PERILS OF SOFTWARE RELIABILITY MODELING*

J. Robert Horgan, Aditya P. Mathur, Alberto Pasquini, and Vernon J. Rego

February 16, 1995

Abstract

Using arguments based on an analysis of the state of the art in the field of software reliability estimation and related practice, we warn practitioners that the road to reliability estimation is fraught with peril. Our analysis is based on several factors, with key factors including (a) difficulties in estimating accurate operational profiles, and (b) inaccuracies in reliability estimates caused by erroneous operational profiles and/or invalid model assumptions. To circumvent these perils, we advocate new approaches to software reliability estimation, particularly robust approaches with minimal model assumptions, and approaches which combine failure data with code coverage data.

1 Introduction

Existing methods for estimating software reliability are fraught with risk. We present a critique of such methods and encourage new approaches which seek to reduce or eliminate factors that imperil the development and use of reliable software. Indeed, many researchers have recognized problems associated with software reliability estimation. For example, Hamlet [18] gives a detailed critique of sampling models and the use of operational profiles in the software reliability estimation process. Here, we go a step further in outlining additional problems in the estimation process and in the use of operational profiles, as well as problems in the use of reliability growth models. We argue that the standard approaches to reliability estimation of software are intrinsically flawed. If used at all, they must be used with knowledge and extreme caution. We base our arguments for this viewpoint on (a) the difficulties in obtaining accurate operational profiles, and (b) problems inherent to reliability estimation based on simple failure data.

The *reliability* of a piece of software is generally accepted to be the probability of failure-free operation of the software over a given time interval known as the *exposure period*. Many methods for estimating such a probability have been proposed. Several accepted methods

*This research was supported in part by an award from the Center for Advanced Studies, IBM Toronto Laboratories, NSF award CCR-9102331, and NATO award 5-2-05/RG No. 900108. J. Robert Horgan is with Bellcore, Aditya P. Mathur and Vernon J. Rego are with Purdue University, and Alberto Pasquini is with ENEA. All correspondence regarding this paper may be sent to Aditya P. Mathur, 1398 Department of Computer Sciences, Purdue University, W. Lafayette, IN 47907, USA.

currently in use emphasize the manner in which a piece of software is used, aptly termed the *operational profile*, in estimating the software's reliability. Given a software system that has been integrated using constituent components, the software is *tested* in accordance with the system's operational profile. During such tests, data which result in system-failure or *failure-data* are obtained, and subsequently fed to one or more reliability estimation models [16]. As a result, one obtains a set of estimates of the probability of failure-free operation of the system for given exposure periods. Typically, a manager bases his decision on whether the software is reliable enough for release based on a threshold value of the estimated probability. If the estimate(s) of reliability falls below the threshold, the software is considered unsuitable for deployment without further testing and debugging. If the estimate(s) fall above the threshold, the software is considered to have satisfied at least one condition for deployment.

Using simple arguments, and supported both by our experiences and empirical data, we reveal flaws in current-day processes for software reliability estimation. With the same tools, we motivate the need for new ways of obtaining reliable software, proposing two new approaches for the estimation of reliability. These approaches are indicative of a general trend to move away from simple failure-data related predictions, also known as *black-box* testing, to predictions that seek to utilize relevant information present within the software system itself. Because of the use of such additional information, such testing has come to be known as *white-box* testing. One approach aims at improving the accuracy of reliability estimates through the use of various code *coverage* measures. Another approach aims at integrating software reliability estimation with the software development process. Both approaches are independent but complement one another. While the former approach targets the improvement of estimates, albeit estimates obtained from standard models, the latter approach extends the scope of estimation by taking it to the level of individual software components, opening up the possibility of estimate re-use etc. We advocate such methodologies which exhibit the potential to advance the state of practice by moving software reliability estimation tasks into the development process.

2 Difficulties in estimating Operational Profiles

The space of acceptable inputs for most software systems is extremely large and, for all practical purposes, may be considered infinite. Typical use of the system will generally entail inputs distributed in nontrivial ways over the input space. Determining an operational profile based on system use or typical inputs is a formidable task. Nevertheless, obtaining an operational profile is key to the current practice of reliability estimation with most accepted models. Indeed, because of this, a significant amount of work has gone into establishing procedures for operational profile estimation. Musa [28] provides a detailed methodology on the building of operational profiles. Clearly, in such an estimation process customer input is critical, because the customer is supposed to know how the software purchased will be used. Thus, the first step in this methodology is to identify a customer profile. This profile is refined in a sequence of steps to obtain an operational profile which, in turn, depends on how users are expected to use the software.

Once an operational profile has been built, the next step is to build a set of sample test cases from the input domain, reflecting typical use of the system. The given software system is then exercised using these test cases, consequently yielding failure data. This failure data is input to one or more reliability models to estimate the reliability of the system.

As indicated earlier, an operational profile is an estimate of the relative frequency with which various features of the software will be used. This frequency is user dependent and is necessarily derived through a usage analysis [28]. In the following section we identify situations in which it may not be possible for one to obtain an operational profile with any degree of accuracy. In such cases, one has little choice but to rely on educated guesstimates. Based on several informal discussions with practicing engineers over recent years, we contend that the problems cited below are encountered often during testing and reliability estimation. We are not aware of existing documentation of such problems. We present several realistic scenarios to emphasize the nature of the problems; for obvious reasons, we do not give concrete example of such software systems, or environments where such problems have arisen.

2.1 New software

When a new system is designed, a customer base may or may not exist. If a customer base exists, it is unlikely that the developer has access to information concerning how the new system is to be used. In any case, building an operational profile of system usage is close to impossible for new software. As an example, consider the development of a system meant to control an instrument for experimentation aboard a spacecraft. Suppose that the experiment is unique and is being performed for the very first time. Features in this system will correspond to requirements derived from an analysis of the instrument and the nature of its expected use. Therefore, one is likely to have a list of features without a customer base. Consequently, there is no recourse but to rely on guesstimates of the occurrence probabilities of individual features.

Musa reports successful experiences in developing trustworthy operational profiles [28]. Indeed, most of the experiences reported involve applications where operative usage of the software is predictable because the usage is related to identifiable events pertaining to human activity. For example, in software for a telephone switching system one can estimate the number of calls per day on a given day and use this information to decide on the probability of invoking the call-processing modules in the switching system in relation to the probability of invoking other modules. In other applications, such as the control of a chemical or nuclear process, distinct software components are activated by a complex pattern of events triggered by characteristics of the process being controlled and the frequency of such events can hardly be estimated apriori. In some cases very critical software components are activated by events whose frequency of occurrence during operative usage of the software is completely unknown. As a typical example, consider the software components which perform failure detection and containment functions in the flight control system of a spacecraft. Some of such components are activated by hardware malfunction – such as the erasure or corruption of the RAMs content due to single event upset and latch-up, i.e., due to flux of electrons, protons and heavier ions (cosmic rays) [4].

2.2 New features

The continued development of an existing software system may give rise to new versions of the system. Typically, one or more new features are added to the system if there is a perceived need for these features, and the cost of adding the features does not offset their benefits. Despite the existence of a user base familiar with a current or older versions of the system, it is hardly likely that the developer has access to prior information on how the new version, and in particular the new features, will be used. Once again, the developer has to rely on guesstimates of the occurrence probabilities of the new features. This situation is further aggravated by the fact that the addition of new features may cause a significant change in how the system will be used. A new feature f_n that is well accepted to the extent of becoming popular, its use preferred to that of an older feature f_e will undoubtedly alter the frequency with which feature f_e is used. While we may anticipate such a change, measuring its effects is a nontrivial matter.

2.3 Feature definition

Generally speaking, a feature is not a well defined entity. Given a system which provides two features f_1 and f_2, it is questionable whether the use of these features in different sequences also defines a feature. Because of the large variety of possible feature permutations, the inclusion of sequences of features into the operational profile will result in huge profiles that are both difficult to build and difficult to manage. As another example, suppose that a sort module with two features f_1 and f_2 is embedded in a large system. Suppose also that the module is internal to the system and the user has no direct access to it. If the system provides a feature f that occasionally invokes the sort module, there is some question as to whether the operational profile must treat f as a feature in isolation, or treat the combinations f, f_1, and f, f_2 as two distinct feature composites.

2.4 Feature granularity

Consider a program that contains a total of N lines of executable code. If the operational profile consists of k distinct features, the program will have an average of $\frac{k}{N}$ lines per feature. In practice, the number of lines per feature is either more or less than this average. For example, for a system with 100,000 lines of code and 100 features in the operational profile, the likelihood of finding features that correspond to more than 1000 lines of code is high. To test the code well it would be desirable to specify features with finer granularity, resulting in fewer lines of code per feature. If feature definitions are based on direct use, it may be difficult to specify features with fine granularity.

2.5 Multiple and unknown user groups

An operational profile is essentially meant to be a summary of typical usage. It is assumed that users which generate such usage-frequency data belong to a relatively homogeneous class. A reliability estimate obtained with an operational profile built over such an assumption is at best valid for this restricted class of users. On the other hand, developers often

tend to want software reliability estimates that are independent of user-related assumptions. Consider, for example, the reliability needs of developers of operating systems, and the unmitigated variety of operating system users. It is not clear how an operational profile can be built to meet the needs of such developers.

At this stage of our discussion, we are led to believe that the estimation of operational profiles is both a difficult and an error prone task. The estimation of software reliability based on operational profiles is often projected as "*customer* or *user* centered" system testing. But based on the arguments given above, we believe that this estimation is at best a "*known* user centered" approach. If an *unknown* user is let loose on a software system, it is unlikely that his usage profile will match the operational profile of the system. Consequently, there is little reason to believe that the reliability estimates projected with the aid of the misleading profile will have any meaning to such a user.

3 Problems with reliability estimation

In principle, reliability estimation is supposed to yield accurate estimates in the presence of an accurate operational profile. In the following sections, we argue that for a number of reasons this is far from true. Factors that hinder unbiased estimation in the presence of accurate operational profiles range from the use of unacceptably weak test sets which favor fault-free sections of code, to problems with feature definitions, highly skewed profiles and the dampening effects of testing saturation. Below, we investigate these issues and outline some key problems.

3.1 Inadequate test set

When black-box testing is done, based on an operational profile, input test cases are constructed based on the features contained in the profile. Negative bias in the estimate of a probability may cause a feature's occurrence probability to be unacceptably low in the profile. Even worse, poor construction of the profile may result in the feature being eliminated altogether, i.e., causing it to exhibit an occurrence probability of zero. As a result, this feature undergoes a weak test or remains untested. A problem with conducting tests in this manner is the complete absence of a program-based notion of *test-set adequacy*. Thus, for example, one does not know if the test set has caused each statement in the program to be executed at least once. For a formal definition of test-adequacy see [25]. Consequently, after a program is tested and failure data is obtained, there is virtually no way of determining the "goodness" of the set of test cases used during testing. In the absence of any program based evaluation of test-set adequacy, black-box testing places an unjustifiable amount of faith in the the power of operational-profile guided statistical sampling of a program's input domain.

It should be obvious that an operational profile which departs from the "true" usage of a system, if such a true usage can be defined, can result in a *weak* test-set. Such a test-set may result in a feature not being tested at all, or perhaps a weak testing of the feature. If errors exist in the implementation of this feature, it should be clear that failure data

generated during testing will not contain failures indicative of these errors; this failure data may lead to overestimates of software reliability.

3.2 Coarse features

A feature may correspond to several lines of code. In black-box testing there is an attempt to generate test cases which exercise all aspects of a feature well. There is, however, no good measure of the "goodness" of such an exercise. Code related to this feature may never be exercised even though the feature occurs with high probability in the operational profile. This situation is likely to occur when test cases are sampled uniformly randomly from the input domain, or when test cases are generated manually without knowledge of the current extent to which the code corresponding to the feature in question has been exercised.

Empirical data obtained from two applications that have been tested extensively over several years indicates manual test-case generation, even in the presence of full knowledge of program features and functions used to implement these features, does not lead to high levels of code coverage [14]. We must then conclude that inadequate testing of a feature is likely to result in misleading failure data and poor reliability estimates.

3.3 Skewed operational profile

As indicated earlier, an operational profile of a software system is built by enumerating the possible input states and their probabilities of occurrence [29]. Each input state corresponds to a path of control flow through the software system under evaluation. To increase system reliability, the software is subjected to new test cases which are sampled in accordance with the occurrence probabilities of the input states. A different approach is to specify the sequence of modules to be executed instead of the input states [10]. Each sequence of modules is associated with a probability of occurrence. Yet another approach [33] considers module reliability and the structure of their interactions. The structure and relative frequency of these interactions is determined via inter-module transition probabilities. The overall system reliability is obtained from both the reliability of the modules and these transition probabilities. Sensitivity coefficients are then derived to indicate which modules are most critical in affecting system reliability. To increase the overall system reliability, new test cases are selected at random using the sensitivity coefficients (which indicate the modules that must undergo more careful testing). Despite methodological test-case sampling, these approaches do not take into account the structure of the software under evaluation and the code coverage derived through the execution of test cases. If an operational profile is *skewed*, placing great emphasis on certain test cases while de-emphasizing others, there is considerable scope for blatantly misdirected testing effort.

Consider a scenario where software is used to perform crucial functions in process control systems: often, a major portion of the software's operation is spent within simple and straightforward looping operations. Typical examples of such operations include control of process variables in chemical or nuclear process operations, or control of response variables in patient monitoring systems in medical environments. In these systems, complex and critical functions are usually activated only in case of transients or relatively infrequent and abnormal conditions. As a specific example, we can report an instance involving the control

system of a scientific instrument planned for future use in a joint Russian-European space mission to Mars [35]. In this application, a single module which represents only 3.01% of the code consumes as much as 99.99997% of the system's operating time. In addition, this module performs very simple operations (essentially checking for the presence of conditions requiring the activation of other specific modules) and has a simple and linear repetitive structure. The use of an operational profile in such a situation, either via occurrence probabilities of input states or via sensitivity coefficients, is bound to result in an *over-test* of the simplest part of the software system. Operational-profile based testing will cause testing of the module in question to continue even after complete code coverage of the module has been attained, suggesting misdirected testing effort geared towards code that has a low likelihood of yielding faults. Naturally, continued testing of this module will cause the reliability of the entire system to grow because of the absence of failure data. We must conclude then that such an effort leads to misleading reliability estimates.

3.4 Problems with other models

Models used in the prediction of software reliability include a variety of other methodologies besides the methodology of reliability growth. Below, we briefly examine some of these methods and indicate potential problems and pitfalls in their use.

A methodology proposed by Nelson [30] for software reliability estimation may be considered a pioneering methodology for models based on *statistical testing*. In essence, the methodology advocates that software be tested using test cases generated randomly, but with a distribution governed by the system's operational profile. System reliability is estimated by computing the ratio of frequency of successful executions to the total number of tests. Unfortunately, the number of test cases required to obtain significant confidence bounds on this estimate is generally too large to be acceptable in practice. Further, as in the case of reliability growth models, reliance on an estimate of the system's operational profile causes the approach to suffer from all the shortcomings associated with pure profile-based testing.

A refinement of Nelson's approach is proposed by Parnas [34]. He suggests that in safety critical applications, random test cases generated using the operational profile may be used to show that the failure probability falls below a specified upper bound. In similar vein, other researchers propose *binning* the input domain, and combining information on test results with information obtained from code analysis [26]. To the best of our knowledge, however, these models remain unused in software practice. Perhaps some of these approaches are best viewed as theoretical frameworks and stepping stones for novel research in the area, but not yet practically viable. Other approaches have been refined enough to be practically viable but suffer from lack of convincing validation. Hamlet provides [18] an analysis of demanding and unrealistic assumptions required by some of these models.

There is a class of models which base reliability estimation on the number of test cases executed and on an analysis of the program structure exercised by those test cases [39]. Some models also rely on test selection strategies that attempt to maximize reliability growth [1, 36]. A key problem with these models is the amount of effort required in their application. A validation effort described in [2] gives ample evidence of the unsuitability of some of

these models in practical settings. Even worse, these models rely on assumptions involving fault distribution and/or inference drawn from the correct execution of the software on one test case on its expecxted behavior on another test case. This assumption is similar to the nearest neighbor dependency assumption [1]. Again, these approaches are best viewed as frameworks for further research but not as usable models in current practice.

Because there is little consolidation of such models, or systematic use and validation, we have no access to reported experiences of their application in practice, or with tools that facilitate their use. The latter is particularly frustrating to practitioners because of the amount of effort typically required in their use. A detailed survey of these models and a description of possible improvements can be found in [3].

3.5 Reliability estimation towards the end of development

It is usually the case that estimation of system reliability is done late in the development process, upon completion of system integration and prior to release. At this stage, one or more reliability growth models are used in tandem with system tests. Unfortunately, attempts at reliability estimation this late in the process provides no information on the reliability of individual components during their development. Only after system integration is it typical for an engineer to get an estimate of its reliability. We feel that it is both useful and necessary for a software manager to have access to reliability estimates of the constituent components of a system. Armed with such knowledge, a manager will be able to make informed decisions on how construction- and architecture-related testing and reliability estimation resources may be allocated during the rest of the software development phase.

A possible reason for why reliability growth modeling may simply not be applicable to software components during the early stages of software development may be attributed to limitations on component size. It is argued [29] that estimates given by reliability growth models converge only in the presence of a sufficiently large number of software failures. Software components that are not large enough to yields a sufficient number of failures are not amenable to reliability estimation.

3.6 Saturation effect and reliability overestimation

We argue that during testing, all testing methods eventually tend to limits in their ability to reveal faults. Though these limits are software dependent, they follow a clear pattern which helps identify the strengths of different testing methods. The tendency of a given testing technique to reach a limit in its ability to reveal new faults is due to a *saturation effect* caused by the software on the technique. We hasten to point out that the saturation effect has serious ramifications for software reliability estimation regardless of operational profile use.

As shown in Fig. 1, different testing techniques tend to saturate at different points on the scale of fault revelation. Because of this the strengths of different techniques, as measured by their fault-revealing ability, become apparent. We now explain how such fault revelation limits may result in significant over- or under- estimates when a given testing method is used in conjunction with existing reliability models [12].

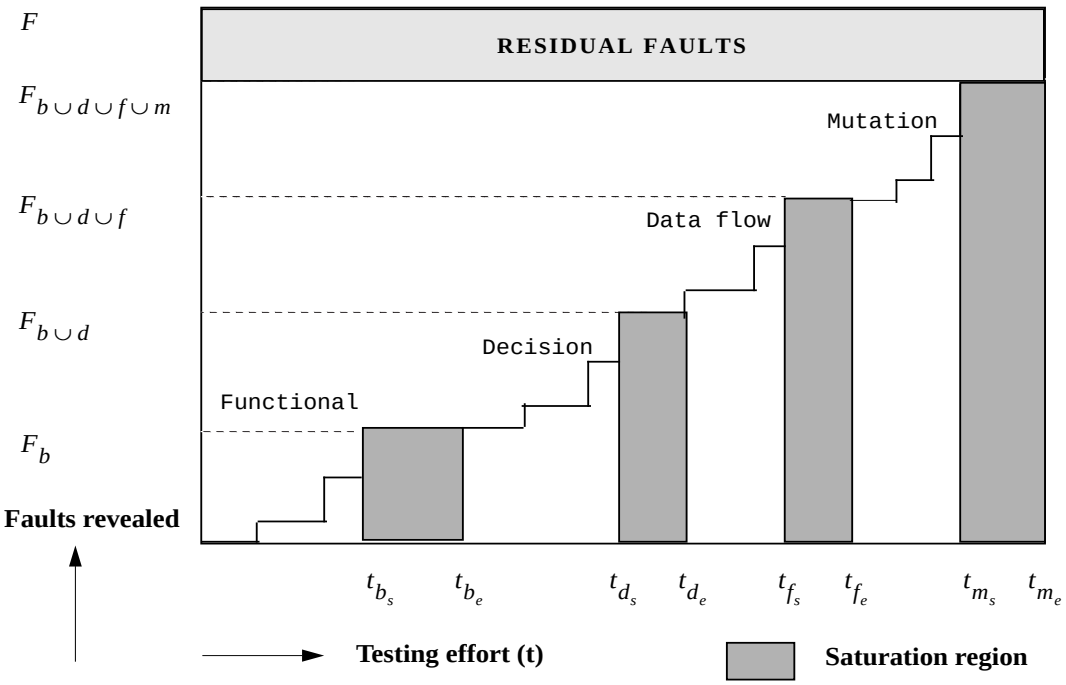


Figure 1: Saturation effect of testing methods assuming that the testing methods are applied in the following sequence: functional, decision, data flow, and mutation.

Testing effort

As testing progresses, data including calendar time, CPU-time spent in executing the software under test, and the number of test cases developed becomes increasingly available. Here we refer to the *late* phases of testing, where reliability growth models can be applied. A reliability growth model uses failure data generated during this testing phase. For example, Musa's execution time model [29] uses the total CPU-time spent executing the program under test; other researchers have used the number of test cases [10]. We consider the CPU-time and the number of test cases as indicators of *testing effort*. Thus, as testing effort increases, faults are discovered and removed. This may result in an increase in program reliability. We denote the testing effort by t_x, where x indicates the testing method that was in use when the effort was measured.

Limits of testing methods

At the start of testing, a program contains a certain number, say F, of faults. As testing progresses, the number of remaining faults decreases. This decrease may be stochastic, i.e. there exists a window in time such that the number of remaining faults at the end of the window is less than the faults remaining at the beginning. When applied, each testing method approaches a limit on the number of faults that it can reveal for a given program.

As shown in Fig. 1, we assume that for functional testing this limit is reached after t_{b_s} units of testing effort has been expended. Also, functional testing is shown to have revealed

F_b out of F faults when its limit is reached. In general, $F \geq F_b \geq 0$. Due to the imprecise nature of functional testing, it is difficult, if not impossible, to determine when such testing is complete. In practice a variety of criteria, both formal (e.g. a reliability estimate) and informal (e.g. market pressure), are used to terminate functional testing. If no other form of testing is used, this also terminates testing of the product.

It is reasonable to assume that as functional testing proceeds, the removal of faults found during testing causes the reliability of the software to grow. This may not be true in specific instances. For example, after a fault is found in version P_i of the software and removed to obtain version P_{i+1}, the reliability of P_i may be greater than that of P_{i+1}. This may happen if the fault removed from P_i exposes new faults in P_{i+1} (see page 261 in [29]). Despite this decrease in reliability when moving from P_i to P_{i+1} we assume that there exists a version $P_k, k > (i+1)$ such that the reliability of P_k is more than that of P_i.

Once the limit mentioned above is reached, no additional faults are found. A tester, generally unaware of the fact that the method's limit is reached, continues testing without discovering any more faults. If a reliability growth model is used for reliability prediction, continued functional testing beyond the limit will cause the resulting reliability estimate to improve with time, even though the true reliability of the program remains fixed. By increasing the number of test cases used in the saturation region, a reliability estimate can be iteratively improved to reach an unjustifiably optimistic value.

In accordance with the scenario charted earlier, let us suppose that after t_{b_e} units of functional testing, one switches to decision coverage based testing. New test cases are developed to cover decisions as yet uncovered. Eventually, decision coverage reaches its fault revelation limit after a total of t_{d_s} units of testing. At this point $F_{b \cup d}$ faults have been revealed, with F_d being the number of faults revealed by decision coverage. Note that at this point 100% decision coverage may not have been achieved. However, the tester is unaware that the limit has been reached and continues testing until t_{d_e} by which time all decisions have been covered. Empirical evidence suggests that $0 \leq F_b \leq F_{b \cup d} \leq F$.

Continuing in this manner, we assume that the next switch of testing methods occurs at time t_{d_e}, when data flow testing begins. Following this, mutation testing begins at time t_{f_e}. As shown in Fig. 1, the limits of data flow and mutation are reached at times t_{f_s} and t_{m_s}, respectively, with a total of $F_{b \cup d \cup f}$ and $F_{b \cup d \cup f \cup m}$ faults revealed. Based on empirical evidence, we assume that in most cases $0 \leq F_b \leq F_{b \cup d} \leq F_{b \cup d \cup f} \leq F_{b \cup d \cup f \cup m} \leq F$.

Empirical basis of the saturation effect

It is possible to construct examples of programs to argue theoretically that every structure based testing method will eventually reach a fault revelation limit and thus fail to reveal at least one or more faults. This saturation effect has been illustrated for several structure based methods by a few independent empirical studies [9, 15, 42]. Due to the imprecise nature of functional testing, a proof of functional testing's saturation effect appears to be infeasible at this time. Howden's work [17] provides some empirical justification for saturation in functional testing. Yet another measurement was done using two widely reported programs, TEX [21] and AWK [20]. Both programs have been in use for years and tested by Knuth and Kernighan, respectively, based on functionality. The results [14]

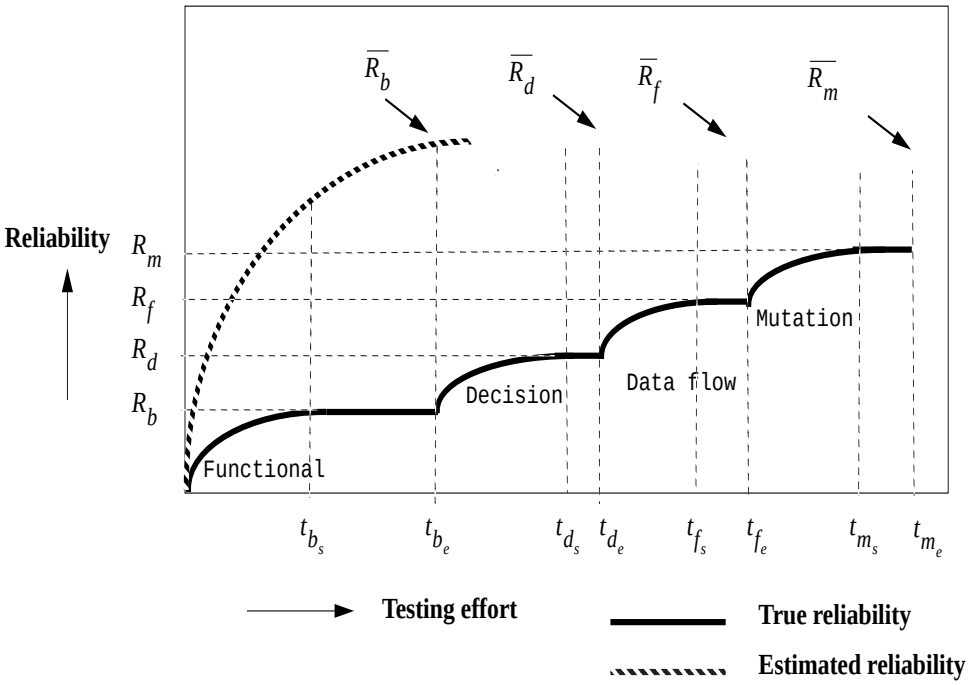


Figure 2: Overestimation of reliability due to saturation effect.

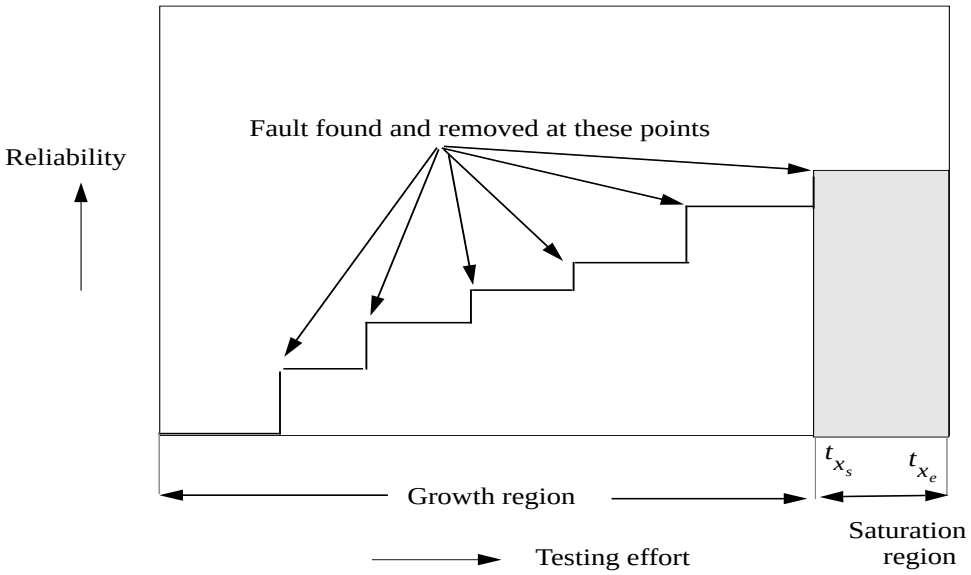
strongly suggest that (1) intensive functional testing may fail to test a significant part of the code and, therefore, (2) may fail to reveal faults in untested parts of the code. Both empirical observations justify the claim that functional testing, like the other testing methods, exhibits a saturation effect.

Reliability overestimation due to saturation effect

We now argue that the saturation effect can lead to overestimates of software reliability. Fig. 2 shows the reliability R as faults are removed, and the estimate of R denoted by $\bar{R}$. The testing effort axis is labelled the same as in Fig. 1. Assuming that faults are independent, R increases as faults are removed from the program. $\bar{R}$ may, however, either increase or decrease as faults are removed. This non-monotonic behavior of $\bar{R}$ is due to the time, or effort, dependence of the input data used by most models. For example, increasing inter-failure times will usually lead to increasing values of $\bar{R}$ as given by the Musa model.

We assume that the values of $\bar{R}$ generated by a model is a stochastically increasing estimate. This implies that even though $\bar{R}$ may fluctuate, it will eventually increase if the number of remaining faults decreases.

In Fig. 2, we indicate that as functional testing progresses, and faults are discovered and corrected, $\bar{R}$ increases. In practice, however, it is not possible to detect when a saturation point, t_{b_s} in this case, has been reached. Thus, testing may continue well past the saturation point. As shown in the figure, testing in the saturation region does not cause R to increase, though it causes $\bar{R}$ to increase. This increase in $\bar{R}$ is explained by observing that the last value in the sequence of inter-failure data, i.e. $(t - t_{b_s})$, increases without any new fault



t_{x_s} : Start of saturation region for testing technique x
t_{x_e} : End of saturation region for testing technique x

Figure 3: A realistic growth region.

being detected. This increase in $\bar{R}$, with R remaining constant, can lead to a significant overestimate of the reliability. In Fig. 2, $\bar{R}_b$ is the estimate of the true reliability R_b at the end of functional testing.

The above reasoning applies to other phases of testing as well, including testing based on white-box methods (which take program structure into account). In each case, there is a period of testing when the value of $\bar{R}$ increases even though R remains fixed. As before, such an increase can lead to overestimates as shown in Fig. 2.

Fig. 2 indicates that R is a monotonically increasing function of testing effort in the growth region. This may not be always true. In general, the growth region will appear as shown in Fig. 3. Thus, for example, if t denotes CPU-time spent in executing P, then R will grow during periods when faults are found; otherwise it will remain constant. This step-wise rise of R will cause $\bar{R}$ to fluctuate but increase stochastically, as governed by the underlying model.

4 New approaches for reliability estimation

In the discussion above we identified several limitations of the existing reliability models. For prediction these models use a part of the information that can be collected during testing, namely the operational profile, the times between successive failures or inherent fault density, and the fault detection rate. Other information such as code complexity and test coverage are not taken into account even if they are known to heavily affect reliability [19, 27]. These models use a black box approach to reliability modelling and

attempt to simulate the shape of a mathematical function, that is, the shape of the function that represents reliability growth as observed. The models disregard the causes responsible for the phenomenon represented by that function.

According to Littlewood and Strigini [23] this scarcity of information is the reason for the limited confidence we can gain from a reliability growth study: "... If we want to have an assurance of high dependability, using only information obtained from the failure process, then we need to observe the system for a very long time..." and this time can be unacceptably long in practical cases.

Even recent research trends did not change this status. Some authors [8, 32] observed that none of the current reliability growth models can be definitely considered as the model offering the most accurate estimation. They then suggested combining the different predictions obtained from several models. They proposed different criteria to weight the predictions statically or dynamically and in some cases [32] developed tools to automate model selection and combine their predictions. Others have suggested the use of re-calibration techniques [6, 7] that improve the predictive quality of reliability models by letting them "learn" from their past errors. These research efforts are once again suggesting a technical artifact to better pursue the shape of a mathematical function; they are not yet looking "inside the software".

Some researchers have proposed approaches that attempt to use additional code-related information to improve reliability prediction. As referenced above Miller and others [26] suggested combining the information provided by random testing on a binned input space with other information obtained from the analysis of the code. Ohba suggested [31] a model to predict reliability using a coverage based model. The relation between test coverage and reliability has been recently underlined by several researchers [11, 12, 24, 19, 40]. Some have proposed models of this relation [24, 41] and are now trying to validate these models. In the next section we outline yet another attractive approach to integrating the information about the testing coverage during the application of existing software reliability growth models.

We also outline an approach to reliability estimation that might be suitable for estimating software reliability in terms of the reliability of its components.

4.1 Coverage-based reliability estimation

We now describe an approach to incorporating coverage information in estimating software reliability. We begin by defining the notion of *useful* testing effort. A testing effort E_k is useful if and only if it increases some type of coverage. Note that the definition of usefulness does not specify which coverage should be increased for a test effort to be useful.

One might argue that in practice every test case, against which P has not been executed before, is useful. This argument is acceptable in accordance with our definition of usefulness if input domain coverage is considered to be one type of coverage. We know that there exist disjoint subsets $D_i, 1 \leq i \leq n, n \geq 0$, of the input domain D, known as *partitions*, such that $\cup D_i = D$. The behavior of P is identical for each input in a given partition. Thus, testing P on one element of D_i is equivalent to testing P on all elements of this partition. The problem, however, is that in practice it is not feasible to compute these partitions *a priori*. Instead, we rely on various testing methods to provide us with the relevant partitions.

Therefore, we assume that input space coverage is *not* one of the coverage types to be considered in determining whether an effort is useful or not. Later we will explain how this assumption affects reliability estimates based on time/structure models.

We have already referred to three types of coverages, namely, decision, data flow, and mutation. To illustrate the notion of usefulness, suppose that the first $k-1$ test cases have resulted in a decision coverage of 35%. Now if the kth test case increases this coverage, to say 40%, then we say that it is useful. In case the CPU time spent is the measure of testing effort, then an execution of P that causes an increase in decision coverage results in useful testing effort. Note that there are several ways of measuring structural coverage. It is not clear which is the best and should be used here. We capture the notion of *useless* effort in the definitions below.

Let e_i denote the effort spent during the ith execution of P. Then E_k can be expressed as:

$$E_k = \sum_{i=l_1}^{l_2} e_i \tag{1}$$

where e_{l_1} and e_{l_2}, respectively, are the efforts spent in the first and last executions of P during the kth failure interval.

The effort E_k defined in Eq. 1 consists of one or more atomic efforts e_i. However, an e_i may be useless. To account for such useless efforts, which may bias the inter-failure effort, we define

$$E_k^c = \sum_{i=l_1}^{l_2} \rho e_i \tag{2}$$

where l_1 and l_2 are as in Eq. 1 and ρ is the *compression ratio*. The quantity ρ can be defined in several ways. Below we provide a simple definition.

$$\rho = \begin{cases} 1 & \text{if } e_i \text{ increases coverage} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The use of ρ compresses the inter-failure effort, E_k to E_k^c by ignoring the atomic effort that has been found useless. This process is illustrated in Fig. 4. Along the thick horizontal line, the testing effort is indicated. The leftmost upward pointing arrow indicates the instant when testing began, subsequent upward arrows mark failure points. Two consecutive downward arrows bracket the atomic effort. The first atomic effort is bracketed by the leftmost upward arrow and the first downward arrow. The sequence of total effort spent between successive failures is indicated by the sequence of shaded boxes labelled "Observed". The sequence of shaded boxes, labelled "Useful" just below this line indicate the *filtered* effort data obtained by applying the compression ratio to the observed data.

The filtered effort data can now be used in any of the existing time-based models. Thus, for example, if the Musa model is being used to predict reliability, then the filtered data can serve as the modified sequence of inter-failure times. If $\bar{R}_m$ and $\bar{R}_f$ are reliability estimates generated by the Musa model using, respectively, the original and filtered inter-failure times,

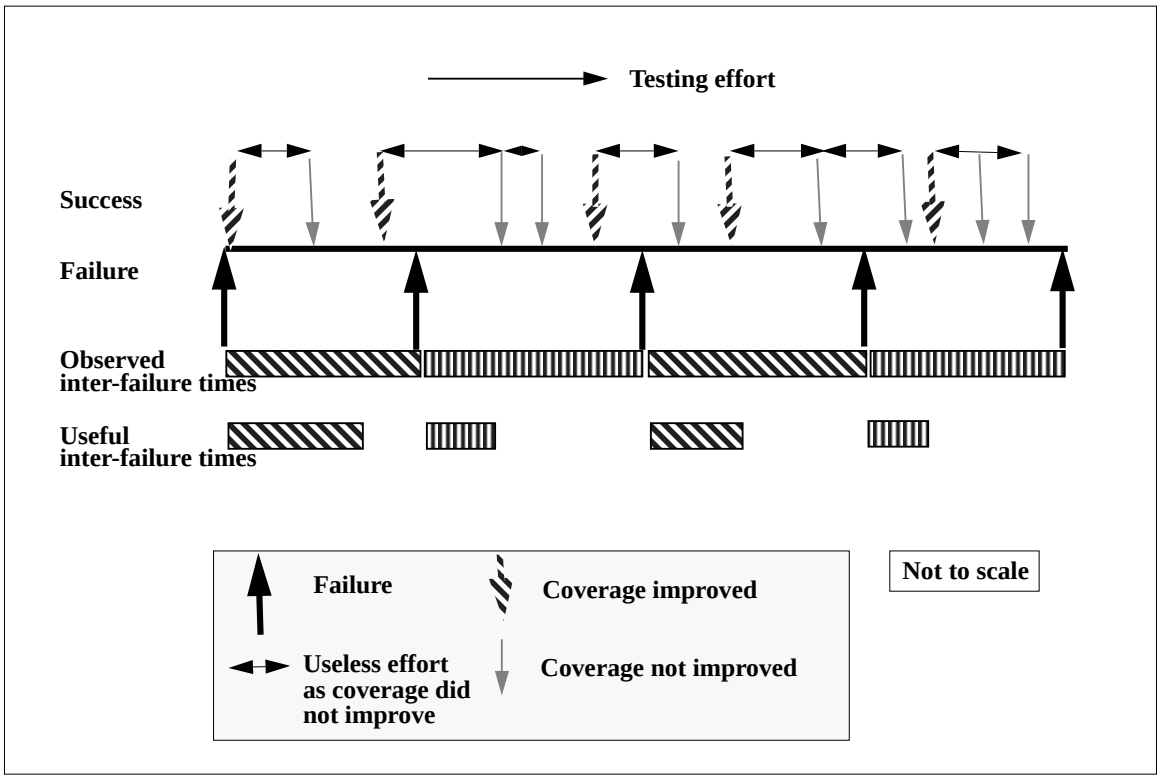


Figure 4: Filtering failure data using coverage information.

then $\bar{R}_f < \bar{R}_m$. The example below illustrates this relationship. Thus, filtering leads to a more realistic reliability estimate than the approach that uses unfiltered data. The filtered data can also be applied to any of the other models such as the Goel-Okumoto model [16]. Empirical studies to investigate the choice of ρ are reported by Chen [11].

4.2 Module based reliability estimation

Here we sketch a method for reliability estimation that attempts to integrate reliability estimation into the software development process. Figure 5 sketches our approach. We assume that the software system under development consists of multiple interacting modules. Further, these modules are developed, tested, and integrated in accordance with some project development plan.

We would like to be able to estimate the reliability of individual modules and combine these estimates to obtain the reliability estimates for larger modules. Systems reliability theory suggests methods for combining reliability estimates of individual modules to obtain that of the system composed of these modules(see Chapter 4 of [29]). There are two problems that inhibit the straighforward application of these methods. First, in the presence of module dependencies one needs to use a more elaborate procedure for combining module reliabilities [22]. Second, one needs a method for obtaining reliabilities of individual modules. We discuss these problems below.

4.2.1 Module dependencies

To solve the first problem above we need to quantify module dependencies. From the architecture of the system under development and the details of each module one may possibly determine the dependency amongst modules. Let us briefly examine the notion of module dependency. Let M_1 and M_2 be two software modules, and assume that the software execution sequence is M_1 followed by M_2. Assume that M_1 and M_2 have each been tested in isolation. Let X_1 be a random variable which takes on the value 0 if M_1 works according to its specification, and 1 otherwise. Similarly, let X_2 be a random variable which takes on the value 0 if M_2 works according to its specification, and 1 otherwise.

Using $P(X_1 = 1) = p_1$ and $P(X_2 = 1) = p_2$ to denote the failure probabilities of modules M_1 and M_2, respectively, standard naive analyses which typically assume *independence* between M_1 and M_2 conclude that:

$$P(\text{system failure}) = P(X_1 = 1) \times P(X_2 = 1) = p1 \times p2.$$

It should be clear, however, that X_1 and X_2 in general cannot be assumed to be independent. We can say that X_2 is independent of X_1 if and only if $P(X_2|X_1) = P(X_2)$. It is easy to argue that this relation does not hold for software modules M_1 and M_2, in general. For example, this occurs when the execution of M_1 causes a change in the state of the execution environment of M_2. That is, even if M_1 executes correctly, its execution places the system in a state that effects the probability of failure of M_2 and makes this probability different from the probability given by the distribution $[P(X_2 = 1) = p_2, P(X_2 = 0) = 1 - p_2]$ that was obtained by *testing M_2 in isolation*. Thus, in general the failure behavior of module M_2 is *dependent* on the behavior of module M_1, or equivalently, random variable X_2 is *dependent* on random variable X_1.

Now suppose that the execution sequence is M_1 followed by M_2, with a feedback loop to M_1. Then not only is M_2 dependent on M_1, but M_1 is also dependent on M_2. Thus we can also say that, in general,

$$P(X_1|X_2) \neq P(X_1)$$

so that X_1 and X_2 are dependent random variables, or M_1 and M_2 are dependent modules.

4.2.2 Module reliability

To solve the second problem listed above we need ways to estimate the reliability of individual modules. If a module is small, say 2000 lines of code, it may not be possible to collect enough failure data for the application of reliability growth models. We therefore need alternate methods for estimating the reliability of modules, small or large. The use of code coverage to predict module reliability seems attractive for at least two reasons. First, it is generally feasible to use coverage testing on modules. One may measure the adequacy of a test set for a module using one or more of the code coverage criteria available. Preliminary work has shown a close correlation between various code coverage measures and the reliability of components of sizes less than 200 lines of code [25]. Based on code coverage data one may be able to predict the reliability of a module and an error in this estimate.

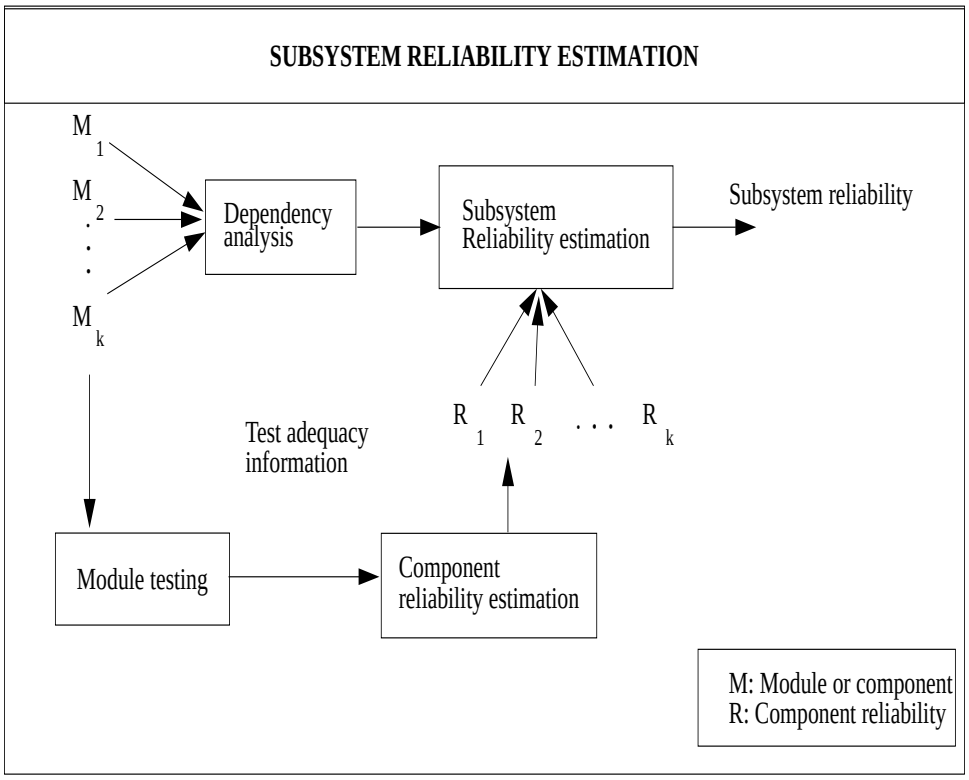


Figure 5: Estimation of component reliability.

These estimates, together with module dependability information may be used to obtain reliability estimates of larger systems made of smaller modules.

4.2.3 Comparing different reliability estimates

By using the module-based approach one obtains the reliability of a subsystem or system in terms of its probability of failure on any given run, where a "run" is defined as in [29]. As shown in Figure 5, one may also obtain reliability estimates using other models, e.g., one based on the data filtering approach. However, in order to be able to compare the time based reliability estimates (e.g. probability of no failure in a specified time) with the pure probabilistic estimate (e.g. probability of failure in any given run), the latter can be transformed into a time based estimate as explained below.

Let the probability of system failure on any given data input be denoted by p. Since consecutive executions of the software system can be considered to be independent, the number of successful runs of the software until the a failure can be described by a geometric random variable X with parameter p, i.e., $P(X = k) = (1-p)^{k-1}p$ for $k \geq 1$. We can convert this notion of probability of failure on any given run into the probability of failure in a given time using the average time required per run, and the expected number of runs $E[X] = 1/p$ until the next failure.

5 Summary

We have pointed out the weaknesses of existing methods for software reliability estimation. We have done so by pointing to (a) the difficulties in obtaining accurate operational profiles and (b) errors in reliability estimates that may arise due to inaccuracies in the operational profile. We then propose two new approaches for reliability estimation that are intended to overcome the deficiencies of the existing approaches. As the primary purpose of this paper is to critique the existing methods of reliability estimation, we have not included data in support of our arguments and proposed methods; instead we have cited relevant publications. It is our hope that the the critique herein will motivate reliability researchers to move in a new direction and warn practitioners to use the existing methods with extreme caution.

Acknowledgement

We express our thanks to Jim Berger, Nozer Singpurwalla, Ronnie Martin and the participants of the Purdue-Bellcore-US West workshop at Boulder 1993 and the round table discussion on Issues in Software Reliability Estimation held on April 14, 1994 at Bellcore.

References

[1] F. B. Bastani, "On the Uncertainty in the Correctness of Computer Programs," IEEE Transactions On Software Engineering, Vol. SE 11, No. 9, September 1985.

[2] F. B. Bastani, G. DiMarco, and A. Pasquini, "Experimental evaluation of fuzzy set based measure of program correctness," *Proceedings of the Fifteenth ICSE*, IEEE Press, 1993.

[3] F. B. Bastani and A. Pasquini, "Assessment of a sampling method for measuring safety critical software reliability," *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, Monterey, CA, November 6-9, 1994.

[4] D. B. Benson, "Magellan spacecraft will need frequent guidance from Earth", ACM Software Engineering Notes, Vol. 15, No. 2, 1990.

[5] J. Brown and M. Lipow, "Testing for software reliability," *Proceedings of the International Conference on Reliable Software*, Los Angeles, April 1975, pp. 518-527.

[6] S. Brocklehurst, P. Y. Chan and B. Littlewood, "Recalibrating software reliability models", IEEE Transactions on Software Engineering, Vol. SE-16, No. 4, 1990.

[7] S. Brocklehurst and B. Littlewood, "New ways to get accurate reliability measures", IEEE Software, Vol. 9, No. 4, July 1992.

[8] S. Brocklehurst, M. Lu, B. Littlewood, "Combination of predictions obtained from different software reliability growth models", *Proceedings of the 10th Annual Software Reliability Symposium,* June 1992, Denver, Colorado

[9] T. A. Budd, "Mutation analysis of program test data," Dissertation, Yale University, May, 1980.

[10] R. C. Cheung, "A user oriented software reliability model," *IEEE Transactions on Software Engineering*, Vol. SE-6, No. 2, March 1980, pp. 118-125.

[11] M. H. Chen, "Tools and techniques for testing-based software reliability estimation," Doctoral Thesis, Department of Computer Science, Purdue University, W. Lafayette, IN 47907.

[12] M. H. Chen, A. P. Mathur, and V. J. Rego, " Effect of testing techniques on software reliability estimates obtained using time domain models," *Proceedings of the 10th Annual Software Reliability Symposium*, IEEE Reliability Society, Denver, Colorado, pp. 116-123, June 25-26, 1992.

[13] A. Currit, M. Dyer, and H. Mills, "Certifying the reliability of software", IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, 1986.

[14] J. R. Horgan and A. P. Mathur, "Assessing tools in research and education," *IEEE Software*, May 1992, pp. 61-69.

[15] M. R. Girgis and M. R. Woodward, "An experimental comparison of the error exposing ability of program testing criteria," *Proceedings of the Workshop on Software Testing, Validation, and Analysis*, Banff, Canada, July 15-17, 1986.

[16] A. L. Goel, "Software reliability models: assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 2, December 1985, pp. 1411-1423.

[17] W. E. Howden, "Functional testing," *IEEE Transactions Software Engg.*, Vol. SE-6, No. 2, pp. 162-169, March 1980.

[18] D. Hamlet, "Are we Testing for true reliability ?," IEEE Software, July 1992, pp. 21-27

[19] M. A. Hennell, "Testing for the achievement of software reliability", *Reliability Engineering and System Safety,* Vol. 32, pp. 119-134, 1991.

[20] B. W. Kernighan, *Personal communication.*

[21] D. E. Knuth, "TEX : The Program," Addison Wesley, Reading, MA, 1986.

[22] D. V. Lindley and N. Singpurwalla, "Multivariate distributions for the life lengths of components of a system sharing a common environment," *J. of Applied Prob.*, 23, pp. 418-431, 1986.

[23] B. Littlewood and L. Strigini, "Validation of ultra-high dependability for software-based systems", Communication of the ACM, Vol. 36, No. 1, Jan. 1993.

[24] Y. K.Malaiya, N. Li, J. Bieman, R. Karcich, and R. Skibbe, "The relationship between test coverage and reliability," *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, Monterey, CA, November 6-9, 1994.

[25] A. P. Mathur and W. E. Wong, "An empirical comparison of mutation and data flow-based test adequacy criteria," Journal of Software Testing, Verification, and Reliability, Vol. 4, No. 1, pp 9-31, March 1994.

[26] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill and J. M. Voas, "Estimating the probability of failure when testing reveals no failures", IEEE Transactions on Software Engineering, Vol. SE-18, No. 1, 1992.

[27] J. C. Munson and T. M. Khoshgoftaar, "The use of software complexity metrics in software reliability modeling", Proceedings of the ISSRE '91, IEEE Computer Society, 1991.

[28] J. D. Musa, "Operational profiles in reliability engineering," *IEEE Software*, March 1993, pp. 14-32.

[29] J. D. Musa, A. Iannino, and K. Okumoto, *Software reliability: measurement, prediction, application*, McGraw-Hill, New York, 1987.

[30] E. Nelson, "Estimating software reliability from test data," Microelctronics and Reliability, Vol 17, Pergamon Press, New York, 1978.

[31] M. Ohba, "Software quality = test quality $\times$ test coverage," *Proceedings of 6th ICSE*, Tokyo, September 1992, pp. 287-293.

[32] M. R. Lyu and A. Nikora, "Applying reliability models more effectively", IEEE Software, Vol. 9, No. 4, July 1992.

[33] J. H. Poore, H. D. Mills and D. Mutchler, "Planning and certifying software system reliability", IEEE Software, Vol. 10, No. 1, January 1993.

[34] D. L. Parnas and A. J. Van Schouwen, and S. P. Kwan, "Evaluation of safety critical software," Communications of the ACM, Vol. 33, No. 6, June 1990.

[35] A. Pasquini, "Measuring software reliability in a space application", Proceedings of the European Safety and Reliability Conference, Elsevier Applied Science, June 1992.

[36] A. Pasquini and G. DiMarco, "A fault domain based measure of software reliability," *Proceedings of the 10th Annual Software Reliability Symposium*, IEEE Reliability Society, Denver, Colorado, pp. 9-15, June 25-26, 1992.

[37] H. A. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," *Proceedings of FTCS-18*, 1993, Tolouse, France, pp. 208-217.

[38] Z. Segall, et al., "Fiat - fault injection based automated testing environment," *Proceedings of FTCS-18*, 1988, pp. 102-107.

[39] M. L. Shooman, "A micro software reliability model for prediction and test apportionment", Proceedings of the International Symposium on Softwre Reliability Engineering, Austin, Texas, IEEE Computer Society, 1991, pp. 52-59

[40] A. Veevers "Software coverage metrics and operational reliability," *Proceedings of the Symposia Series*, Vol. 17, pp. 67-69, 1990.

[41] A. Veevers and A. Marshall, "A relationship between software coverage metrics and reliability," Software Testing, Verification and Reliability, Vol. 4, pp. 3-8, 1994.

[42] P. J. Walsh, "A measure of test case completeness," Dissertation, State University of New York, Binghamton, NY, 1985.

[43] W. E. Wong, "On mutation and data flow," Technical Report, SERC-TR-P149 , Software Engineering Research Center, Purdue University, W. Lafayette, IN 47907, 1991.

[44] P. Zave, "Feature interactions and formal specifications in telecommunications," IEEE Computer, August 1993, pp. 20-29.