cases executed for testing those live mutants when the testing stops. The maximum likelihood estimate of the testability p will be

$$\hat{p} = \frac{r}{\sum_{i=1}^r \nu_i + \sum_{j=r+1}^{m_f} \nu_j^+}. \quad (8)$$

The derivation is given in Appendix A. Once p is estimated, all the injected faults are removed and the same testing strategy is now used to test and debug the program.

3.2 Estimation of $\alpha_0(i)$

The prior probability $\alpha_0(i)$ can be either known from previous testing and usage, or unknown. In the latter case, we first assume $\alpha_0(i) = \alpha_0$ for all i 's with respect to a single p . The assumption on single α_0 will also be relaxed in Section 3.5. Intuitively, α_0 is related to how well the program was written. Its estimate depends on the testing effort and the number of faults found during the testing. For a given block, the probability that a fault in it will be detected at the n th test is $\alpha_0(1-p)^{n-1}p$, where p is the testability. Similarly, the probability that no fault is found in that block after n tests is $(1-\alpha_0) + \alpha_0(1-p)^n$.

Suppose there are N blocks in the program and s of them have been detected with faults. Let the rest of $N-s$ blocks, where no faults have been found in them, be tested successfully by $n_1^+, n_2^+, \dots, n_{N-s}^+$ tests, respectively. Then, the maximum likelihood estimate of α_0 is the solution to the following equation.

$$\frac{s}{\alpha_0} = \sum_{i=1}^{N-s} \frac{1 - (1-p)^{n_i^+}}{(1-\alpha_0) + \alpha_0(1-p)^{n_i^+}}. \quad (9)$$

The derivation is again given in Appendix A. Equation (9) can only be solved numerically, except for the special case when all $n_i^+ = n^+$,

$$\hat{\alpha}_0 = \frac{s}{N[1 - (1-p)^{n^+}]} \quad (10)$$

¹Following convention, the notation \hat{p} is an estimate of p . The same convention is also applied for other parameters.

This can also serve as the initial value in solving Equation (9) by letting n^+ in Equation (10) be the average value of the n_i^+ 's.

If the program has been previously tested, such information should be used to form a prior distribution, denoted by $h_i(\alpha)$, for $\alpha_0(i)$. Then the posterior distribution of $\hat{\alpha}_t(i)$, based on the above likelihood equation (i.e. Equation (20)) is

$$g_i(\alpha) \propto h_i(\alpha) \alpha^s \prod_{i=1}^{N-s} [1 - \alpha + \alpha(1 - \hat{p})^{n_i^+}] \quad (11)$$

The point estimate of $\alpha_t(i)$ is the α that maximizes Equation (11). A commonly used prior distribution for a proportion parameter is the beta distribution. For example, if a block has been tested τ times without failure, we may let

$$h_i(\alpha) = \frac{(1 - \alpha)^\tau}{1 + \tau}, \quad 0 < \alpha < 1 \quad (12)$$

A derivation of this prior distribution can be found in [19].

3.3 Estimation of R_t for operational profile $(X, \phi(x))$

The reliability function $\pi_t(x)$ in Equation (6) can be computed if $S(x)$ and q_i are known. Once an input x is selected, we can find out $S(x)$ by running x and recording the executed blocks. It is reasonable to assume that $q_i \leq p$ because test cases selected by testing experts at the testing stage should have, on the average, a higher probability to reveal more faults than those cases used by the users after the program is tested. Thus, by letting $q_i = \hat{p}$, we obtain a conservative estimate of the reliability. From Equation (6), $\pi_t(x)$ can be estimated as

$$\hat{\pi}_t(x) = \prod_{i \in S(x)} [1 - \hat{\alpha}_t(i) \hat{p}] \quad (13)$$

where $\hat{\alpha}_t(i)$ is the $\alpha_t(i)$ in Equation (4) with p and $\alpha_0(i)$ replaced by their estimates \hat{p} and $\hat{\alpha}_0$, respectively.

To estimate R_t in Equation (7), we can use a random sample from X according to the distribution function $\phi(x)$. If W is a sample taken, then R_t can be estimated by

$$\hat{R}_t = \frac{1}{|W|} \sum_{x \in W} \hat{\pi}_t(x) \quad (14)$$

where $|W|$ denotes the sample size of W . Sometimes it is more convenient to take a sample when the input space X is partitioned according to different applications. Let L be the number of partitions, and ϕ_k be the the frequency of partition k being selected and $R_t(k)$ be the reliability for the input from this partition. Then the overall reliability of the software is

$$R_t = \sum_{k=1}^L \phi_k R_t(k). \quad (15)$$

If a random sample from each partition is taken, then an estimate for $R_t(k)$, denoted by $\hat{R}_t(k)$, can be computed by Equation (14), and R_t can be estimated by

$$\hat{R}_t = \sum_{k=1}^L \phi_k \hat{R}_t(k). \quad (16)$$

Since a change in the operational profile is equivalent to a change of ϕ_k , the reliability of any given operational profile with $\{\phi'_k\}_{k=1}^L$ can be estimated by

$$\hat{R}'_t = \sum_{k=1}^L \phi'_k \hat{R}_t(k)$$

without additional sampling or testing.

3.4 A numerical example

A simple numerical example is used to explain how our method works. Suppose a program contains 1000 blocks (i.e. $N = 1000$) with 50 faults injected. Assume also that after the testing 40 faults were detected with an average of 5 tests and the remaining 10 faults were not detected, each after 60 tests. To use Equation (8), we have $m_f = 50$, $r = 40$, $\sum_{i=1}^r \nu_i = 40 \times 5 = 200$, and $\sum_{j=r+1}^{m_f} \nu_j^+ = 60 \times 10 = 600$. Thus

$$\hat{p} = \frac{40}{200 + 600} = 0.05.$$

In other words, the probability of detecting a fault each time the block is executed is 0.05. Suppose when the same testing strategy, as that in estimating \hat{p} , was used to test the program, 20 faults were detected after running 75 test cases. Suppose α_0 , the prior probability that a block containing an error, is the same for all the blocks and it is unknown. Now further suppose that the probability of each block being executed is the same (i.e. all blocks are equally accessible to all the inputs) and on average each test case executes 600 blocks. Then each block is tested $n^+ = 75 \times 600/1000 = 45$ times. By Equation (10),

$$\begin{aligned} \hat{\alpha}_0 &= \frac{20}{1000(1 - (1 - 0.05)^{45})} \\ &= 0.0222. \end{aligned}$$

This implies that there were likely $0.0222 \times 1000 = 22.2$ faulty blocks initially. Since each block is tested 45 times, $\hat{\alpha}_t(i)$ in Equation (4) becomes

$$\begin{aligned} \hat{\alpha}_t(i) &= \frac{0.0222 \times (1 - 0.05)^{45}}{1 - 0.0222 + 0.0222(1 - 0.05)^{45}} \\ &= 0.00225. \end{aligned}$$

This means there would likely be $1000 \times 0.002253 = 2.25$ faults left in the program. The number 2.25 is consistent

with the initial 22.2 faults minus 20 detected ones. Since each test executes approximately 600 blocks, from Equation (13) we have

$$\begin{aligned}\hat{\pi}_t(x) &= (1 - 0.00225 \times 0.05)^{600} \\ &= 0.9346.\end{aligned}$$

Because we assume all blocks have the same α_0 and are tested equally, $\hat{\pi}_t(x)$ is independent of x . Therefore, $\hat{R}_t = \hat{\pi}_t(x) = 0.9346$. This low reliability can be due to the low testability ($\hat{p} = 0.05$) or a low test effort (only 75 test cases). If the test effort in terms of the number of tests executed is doubled while all the other parameters are kept the same, the reliability is increased to 0.994.

Note that testability and test effort are not the only factors which affect the reliability estimation. The relation between how a program is tested and how it will be used is also important. Table 1 shows how reliability estimation varies at different scenarios with the same assumption as before, that is, 1000 blocks, 75 test cases, 20 faults detected, and approximately 600 blocks being executed by each test run. Assume also that each run in the operation profile executes about 600 blocks. The first three scenarios (the first three rows, not including the header, of Table 1) suggest that reliability is strongly affected by testability. As testability increases, reliability increases provided other factors remain unchanged.² The last four scenarios (the bottom four rows of Table 1) show the importance of relation between testing and usage. Although the total number of blocks executed (the block coverage) is the same, in the third scenario every block is executed 45 times, whereas in the other three scenarios half of the blocks are executed 75 times and the other half are only executed 15 times. As a result, the reliabilities are different. When blocks are executed the same way in testing as they are used in the operational profile (the last row of Table 1), the software has a higher reliability. In short, it is clear that reliability cannot be estimated by using only the information of how much testing effort is spent and how many faults are detected. The relation between how a program is tested and how it is used must also be considered.

3.5 Extension to many types of faults

Different types of faults have different orders of difficulty of being detected [25]; therefore, we need to relax the assumption that a single testability p applies for all types of faults. Let there be K types of faults, classified according to their testability. For a given type j , the testability p_j can be estimated by Equation (8) with m_f , r , ν_i , and ν_i^+

²In general, after a program has passed a series of tests, with no failed test cases, high testability implies high reliability. However, in some unusual cases as discussed in [1], if faults are still not detected when testing stops, a high value of testability implies a more unreliable program.

being the testing results for the j th type of faults injected into the program. Let the estimated value be \hat{p}_j . To estimate the prior probability $\alpha_0(i)$ of having j th type of fault in block i , Equation (9) can be used by replacing p with p_j and letting s be the number of j th type of faults detected. Let the estimate be $\hat{\alpha}_0(i)_j$. Substituting p by \hat{p}_j and $\alpha_0(i)$ by $\hat{\alpha}_0(i)_j$ in Equation (4), we obtain the probability that the i th block has j th type of fault after testing. Let it be denoted as $\hat{\alpha}_t(i)_j$. By the competent programmer assumption (i.e. the chance of having two or more faults in one block is negligibly small), we have

$$\hat{\pi}_t(x) = \prod_{i \in S(x)} [1 - \sum_{j=1}^K \hat{\alpha}_t(i)_j \hat{p}_j] \quad (17)$$

Equations (14) and (16) for \hat{R}_t are the same. For a highly reliable program, all the values $\hat{\alpha}_t(i)_j \hat{p}_j$ are small, and in this case Equation (17) can be approximated by

$$\hat{\pi}_t(x) = 1 - \sum_{j=1}^K [1 - \prod_{i \in S(x)} (1 - \hat{\alpha}_t(i)_j \hat{p}_j)] \quad (18)$$

Or, equivalently, the overall failure rate is the sum of the individual failure rates of each type of fault.

To continue our previous example, suppose there are two types of faults and our testing strategy gives $\hat{p}_1 = 0.15$ and $\hat{p}_2 = 0.01$. Suppose that after 75 test cases are used to test the program, 20 Type 1 faults and three Type 2 faults are detected. Again, we assume the probability of each block being executed is the same in both the testing phase and in the operational profile. Then the failure rate due to a Type 1 fault is 0.0012 as computed in Table 1. Following the same steps of computation, the failure rate due to type 2 faults is 0.031. Thus, the total failure rate is $0.0012 + 0.031 = 0.0322$ and the corresponding reliability is 0.9678. This result agrees with the intuition that the hard-to-find faults are more responsible for the future failure than the easy-to-find ones.

4 A Case Study

We next present a case study to show the merits of using our method.

4.1 Description of the experiments

An application developed for the European Space Agency was used. This application, written in C, has 135 functions with 6107 executable lines of code in 2970 blocks. Hereafter, we refer to this application as `Space`. In this case study, instead of using blocks which are numerous, we chose each function as a basic reliability unit. This reflects

Table 1. Reliability estimation at different scenarios as discussed in Section 3.4

Testability p	How blocks are tested	How blocks are executed in the operational profile	Reliability R_t	Failure rate $1-R_t$
0.01	all $n_i^+ = 45^\dagger$	uniform [§]	0.8770	0.1230
0.05	all $n_i^+ = 45$	uniform	0.9345	0.0655
0.15	all $n_i^+ = 45$	uniform	0.9987	0.0013
0.15	half $n_i^+ = 75^\ddagger$ half $n_i^+ = 15$	uniform	0.9197	0.0803
0.15	half $n_i^+ = 75$ half $n_i^+ = 15$	Only the least tested blocks are executed	0.8697	0.1303
0.15	half $n_i^+ = 75$ half $n_i^+ = 15$	same as how they are tested	0.9728	0.0272

[†] Every tested block is executed 45 times.

[‡] Half of the tested blocks are executed 75 times and the other half are only executed 15 times.

[§] The notation “uniform” means the probability of each block being executed is the same.

the flexibility of our method in that it can be applied at different granularity levels.

To obtain an operational profile for *Space* we identified the possible functions of the program and generated a graph capturing the connectivity of these functions. And to estimate their occurrence probability, each arc in the graph was assigned a transition probability determined by interviewing the program users.

Based on this profile, 90 distinct test sets, each containing 200 distinct test cases, were randomly generated. In generating these random test sets, we used different initial seeds. This was done to ensure that test cases in every random test set were drawn from a different independent random sequence. Three testing strategies were then used to remove *redundant* tests from these sets based on the block, decision, and all-uses coverage, respectively. In each of the first thirty test sets, a screening was conducted by discarding a test case if it could not cover at least one block that was not already covered. This guaranteed that each consequent test case was not redundant in terms of block coverage. However, we did not re-examine whether the previous test cases were still necessary (i.e. once a test case was included in a test set, it was not excluded due to the inclusion of any new test cases). Test sets so generated are referred to as *block tests* and as TB_i , $1 \leq i \leq 30$. Similarly, test cases in the next thirty test sets and the last thirty test sets were filtered by using decision and all-uses coverage, respectively. We refer to these test sets as *decision tests* and *all-uses tests* and call them TD_i and TA_i , $1 \leq i \leq 30$, respectively. The trace of all blocks, functions, and files executed during the testing was recorded with a testing coverage measurement tool called ATAC [12].

A set of 16 faults for *Space*, obtained from the error-log maintained during the real testing and integration phase of this program, was used in this study. For convenience, each fault has been numbered as \mathbf{Fk} where the integer \mathbf{k} ($1 \leq \mathbf{k} \leq 16$) denotes the fault number. This number does

not indicate the order in which faults were detected during experimentation. A description of these faults can be found in [29]. These 16 faults were randomly divided into two groups. One group had ten faults and the other group contained the remaining six. Such division was performed 90 times with a different initial seed used each time. The reason is the same as explained in the previous paragraph to ensure that faults were divided by a different independent random sequence. The resulting groups are referred to as FA_i (with ten faults) and FB_i (with six faults), $1 \leq i \leq 90$, respectively. Another way to categorize these 16 faults is by their difficulty of being detected. Those easy faults with \hat{p} values (the estimated testability using Equation (8)) greater than, or equal to, 0.25 are classified as Type 1 faults. And those difficult faults with \hat{p} values less than 0.25 are classified as Type 2 faults.

The experiments conducted as a part of this study have the following pattern:

- **Step 1:** Using Equation (8) to estimate the testability \hat{p} . One group of faults in FA_i , $1 \leq i \leq 90$ were injected, one at a time, into the *Space* program as mutants that were then tested against a test set. The first thirty groups were tested against block tests, the next thirty against decision tests, and the last thirty against all-uses tests. For example, faults in FA_{31} were tested against TD_1 and faults in FA_{65} were tested against TA_5 . Each pair of a fault group and a test set is regarded as a separate experiment. All together there were 90 experiments. Table 2 lists the testabilities estimated for faults in FA_1 using all the 48 tests in test set TB_1 . The average estimated testability over the 90 experiments for faults in Type 1 and Type 2 is 0.440 and 0.044, respectively.
- **Step 2:** Using Equations (4) and (9) to estimate the probability for the i th function containing the j th type of fault $\hat{\alpha}_t(i)_j$. Similar to Step 1 except that faults in

Table 2. Estimated testability using test set TB_1 (with 48 test cases) on faults in FA_1

Fault	Type	Number of times that the fault was executed	Number of times that the fault was detected	Testability \hat{p}_j
F2	1	3	2	0.67
F3	1	7	4	0.57
F5	2	36	5	0.14
F6	2	29	5	0.17
F8	2	25	3	0.12
F9	1	5	3	0.60
F12	1	9	6	0.67
F13	1	20	14	0.70
F15	2	48	1	0.02
F16	2	5	0	0.00

$FB_i, 1 \leq i \leq 90$ were used. For the purpose of illustration, samples of the data set collected using TB_1 and FB_1 are shown in Table 3. For example, function `angclaus` has a probability 0.0001 to contain a Type 1 fault and a probability 0.0242 to contain a Type 2 fault.

- **Step 3:** Using Equations (13) and (14) to estimate the reliability \hat{R}_t . A random sample with 200 distinct test cases according to the operational profile as explained in the beginning of this section (Second paragraph of Section 4.1) was used.

Table 3. Estimated probability obtained using test set TB_1 and faults in FB_1 for functions containing a Type 1 or a Type 2 fault. Due to a space limitation, only ten of the 135 functions in $Space$ were listed.

Function	$\hat{\alpha}_t(i)_1$	$\hat{\alpha}_t(i)_2$
<code>addscan</code>	0.0000	0.0022
<code>adremdef</code>	0.0000	0.0013
<code>ampunit</code>	0.0000	0.0012
<code>ampval</code>	0.0000	0.0002
<code>angclaus</code>	0.0001	0.0242
<code>angledir</code>	0.0004	0.0442
<code>anglerot</code>	0.0001	0.0141
<code>angstep</code>	0.0001	0.0101
<code>angunit</code>	0.0000	0.0021
<code>angval</code>	0.0000	0.0011

4.2 Results and Analysis

In Equation (6), we assume faults are independent regardless of the existence of other faults.³ As a result, the

³This statement cannot be considered true in general cases. However, if one accepts the competent programmer assumption this becomes a minor problem, since programs under this assumption have usually only a few sparse faults. This restriction needs to be further investigated and additional effort is needed to relax it.

chance of these faults interfering with each other is small. Table 8 shows q_1 , the revealability of Fault **F1**, when it is injected in $Space$ together with a group of five others randomly selected from the remaining 15 faults. Each experiment in this table uses a different set of five faults selected from a different independent random sequence. The number of times that **F1** was revealed is in column 2, the number of times the function which contains **F1** was tested is in column 3, and the corresponding revealability is in column 4. From these data, it is clear these revealabilities are very similar. A statistical test (chi-square test) shows the revealabilities are indeed very similar with a p-value equal to 0.780. The same experiments were repeated for the other 15 faults. None of them shows any significant interaction.

Table 4. Revealability of **F1** when it was injected each time with other five randomly chosen faults

Experiment	Number of times revealed	Number of times tested	Revealability
1	5	8	0.625
2	3	6	0.500
3	6	9	0.667
4	5	8	0.625
5	6	13	0.461
6	6	6	1.000
7	8	8	1.000
8	7	8	0.875
9	8	12	0.750
10	5	7	0.714
11	5	8	0.625
12	3	6	0.500
13	6	11	0.545
14	3	4	0.750
15	6	9	0.667
16	6	8	0.750
17	6	10	0.600
18	6	9	0.667
19	3	5	0.600
20	8	11	0.727
21	7	10	0.700
	Sum: 119	Sum: 176	Average: 0.676

The average testabilities, \hat{p} using Equation (8), of the 16 faults with respect to all 30 block tests, decision tests and all-uses tests are in Table 5. The revealabilities, \hat{q} also using Equation (8), of these faults based on inputs selected randomly from the operation profile are in the 5th column of Table 5 as well. Results from the two sample proportional test which examines whether \hat{p} and \hat{q} are statistically different for each fault are in the last column. If 0.05 level of significance is used, our data in Table 5 indicate that the revealability and the testability are not significantly different except for **F5**, **F6**, **F8**, **F9**, and **F10**. When all the 16 faults are considered together, as shown in the last row of Table 5, \hat{q} (0.187) is less than \hat{p} with respect to block tests (0.219), decision tests (0.218) and all-uses tests (0.213), but the difference is small. Hence, by assuming $q_i = \hat{p}$ as in Equation (13), we can obtain a conservative estimate of the

reliability.

Table 5. Comparison of testability (\hat{p}) and revealability (\hat{q})

Fault	Average \hat{p} with respect to			\hat{q}	p-value
	block tests	decision tests	all-uses tests		
F1	0.684	0.640	0.670	0.676	0.930
F2	0.830	0.793	0.843	0.764	0.626
F3	0.522	0.455	0.367	0.470	0.101
F4	1.000	1.000	1.000	1.000	1.000
F5	0.105	0.112	0.086	0.050	0.001
F6	0.111	0.097	0.089	0.071	0.051
F7	1.000	1.000	1.000	1.000	1.000
F8	0.048	0.080	0.066	0.044	0.019
F9	0.729	0.746	0.776	0.873	0.039
F10	0.754	0.723	0.742	0.901	0.002
F11	0.192	0.160	0.222	0.172	0.701
F12	0.582	0.581	0.597	0.688	0.067
F13	0.669	0.654	0.698	0.688	0.314
F14	0.076	0.124	0.114	0.073	0.308
F15	0.022	0.027	0.021	0.016	0.058
F16	0.015	0.027	0.040	0.041	0.774
Average	0.219	0.218	0.213	0.187	0.001

We next compare the reliability estimated by using our method with that from Musa’s Basic execution model and the logarithmic Poisson (LP) model [20]. The reason for chosen these two models are given in Appendix B. Since we know the real reliability R after debugging, we first compare $\Delta = |\hat{R} - R|$, where \hat{R} is an estimate by whatever the methods we are interested. For the 30 block, decision, and all-uses cases tested and debugged, the times that (14) has a smaller Δ than the other methods are given in Table 7. When all the 90 test sets are considered together, the winning ratio of our method is 88:2 over Musa’s basic model and 59:31 over Musa’s LP model. They are both statistically significant at the 0.01 level (p-value < 0.01). From these observations, it is clear that our method is better.

To check the detailed performance of $\Delta = |\hat{R} - R|$, the log failure ratio

$$\lambda = \log_{10}\left(\frac{1 - \hat{R}}{1 - R}\right), \quad (19)$$

is used. The reason is that $\Delta = |\hat{R} - R|$ is not a fair formula to summarize different reliability discrepancies. For example, $R = 0.90$ being estimated as 0.89 with a difference 0.01 looks much worse than $R = 0.999$ being estimated as 0.995 with a difference 0.004. However, this is not the case from the failure rate point of view. In the former the failure rate equals $1.00 - 0.90 = 0.10$ being estimated as 0.11 which is 10% overestimated; whereas in the latter the failure rate equals 0.001 being estimated as 0.005 which is four times (i.e. 400%) overestimated. Relative error $|\hat{R} - R|/R$ and relative failure rate discrepancy $|\hat{R} - R|/(1 - R)$ also have a similar problem. The value 10^{-4} is used when $R = 1$ to avoid a 0 denominator.

When Equation (19) is used, the ideal case is \hat{R} equals R which gives a zero value to λ . Otherwise, the larger the $|\lambda|$, the worse the estimate of R . If \hat{R} is greater than R (i.e. overestimating the reliability or underestimating the failure rate), then λ is less than zero. Similarly, if \hat{R} is less than R (i.e. underestimating the reliability or overestimating the failure rate), then λ is greater than zero. The log failure ratios of the estimated reliabilities with respect to the *true* reliabilities using our method, Musa’s basic model and Musa’s LP model were computed by using Equation (19). The mean values and the standard deviations for λ are given in Table 4.2. These data suggest that all three methods are biased toward overestimating the failure rates (i.e. giving a conservative estimated reliability). One explanation is that if a program has only a few faults, after good testing and debugging, it is possible that all the faults have been removed. This makes the reliability equal 1, but this fact is difficult to confirm by limited testing. From Table 4.2, we also observe that the mean value of our method in each case is smaller than that of Musa’s methods, which implies our method provides a more accurate estimated reliability.

Table 6. Number of wins based on using $|\hat{R} - R|$ of our method over two Musa’s models

	Musa’s basic model [†]	Musa’s LP Model [‡]
block tests	28/30	17/30
decision tests	30/30	20/30
all-uses tests	30/30	22/30

[†] Our method is better at the 0.01 level of significance.

[‡] Our method is better at the 0.05 level of significance except for the entry “17/30”.

Table 7. Mean values and standard deviations of log failure ratios in Equation 19

with respect to all 30 block tests			
	Our method	Musa’s basic model	Musa’s LP Model
mean value	0.552	1.500	0.678
std. dev.	1.111	1.041	1.136

with respect to all 30 decision tests			
	Our method	Musa’s basic model	Musa’s LP Model
mean value	0.406	1.435	0.675
std. dev.	0.914	0.908	0.967

with respect to all 30 all-uses tests			
	Our method	Musa’s basic model	Musa’s LP Model
mean value	0.399	1.481	0.614
std. dev.	1.099	1.016	1.208

5 Concluding Remarks

This paper presents a method for estimating software reliability based on the evaluation of program testability. The method is more refined than conventional software reliability growth models because it takes into account characteristics of the modelled process that are not considered by the latter. These characteristics are the program testability and the structural or functional coverage obtained during testing. Once testability has been estimated, the application of our method does not require more effort than do ordinary reliability growth models. We report an application of our method to a real software system. Within the limits of this specific application, our method has shown better predictive ability when compared with conventional reliability growth models.

We consider this work a step in the direction of using testability to estimate software reliability. Some problems are still unsolved and need further research, e.g., to extend the current point estimation to interval estimation. Certain assumptions, such as the independence of faults, need to be relaxed in order to make the method generally acceptable. Our future direction includes these aspects, as well as a consideration of an imperfect oracle and faults that can only be detected when some combinations of blocks are executed either in the same execution or different executions.

References

- [1] A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *IEEE Trans. on Software Engineering*, 22(2):97-108, February 1996.
- [2] T. A. Budd, "Mutation Analysis of Program Test Data," PhD thesis, Yale University, New Haven, CT, May 1980.
- [3] . H. Chen, M. R. Lyu, and W. E. Wong, "An empirical study of the correlation between code coverage and reliability estimation," in *Proceedings of the Third IEEE International Software Metrics Symposium*, pp 133-141, Berlin, Germany, March 1996.
- [4] . H. Chen, M. R. Lyu, and W. E. Wong, "Incorporating code coverage in the reliability estimation for fault-tolerant software," in *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, pp 45-52, Durham, NC, October 1997.
- [5] R. C. Cheung, "A user-oriented reliability model," *IEEE Trans. on Software Engineering*, SE-6(2):118-125, March 1980.
- [6] S. R. Dalal, J. R. Horgan, and J. R. Kettenring, "Reliable software and communication: Software quality, reliability and safety," in *Proceedings of the 15th IEEE International Conference on Software Engineering*, pp 425-435, Baltimore, MD, May 1993.
- [7] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pp 158-171, San Diego, CA, January 1996.
- [8] R. A. DeMillo and A.P. Mathur, "On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software," in *Thirteenth Minnowbrook Workshop on Software Engineering*, July 1990.
- [9] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. on Software Engineering*, SE-10(7):438-444, July 1984.
- [10] F. Del Frate, P. Garg, A Mathur, and A. Pasquini, "On the correlation between code coverage and software reliability," in *Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, pp 124-132, Toulouse, France, October 1995.
- [11] R. G. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. on Software Engineering*, 16(12):1402-1411, December 1990.
- [12] J. R. Horgan and S. A. London, "ATAC: A data flow coverage testing tool for C," in *Proceedings of Symposium on Assessment of Quality Software Development Tools*, pp 2-10, New Orleans, LA, May 1992.
- [13] W. E. Howden and Y. Huang, "Software trustability," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pp 143-151, Monterey, CA, November 1994.
- [14] A. Koenig, "C Traps and Pitfalls," Addison-Wesley, New York, 1988.
- [15] N. Li and Y. K. Malaiya, "On input profile selection for software testing," in *Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering*, pp 196-205, Monterey, CA, November 1994.
- [16] B. Littlewood, "Software reliability model for modular program structure," *IEEE Trans. on Reliability*, 28(3):241-246, 1979.
- [17] Y. K. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe, "The relation between software test coverage and reliability," in *Proceedings of the Fifth*

IEEE International Symposium on Software Reliability Engineering, pp 186-195, Monterey, CA, November 1994.

- [18] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation based test adequacy criteria," *The Journal of Software Testing, Verification, and Reliability*, 4(1):9-31, March 1994.
- [19] K. W. Miller, L. J. Morrel, R. E. Noonan, S. K. Park, D. M. Nicol, B. M. Murril, and J. M. Voas, "Estimating the probability of failure when testing reveals no failure," *IEEE Trans. on Software Engineering*, 18(1):33-43, January 1992.
- [20] J. D. Musa, A. Iannino, and K. Okumoto, "Software Reliability Measurement Prediction Application," McGraw-Hill, New York, 1987.
- [21] A. J. Offutt and J. H. Hayese, "A semantic model of program faults," in *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pp 195-200, San Diego, CA, January 1996.
- [22] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Software-Practice and Experience*, 26(2):165-176, February 1996.
- [23] P. Piwowarski, M. Ohba, and J. Caruso, "Coverage measurement experience during function test," in *Proceedings of the 15th International Conference on Software Engineering*, pp 287-301, Baltimore, MD, May 1993.
- [24] K. Siegrist, "Reliability of system with Markov transfer of control," *IEEE Trans. on Software Engineering*, 14(7):1049-1053, July 1988.
- [25] M. Trachtenberg, "Order and difficulty of debugging," *IEEE Trans. on Software Engineering*, 9(6):746-747, November 1983.
- [26] G. S. Varadan, "Trends in reliability and test strategies," *IEEE Software*, 12(3):10, May 1995.
- [27] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. on Software Engineering*, 18(8):717-727, August 1992.
- [28] J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE Software*, pp 17-28, May 1995.
- [29] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault

detection effectiveness: A case study in a space application," in *Proceedings of the 21st Annual International Computer Software and Application Conference*, pp 522-528, Washington, D.C., August 1997.

- [30] A. Wood, "Software reliability growth models: Assumptions vs. reality," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pp 136-141, Albuquerque, New Mexico, November 1997.

Appendix A: Derivation of Equations (4), (8) and (9)

Equation (4): Let SPT= a statement passed n_i tests, SC= the statement is correct, and SF= the statement is faulty. Then

$$\begin{aligned}\alpha_t(i) &= P(SF|SPT) \\ &= \frac{P(SF \cap SPT)}{P(SPT)} \\ &= \frac{P(SPT|SF)P(SF)}{P(SPT|SF)P(SF) + P(SPT|SC)P(SC)}\end{aligned}$$

Since $P(SF) = \alpha_0(i)$, $P(SC) = 1 - \alpha_0(i)$, $P(SPT|SF) = (1 - p)^{n_i}$, and $P(SPT|SC) = 1$, we have Equation (4).

Equation (8): The likelihood function is

$$\ell(\alpha_0) = \prod_{i=1}^r (1-p)^{v_i-1} p \prod_{i=s+1}^{m_j} (1-p)^{v_i^+}$$

Taking log on the right side and differentiating with respect to p and setting the derivative to 0, the solution is Equation (8).

Equation (9): the likelihood equation of detecting s blocks with faults at tests n_1, n_2, \dots, n_s , respectively, is

$$\ell(\alpha_0) = \prod_{j=1}^s \alpha_0(1-p)^{n_j-1} p \prod_{i=1}^{N-s} [(1-\alpha_0) + \alpha_0(1-p)^{n_i^+}]$$

Taking log on the right side and differentiating with respect to α , setting the derivative to 0, and assuming $\alpha_0(i) = \alpha_0$, we have Equation (9)

Appendix B: Choice of Models for Comparison

It is widely accepted that no model can be considered appropriate in all applications [1]. Best models for having future predictions from specific data must be selected by analysing the accuracy of past predictions on the same data. We selected for comparison the model offering the best fit with the available interfailure data. The model accuracy was evaluated using CASRE [4] that provides several analysis techniques. We selected as indicators: prequential likelihood, bias, noise and trend. Use of these techniques has been proposed in [3] and they are described in detail in [2]. Six models were applied to the 90 data sets, but the limited number of failures of each data set and the characteristics of the interfailure data reduced the number of applicable models to the following four: Geometric [5], Jelinsky Moranda [6], Musa basic [7, 8] and Musa Okumoto [7, 8]. The assumptions of these models are described in detail in [2]. Table 1 shows the average of the indicators, over most of the 90 data sets (not all the models were applicable to all the data sets). The table shows also the goodness-to-fit of the models using the Kolmogorov distance. The Kolmogorov distance is a measure of the discrepancy between the sample cumulative distribution function and the model. It is used to test the hypotheses H_0 : the model has no lack of fit to the data against the alternative H_1 : the model has lack of fit to the data. Only the two Musa's models can claim no lack of fit at 0.05 significance level. They are used for comparisons with the our method.

[4] M. R. Lyu, and A. P. Nikora, "CASRE - A Computer Aided Software Reliability Estimation Tool", Proc. of CASE '92, Montreal, Canada, July 1992, pp. 264-275.
 [5] P. B. Moranda, "Predictions of Software Reliability during Debugging", Proc. of the Annual Reliability and Maintainability Symposium, Washington D. C., 1975.
 [6] P. B. Moranda, "Event-Altered Rate Models for General Reliability Analysis", IEEE Transactions on Reliability, Vol. R-28, no. 5, pp.376-381, 1979..
 [7] J. Musa, "A Theory of Software Reliability and its Applications", IEEE Transactions on Software Engineering, Vol. SE-1, no. 3, pp. 312-327, 1975.
 [8] J. D. Musa, A. Iannino and K. Okumoto, "Software Reliability: Measurement, Prediction, Application", McGraw-Hill, 1987.

Table 8. Prediction Accuracy of Models

Model	Prequential	Bias	Noise	Trend	Goodness to fit test	
	Likelihood				Kolm. Dist.	Sign. at 0.05
Geometric	9.584e+0	5.654e-1	1.826e-1	5.418e-0	6.045e-1	yes
Jelinsky Moranda	9.604e+0	5.332e-1	2.274e-1	5.510e-0	5.930e-1	yes
Musa basic	1.050e+1	6.607e-1	4.307e-1	5.58006e-0	3.317e-1	no
Musa LP	1.047e+1	5.553e-1	3.122e-1	5.555e-0	4.213e-1	no

The reference need to take care of.

[1] A. A. Abdel-Ghaly, P. Y. Chan and B. Littlewood, "Evaluation of competing software reliability prediction", IEEE Transactions on Software Engineering, Vol. SE-12, no. 9, pp. 950-967, September 1986.
 [2] W. H. Farr, O. D. Smith, "Statistical Modeling and Estimation of Reliability Functions for Software - User's Guide", NSWC DD TR 84-373, 1993.
 [3] B. Littlewood, A. A. Abdel-Ghaly, and P. Y. Chan, "Tools for the Analysis of the Accuracy of Software Reliability Predictions", Software Systems Design Methods, ed. J. K. Skwirzynski, NATO ASI Series, Vol. F22, Springer and Verlag, 1989, pp.299-333.