Software Reliability—Status and Perspectives

C. V. RAMAMOORTHY, FELLOW, IEEE, AND FAROKH B. BASTANI, MEMBER, IEEE

Abstract—It is essential to assess the reliability of digital computer systems used for critical real-time control applications (e.g., nuclear power plant safety control systems). This involves the assessment of the design correctness of the combined hardware/software system as well as the reliability of the hardware. In this paper we survey methods of determining the design correctness of systems as applied to computer programs.

Automated program proving techniques are still not practical for realistic programs. Manual proofs are lengthy, tedious, and error-prone. Software reliability provides a measure of confidence in the operational correctness of the software. Since the early 1970's several software reliability models have been proposed. We classify and discuss these models using the concepts of residual error size and the testing process used. We also discuss methods of estimating the correctness of the program and the adequacy of the set of test cases used.

These methods are directly applicable to assessing the design correctness of the total integrated hardware/software system which ultimately could include large complex distributed processing systems.

Index Terms—Correctness probability, error-counting models, error seeding, error size, evaluation of test cases, nonerror-counting models, software fault, software reliability models, testing and debugging phase, testing process, validation phase.

I. INTRODUCTION

IN ORDER TO pave the way for the use of digital computers for critical applications, like nuclear power plant safety control systems, it must be shown that the computer system meets the specified reliability constraints. The theoretical basis for methods of estimating the reliability of the hardware is well developed [6]. Since the early 1970's, several models have been proposed for estimating software reliability and some related parameters, such as mean time to failure (MTTF), residual error content, and other measures of confidence in the software. However, software reliability theory is still rudimentary. In fact, the basis of most software reliability models is often viewed skeptically [57], [114]. Since there is no physical deterioration or random malfunction in software, it is preferable to prove that the software meets (or does not meet) its requirements specification. However, current program verification techniques cannot cope with the size and complexity of software for real-time applications [23]. Similarly, exhaustive testing is ruled out by the large number of possible inputs.

Besides, testing is limited by other factors, namely, the difficulty in verifying the output corresponding to an input and the lack of realistic inputs (e.g., in a missile defense system).

Most software reliability models attempt to estimate the reliability of the software based on its error history, either during its debugging phase (when errors which are detected are corrected) or during its validation phase (when errors are not corrected). For example, an operating system which crashes more often than another has a lower reliability. The difficulty lies in the quantification of this measure.

In the following we discuss some of the software reliability models proposed so far. We attempt to give a physical feel for the models, their assumptions, and the type of data required, and some comments on their applicability to real projects. In Section II we give a definition of software reliability. The existing reliability models are classified in Section III. Section IV discusses models applicable during the debugging phase (these are called “reliability growth models”). We introduce the concepts of error size and testing process and use these to discuss the Jelinski-Moranda models [69], the Shooman model [98], the Schick-Wolpert model [92], the Musa model [75], the Littlewood model [56], the Littlewood-Verrall model [53], and an input domain based stochastic model [85]. We also discuss a general framework for these growth models. In Section V we first discuss the Nelson model [109]. Then we introduce the concepts of equivalence classes and continuity in the input domain. These are used for estimating the correctness probability of the program [82]. In Section VI we briefly discuss the application of error seeding to estimating the efficiency of different testing strategies, the reliability of the set of test cases used (i.e., a measure of how well the software has been tested), and the correctness probability of the program.

The following references discuss other reliability models and their applications: [14], [21], [68], [86], [101], [106].

II. DEFINITION

Software reliability has been defined as the probability that a software fault which causes deviation from required output by more than specified tolerances, in a specified environment, does not occur during a specified exposure period [109].

Thus, the software need be correct only for inputs for which it is designed (specified environment). Also, if the output is correct within the specified tolerances in spite of some error, then the error is ignored. This may happen in the evaluation of complicated floating point expressions where many approximations are used (e.g., polynomial approximations for COSINE, SINE, etc.).
It is possible that a failure may be due to errors in the compiler, operating system, microcode, or even the hardware. These failures are ignored in estimating the reliability of the application program. However, the estimation of the overall system reliability will include the correctness of the supporting software and the reliability of the hardware.

The exposure period should be independent of extraneous factors like machine execution time, programming environment, etc. For many applications, the appropriate unit of exposure period is a run corresponding to the selection of a point from the input domain (specified environment) of the program. However, for some programs (e.g., an operating system) it is difficult to determine what constitutes a "run." In such cases the unit of exposure period may be the calendar or CPU time. Thus, we have

\[
R(i) = \text{reliability over } i \text{ runs} = P\{\text{no failure over } i \text{ runs}\}
\]

or

\[
R(t) = \text{reliability over } t \text{ seconds} = P\{\text{no failure in interval } [0, t]\}.
\]

(\(P\{E\}\) denotes the probability of the event \(E\).)

Definition (1) leads to an intuitive measure of software reliability. Assuming that inputs are selected independently according to some probability distribution function, we have

\[
R(i) = [R(1)]^i = (R)^i
\]

where \(R \equiv R(1)\). We can define the reliability \(R\) as follows:

\[
R = 1 - \lim_{n \to \infty} \frac{n_f}{n}
\]

where

\(n = \text{number of runs},\)
\(n_f = \text{number of failures in } n \text{ runs.}\)

This is the operational definition of software reliability. We can estimate the reliability of a program by observing the outcomes (success/failure) of a number of runs under its operating environment. If we observe \(n_f\) failures out of \(n\) runs, the estimate of \(R\), denoted by \(\hat{R}\), is

\[
\hat{R} = 1 - \frac{n_f}{n}.
\]

This method of estimating \(R\) is the basis of the Nelson model [109].

III. Classification of Software Reliability Models

Fig. 1 shows a classification of some of the existing software reliability models. The classification scheme is based primarily on the phase of software life cycle during which the model is applicable. The main feature of a model serves as a subclassification. Most of the existing models can be used during the software testing and debugging phase, validation phase, or operational phase. There are several models which do not yield a reliability figure but which measure some parameters useful in evaluating a given software system, for example, the number of remaining errors, test reliability, or confidence in the program. These models are classified as models which yield some measure of the correctness of the program.

A. Testing and Debugging Phase

During this phase the implemented software is tested and debugged. It is often assumed that the correction of errors does not introduce any new errors. Hence, the reliability of the program increases, and therefore the models used during this phase are also called reliability growth models.

The error-counting models estimate both the number of errors remaining in the program as well as its reliability. Deterministic, Bayesian, and Markov error-counting models have been proposed. The deterministic models assume that if the model parameters are known, then the correction of an error results in a known increase in reliability. This category includes the Jelinski-Moranda [47], Shooman [24], [95], [96], Musa [75], and Schick-Wolverton [92] models. The general Poisson model [4] is a generalization of these four models. The Bayesian model due to Littlewood [56] models the (usual) case where larger errors are detected earlier than smaller errors. All the preceding models neglect the time required to correct an error. This aspect is modeled as a Markov process by Trivedi and Shooman [108]. The model also yields an estimate of the availability of the software system.

The number of errors remaining in the program is useful in estimating the maintenance cost. However, with these models it is difficult to incorporate the case where new errors may be introduced in the program as a result of imperfect debugging. Further, the reliability estimate is unstable if the estimate of the number of remaining errors is low [33], [60].

The nonerror-counting models only estimate the reliability of the software. The Jelinski-Moranda geometric de-eutrophication model [69] and a simple model used in the Halden project [21] are deterministic models in this category. The stochastic models consider the situation where different errors have different effects on the failure rate of the program. The correction of an error results in a stochastic increase in the reliability. A stochastic model based on the nature of the input domain of the program is developed in [85].

The deterministic and the stochastic models assume that the reliability is unchanged during the interval between consecutive error corrections. While this is true in an absolute sense, the reliability estimate as perceived by the person testing the program, increases as the number of successful consecutive runs increases. This situation is modeled by the Bayesian growth models. The model due to Littlewood and Verrall [53] and the Mixed-Gamma model developed in [85] are stochastic and Bayesian.

The models described above treat the program as a black box. That is, the reliability is estimated without regard to the structure of the program. The validity of their assumption usually increases as the size of the program increases. Since programs for critical control systems may be of medium size...
only, these models are mainly used to obtain a preliminary estimate of the software reliability.

B. Validation Phase

Software developed for critical applications, like air-traffic control, must be shown to have a high reliability prior to actual use. At the end of the development phase, the software is subjected to a large amount of testing in order to estimate the reliability. Errors found during this phase are not corrected. In fact, if errors are discovered the software may be rejected.

The Nelson model [109] is based on statistical principles. The software is tested with test cases having the same distribution as the actual operating environment. The operational definition (Section II) is used to obtain the reliability estimate. The only disadvantage of the Nelson model is that a large amount of test cases are required in order to have a high confidence in the reliability estimate. The approach developed in [82] reduces the number of test cases required by exploiting the nature of the input domain of the program. An important feature of this model is that the testing need not be random—any type of test-selection strategy can be used.

C. Operational Phase

In most cases the successive inputs to the program are not independent. For example, in process control systems the sensor inputs are correlated in time due to physical constraints. As a first approximation, the operating environment of a program can be viewed as consisting of a number of different distributions. For a certain (random) period the input is selected using one distribution and then a transition is made to another distribution. Littlewood [55] and Cheung [18] have modeled the input distribution selection mechanism as a Markov process.

The input domain based model developed in [82] incorpo-
rates the feature that the outcome of a particular run may
depend on the outcomes of the previous runs. For example,
if a new test case is identical to some earlier, successful test
case, then its outcome will be successful with certainty. Thus,
in a sense, the software is being continuously validated during
the operational phase.

D. Maintenance Phase

During the maintenance phase the possible activities are:
error correction, addition of new features, and improvements
in algorithms. Any of these activities can perturb the reliability
of the system. The new reliability can be estimated using
the models for the validation phase. However, it may be possible
to estimate the change in the reliability using fewer test cases
by ensuring that the original features have not been altered.
We do not know of any existing software reliability models
applicable during this phase. The input domain based ap-
proach is discussed in [7].

E. Correctness Measures

Software for critical applications must have a reliability esti-
mate of 1. In these cases the confidence in the estimate is very
important. We have classified all methods which provide some
measure of how well the software has been tested under the
"correctness measures" category.

Error seedling [92] and program mutation [22] directly
measure the test reliability [38]—all seeded errors and all
nonequivalent mutations should be detected by the test cases.

The software science (phenomenological) approach due to
Halstead [40] gives an empirical prediction of the error con-
tent of the software based on the number of operators and
operands. If the predicted error content is high, then more
testing is required.

The statistical approach developed at TRW [109] yields a
confidence in the reliability estimate based on statistical
sampling theory. A method of estimating the representativ-
eness of the test cases, that is, whether the test cases are ade-
quately representative of the operating environment, is dis-
cussed in [15]. Ehrenberger has developed statistical models
based on the coverage of the input domain, program functions,
and program structures [14].

An input domain based model for directly estimating the
correctness probability of a program is developed in [82]. An
uncertainty measure based on fuzzy subsets of the input
domain is discussed in [84].

In this section we have classified many software reliability
models without describing them in detail. References [14],
[21], [92], and [106] contain a detailed survey of most of
the models. Some of the models are discussed in the next two
sections.

IV. SOFTWARE RELIABILITY GROWTH MODELS

During the debugging phase the software is tested and all
errors which are detected are corrected. Assuming that no
new errors are introduced, the reliability increases. Hence,
the models used for assessing software reliability based on its
error history are called reliability growth models. Similar
growth models have been developed and used for assessing the
reliability of hardware design [25], [31].

In this section we first develop a general framework for
software reliability growth models using the concepts of error
sizes and testing process. Then we discuss two classes of these
models, namely, error-counting models and nonerror-counting
models. The validation of some of the models and their ap-
lication in other aspects of software development are also
discussed.

A. Error Sizes

A program P maps its input domain I into its output space
O, as shown in Fig. 2. Each element in I is mapped to a unique


Fig. 2. Functional view of a program.


Fig. 3. Elements affected by an error.


Authorized licensed use limited to: University of Nebraska Omaha Campus. Downloaded on April 01,2010 at 10:43:30 EDT from IEEE Xplore. Restrictions apply.
an error is subtle if it has a small size since then it is relatively more difficult to detect the error. The size of an error depends on the way the inputs are selected. Good test case selection strategies, like boundary value testing, path testing, and range testing, magnify the size of an error since they exercise error-prone constructs. Likewise, the observed (effective) error size is lower if the test cases are randomly chosen from the input domain.

We can generalize the notion of “error size” by basing it on the different methods of observing programs. For example, an error has a large size visually if it can be easily detected by code reading. Similarly, an error is difficult to detect by code review if it has a small size (e.g., only when one character is missing).

There is an important relationship between the total size of the remaining errors in a program $V_{er}$ and its operational reliability $R(1)$, as defined in Section II:

$$V_{er} = 1 - R(1),$$

i.e.,

$$V_{er} = \lim_{n \to \infty} \frac{n_f}{n}$$

where $n_f$ is the number of failures in $n$ runs. Thus, the size of the remaining errors can be estimated as follows:

$$\hat{V}_{er} \approx \frac{n_f}{n}.$$

The development phase is assumed to consist of the following cycle:

1) the program is tested until an error is found,
2) the error is corrected and step 1) is repeated.

As we have noted above, the error history of a program depends on the testing strategy employed, so that the reliability models must consider the testing process used.

B. Testing Process

Fig. 4 illustrates a simple example where the error history is strongly dependent on the testing process (see also [94]). Assume that the program has three paths which partition the input domain into three disjoint subsets [Fig. 4(a)]. The shaded regions in the figure correspond to set of elements affected by the errors in the program. Some errors affect elements in more than one path. The errors can also overlap, that is, an input can be affected by more than one error.

Fig. 4(b) illustrates a possible error history when each input is considered as equally likely. Initially errors are frequently detected. As these are corrected, the interval between error detection increases since fewer errors remain. Fig. 4(c) shows the case where a path is tested “well” before testing another path. Thus, whenever a switch is made to a new path the error detection rate increases. Fig. 4(c) also corresponds to the case where different test case selection strategies are used. In the beginning test cases are selected randomly (equally likely) since the probability of detecting errors is high. Later, more elaborate strategies like boundary value testing are used. These exercise error-prone constructs and thus have a higher probability of detecting errors.
\[ f(T_j(k)) = \text{severity of testing process relative to operational distribution}; 0 \leq f(T_j(k)) \leq 1/\lambda_j. \]

Hence
\[ R_j(k|\lambda_j) = P\{\text{no failure after } k \text{ runs} \mid \lambda_j\} \]
\[ = \prod_{i=1}^{k} P\{\text{no failure on the } i\text{th run} \mid \lambda_j\} \]
since successive test cases have independent failure probability. Hence
\[ R_j(k|\lambda_j) = \prod_{i=1}^{k} (1 - f(T_j(i)))\lambda_j, \]
i.e.,
\[ R_j(k) = E_{\lambda_j} \left[ \prod_{i=1}^{k} (1 - f(T_j(i)))\lambda_j \right] \]
(3)
where \( E_{\lambda_j} \cdot \cdot \cdot \) is the expectation over \( \lambda_j \).

For many types of software, e.g., operating systems and real-time process control systems, it is difficult to identify “runs” or there may be a very large number of such runs if we assume that each cycle constitutes a run. In these cases, it is simpler to work in continuous time. The above relation becomes
\[ R_j(t) = E_{\lambda_j} \left[ e^{-\lambda_j \int_0^t f(T_j(s))ds} \right] \]
(4)
where
\[ \lambda_j = \text{failure rate after } j\text{th failure}; 0 < \lambda_j \leq \infty, \]
\[ T_j(s) = \text{testing process at time } s \text{ after } j\text{th failure}, \]
\[ f(T_j(s)) = \text{severity of testing process relative to operational distribution}; 0 \leq f(T_j(s)) \leq \infty. \]

Remarks:
1) As we have noted above, \( f(T_j(s)) \) is the severity of the testing process relative to the operational distribution, where the testing severity is the ratio of the probability that a run based on the test case selection strategy detects an error to the probability that a failure occurs on a run selected according to the operational distribution. Obviously, during the operational phase, \( f(T_j(s)) = 1 \). Thus, a testing strategy which simulates the operational input would be ideal from a reliability modeling point of view since it would simplify the model considerably. But this conflicts with our need to improve the software by detecting as many errors as possible by employing nonrandom testing strategies such as path testing, function testing, and boundary value testing. In general, it is difficult to determine the severity of these test cases and most models assume that \( f(T_j(s)) = 1 \). However, for some testing strategies we can attempt to quantify \( f(T_j(s)) \). For example, in function testing the severity increases as we switch to untested functions since these are more likely to contain errors than functions which have already been tested.

2) In the continuous case, the time is the CPU time.

3) The software reliability models discussed below can be applied (in principle) to any type of software. However, their validity increases as the size of the software and the number of programmers involved increases.

4) This process is a type of doubly stochastic process; these processes were originally studied by Cox in 1955 [19].

1) Error-Counting Models: The models discuss in this section attempt to estimate the software reliability in terms of the estimated number of errors remaining in the program. The Jelinski-Moranda model [47] was the first error-counting model. The Shooman model [24], [95] underwent some changes [96], [98] and is now similar to the Jelinski-Moranda model. The Schick-Wolverton model [92] extended the Jelinski-Moranda model by incorporating a factor representing the severity of the test cases. The Musa model [75] is equivalent to the Jelinski-Moranda model. However, it is better developed and is the first model to insist on execution time. Recently, Littlewood [56] developed a model where the failure rate of a software error is stochastically decreasing, unlike the previous models which assume that all errors have the same failure rate.

a) General Poisson Model: The general Poisson model (GPM) is discussed in [4]. It generalizes the Jelinski-Moranda linear de-eutrophication model [47], [69], the Shooman model [24], [95], [96], [98], and the Schick-Wolverton model [91], [92]. The key parts of the Musa model are also generalized by this model.

The GPM model assumes that [see (4)]
\[ f(T_j(s)) = \alpha e^{-\alpha s}, \]
\[ \lambda_j = (N - M_j)\phi \]
where
\[ N = \text{number of errors originally present}, \]
\[ M_j = \text{number of errors corrected before the } j\text{th failure and after the } (j - 1)\text{th failure}, \]
\[ \alpha, \phi = \text{constants} \]
\[ \therefore R_j(t) = e^{-\phi(N-M_j)}e^{\alpha}. \]

Assumptions: The assumptions of the GPM model are as follows:
1) consecutive inputs have independent failure probabilities,
2) all errors have the same disjoint failure rate \( \phi \),
3) the severity of the testing process is proportional to a power of the elapsed CPU time,
4) no new errors are introduced.

Discussion: Assumption 1) has already been discussed above. Assumption 2) is a major drawback of these models [57]: earlier errors are likely to have a larger failure rate since they are detected more easily. Assumption 3) depends to a large extent on the testing strategy used. Intuitively, as time increases, the severity of the testing increases [92]. Assumption 4) is not true in general and can lead to invalid estimates [4]. Musa [75] partly overcomes this by estimating the total number of errors to be eventually detected.

The GPM-based models have been applied to several projects with mixed results [4], [21], [41], [66], [69], [75], [78], [92], [101], [102]. Unfortunately, with the exception of the Musa model [41], [75], [78], the time used is the calendar time, partly due to the lack of data. Musa [78] has also directly validated the assumptions of his model using several sets of data.
Application: As discussed in the derivation of (4), the execution time should be used. Thus, the data required are the CPU time between failures and the number of errors corrected at each time. The validity of the assumptions increases as the size of the programs (in terms of code length and the number of programmers) increases. If the number of errors discovered is small, then the maximum likelihood estimate of the parameters is unstable [33]. Littlewood and Verrall [60] have derived general conditions which the error data must satisfy in order for the parameters of the Jelinski-Moranda linear de-eutrophication model to have finite values. The MLE's can be computed by solving the following equations for \( \hat{\lambda}, \hat{\alpha}, \hat{\beta} \):

\[
\frac{1}{n} \sum_{i=1}^{n} \frac{1}{N - M_{i-1}} - \sum_{i=1}^{n} \hat{\lambda} t_i^\hat{\alpha} = 0;
\]

\[
\frac{n}{\hat{\alpha}} + \sum_{i=1}^{n} \log t_i - \sum_{i=1}^{n} \hat{\phi}(N - M_{i-1}) t_i^\hat{\alpha} \log t_i = 0;
\]

\[
\frac{n}{\hat{\beta}} - \sum_{i=1}^{n} (N - M_{i-1}) t_i^\hat{\beta} = 0
\]

where

- \( M_i \) = number of errors corrected before the \( i \)th failure,
- \( t_i \) = time (CPU) between the \((i-1)\)th and \( i \)th failures,
- \( n \) = number of failures.

These are discussed further in [4].

b) Littlewood's Model: The GPM is a deterministic reliability growth model. This is shown in Fig. 5(a) where after each error correction the residual failure rate (\( \lambda_j \)) decreases by a known amount. We have already noted that different errors have different failure rates. Thus, a better approach is to model the failure rate as a random variable. This results in Fig. 5(b). Here the change in the failure rate is random. Also, the failure rate is constant during the interval between error corrections. However, if one adopts a Bayesian viewpoint, then the failure rate, as perceived by the tester, is a varying quantity even when no changes are made to the software: the fact that there has been no failure makes us more confident in the program. This is shown in Fig. 5(c). This issue is discussed in detail by Littlewood in [56] and [57].

In Littlewood’s model [56] \( f(T_j(s)) = 1 \) (i.e., the testing is assumed to reflect the operational environment) and \( \lambda_j \) is a random variable. The prior distribution of the failure rate \( \phi \) of each error is assumed to be the gamma distribution with parameters \( \alpha \) and \( \beta \). The posterior distribution is computed as follows:

\[
\text{pdf} \{ \phi = l | \text{no failure due to this error during time } t \} = \frac{\text{pdf} \{ \text{no failure due to this error during time } t | \phi = l \} \text{pdf} \{ \phi = l \} }{\int_0^\infty \text{pdf} \{ \text{no failure due to this error during time } t | \phi = l \} \text{pdf} \{ \phi = l \} \, dl}
\]

\[
= \frac{e^{-lt} \beta^\alpha t^{\alpha-1} e^{-\beta t}}{\Gamma(\alpha)} \cdot \frac{\Gamma(\alpha)}{\Gamma(\alpha)} = \frac{(\beta + t)^\alpha t^{\alpha-1} e^{-\beta t}}{\Gamma(\alpha)}.
\]

Thus, the posterior distribution is gamma \((\alpha, \beta + t)\). If \( M_j \) errors have been corrected, then assuming that each error constitutes an independent error process, \( \lambda_j \) has the gamma \(((N-M_j)\alpha, \beta + t)\) distribution where \( N \) is the original number of errors in the software.

Therefore, from (4),

\[
R_f(t) = \int_0^\infty \frac{e^{-lt} (\beta + s_j)^{(N-M_j)\alpha} \gamma((N-M_j)\alpha, \beta + s_j + t)}{\Gamma((N-M_j)\alpha)} \, ds_j
\]

Fig. 5. Modeling of failure rates. (a) Deterministic. (b) Stochastic. (c) Bayesian.
where

\[ s_j = \text{time of the } j\text{th failure}, \]
\[ t = \text{elapsed time since } j\text{th failure}. \]

Thus, \( R_j(t) \) is a Pareto distribution which appears to be more suitable for software systems since it permits very large error free intervals unlike the exponential distribution [60].

**Assumptions:** The assumptions of Littlewood’s model are as follows:

1) consecutive inputs have independent failure probabilities,
2) at any time, the failure rates of the errors remaining in the program are independently identically distributed random variables; in particular, the distribution is assumed to be the gamma distribution,
3) the program failure rate is the sum of the individual failure rates,
4) the input process simulates the operational environment,
5) no new errors are introduced.

**Discussion:** One major weakness is assumptions 2) and 3) taken together. This is not true in general. For example, consider the extreme case where the failure rates of the errors have the constant distribution \( c \), and the errors affect the same elements in the input domain. Then the program failure rate is also \( c \) and the removal of each error, except the last error, leaves the failure rate unchanged.

A safer approach would be to assume that the errors are disjoint. This would become more reasonable as the size of the software increases. However, the analysis is likely to be complicated since we can no longer assume that the errors have independent failure rates. The data required are the CPU time between errors and the number of errors corrected. Additional aspects of the model are discussed in [56].

2) Nonerror-Counting Models: The models discussed in this section do not consider the number of errors remaining in the program. Instead they consider the effects of the remaining errors. This permits a simple modeling of the possibility that new errors are introduced while correcting an error.

a) Jelinski-Moranda Geometric De-Eutrophication Model: This model was first proposed by Moranda [69], [73]. A generalization of the model is

\[ f(T_j(k)) = \lambda_j = \text{a random variable}. \]

Thus, this is a GPM with geometrically decreasing failure rate. It models the case where consecutive errors have decreasing sizes. This is more realistic than the assumption that all errors have the same size (as in the GPM models discussed in Section IV.C.1a). An application of this model is discussed in [69]. An interesting observation is that the estimate of the parameters of this model may exist even in cases where those of the linear de-eutrophication model do not exist, i.e., fail to converge [106], [21].

b) Input Domain-Based Stochastic Model: This model is discussed in detail in [85]. In (3) let

\[ f(T_j(k)) = 1 \]

Thus, the testing process is assumed to be identical to the operational environment. Let \( \Delta_j \) be the size of the \( j \)th error. Then

\[ \Delta_j = \lambda_{j-1} - \lambda_j. \]

Intuitively, errors which are caught later have a smaller size than those which are caught earlier. However, this is true only in a probabilistic sense. This can be modeled by requiring that

\[ \Delta_j \leq \Delta_{j-1} \]

where \( X \leq Y \) means that \( X \) is stochastically smaller than \( Y \). In order to model a variety of situations, we assume that \( \Delta_j \sim \lambda_{j-1}X \), where \( X \sim F \), and \( F \) is assumed to be piecewise continuous. \((X \sim F \text{ means that the random variable } X \text{ has the distribution } F)\)

A special case is \( F = \beta(r, s), r \gg 1, s \gg 1 \). The \( \beta \) (beta) distribution is discussed in [12] and [44] and is often used in the study of hardware and software reliability [109]. Assuming that initially there is an error present for any input (i.e., \( \lambda_0 = 1 \)), we get

\[ E[\lambda_j] = a^j, \quad \text{where } a = \frac{s}{r+s} \]

\[ \text{MTTF}_j = \left[ \frac{r+s-1}{s-1} \right]^j \approx \frac{1}{a^j} \quad \text{for } s \gg 1. \]

We can attempt to predict the value of the constant “\( a \)”.

Thus, we can attempt to predict the value of the constant “\( a \)”.

For example, if we assume that the size of the next error is symmetrically distributed between 0 and \( \lambda_{j-1} \), then \( r = s \), so that \( a = 1/2 \).

Thus, the simple model permits the prediction of the model parameters and the modeling of various debugging situations.

**Assumptions:** The assumptions of the model are as follows:

1) successive inputs have independent failure probability,
2) an assumption is made regarding the distribution of the change in the residual error size after each correction,
3) an assumption is made regarding the initial error size,
4) the testing process is assumed to be the same as the operating environment; however, this can be easily relaxed.

**Discussion:** The key assumption is the second one. It is possible to validate this assumption by estimating the size of each error detected and corrected and assuming that the remaining errors are disjoint from the corrected errors. The size of each error can be estimated using the sampling technique discussed in Section V.

It is difficult to estimate the parameters of \( F \) based on the error history of the program. Attempts to estimate the MLE of the parameters by trial and error can fail if \( k \) is large since then \( (k) \) becomes very large. A practical approach is to compute a series of approximations to \( R_j(k) \). For example, the
first approximation is

\[ P_j(k) = P \left\{ \frac{k \text{ successful runs before failure}}{j \text{ errors detected}} \right\} \]

\[ = E[(1 - \lambda_j)^k \lambda_j] \approx (1 - E[\lambda_j])^k E[\lambda_j] \]

\[ = (1 - \alpha)^k \alpha^l. \]

This is similar to the Jelinski-Moranda geometric de-eutrophication model [69]. The MLE of \( \alpha \) is determined by solving the following equation for \( \hat{\alpha} \):

\[ \sum_{i=1}^{l} \frac{n_i \hat{\alpha}^l}{1 - \hat{\alpha}^l} = \frac{j(j + 1)}{2} \]

where \( n_i \) is the number of successful runs between the \((i-1)\)th and the \(i\)th errors.

**Application:** In this section we discuss the application of this model to the error data derived from the OECD Halden reactor project [21] and the EPRI (Electrical Power Research Institute) project [62]. Both the Halden and the EPRI projects involved research on the development methodology for critical software for nuclear power plant safety control systems. A major problem is the validation of the software and the assessment of its reliability.

Fig. 6 shows the application of the first approximation of the stochastic model to the Halden project data. Fig. 6(a) shows the estimate of the constant \( \alpha \) and the predicted MTTF’s. From a comparison of the predicted and actual MTTF we cannot conclude much regarding the validity of the model. However, we make two important observations. First, the estimate of \( \alpha \) shows rapid convergence, so that by the jackknife technique of Mosteller and Tukey [74] we can conclude that the fit is reasonably good, i.e., additional data do not change the model parameters much. Second, from Fig. 6(b) we see that the error data lie largely within the 90 percent upper and lower confidence bounds. The fit is relatively good considering the large fluctuations in the actual data.

For the EPRI project, two programs PROGA and PROGB were developed based on the same design. (Another program has been developed based on the same requirement specification, but the error data for it is not yet available.) Both the programs were tested with the same set of test cases. However, the testing was not random—functional testing was used. We assume that

\[ f(\text{n} \text{th function}) = n \]

where the \( \text{n} \text{th function} \) denotes the \( \text{n} \text{th distinct function} \) to be tested since the start of testing. This seems valid since the size of errors is magnified if the \( n-1 \) (reasonably) debugged functions are excluded from testing.

We consider the mixed function testing process, i.e., a function is tested a certain number of times, then a transition is made to another function (which may have been already tested), and so on. The MLE of \( \alpha \) is determined by solving the following equation for \( \hat{\alpha} \):

\[ \frac{j(j + 1)}{2} = \sum_{i=1}^{l} \sum_{k=1}^{l} \frac{n_i \text{ int}_k \text{ F}_{ik} \hat{\alpha}^l}{1 - \text{ F}_{ik} \hat{\alpha}^l} \]
where
\[ j = \text{the total number of errors detected}, \]
\[ n_i = \text{the number of transitions from function to function between the detection of the (i - 1)th and the ith errors}, \]
\[ n_{ik} = \text{the number of times the kth function (in the sequence between the detection of the (i - 1)th and the ith errors) is tested}, \]
\[ F_{ik} = n, \text{ if the kth function (in the sequence between the detection of the (i - 1)th and the ith errors) is the n} \text{th distinct function to be tested}. \]

Fig. 7(a) shows the application of the model to PROGA, yielding \( R(1) = 0.99993 \) and \( \text{MTTF}_{\text{P}} = 13903 \). The application of the model to the data for PROGB is shown in Fig. 7(b).

Further discussions of the model and its application to project management appear in [85].

\( \text{c) Littlewood-Verrall Model:} \) This model is discussed in [53]. In (4) let
\[ f(T_j(s)) = 1; \]
\[ \lambda_j = \text{a random variable.} \]

An innovative feature of this model is the direct modeling of the case where the programmer may introduce new errors while correcting an error. This is achieved by assuming that
\[ \lambda_j \leq \lambda_{j-1}. \]

Let \( g(\lambda_j, j) \) be the probability density function (pdf) of \( \lambda_j \). Then
\[ R_f(t) = \int_0^\infty e^{-\lambda_j t} g(\lambda_j, j) d\lambda_j. \]

Let \( h(t|\lambda_j) \) be the pdf of time to failure given \( \lambda_j \). Then
\[ h(t|\lambda_j) = \frac{d}{dt} [R_f(t|\lambda_j)] = \lambda_j e^{-\lambda_j t}. \]

Littlewood and Verrall have adopted the Bayesian approach, i.e., the pdf of \( \lambda_j \) is updated as additional data are gathered. Let \( \alpha \) be the parameters of \( g \); let \( P_0(\alpha) \) be the prior pdf of \( \alpha \) and \( P_1(\alpha) \) be the posterior pdf of \( \alpha \). Then (see [53])
\[ P_1(\alpha) \propto \left\{ \prod_{i=1}^n \int h(t_i|l) g(l, i|\alpha) dl \right\} P_0(\alpha) \]
where \( t_i \) is the length of the \( i \)th interval. Therefore,
\[ R_f(t_{n+1}) = \int_0^\infty e^{-\lambda_j t_{n+1}} \int g(\lambda_j, j|\alpha) P_1(\alpha) d\alpha d\lambda_j. \]

The particular distribution assumed for \( g \) in [53] is the gamma distribution. This can model a variety of different situations.

**Assumptions:** The following are assumed:
1) successive inputs have independent failure probability,
2) the failure rate is stochastically decreasing,
3) the distribution of \( \lambda_j \), denoted by \( g(\lambda_j, j) \) is assumed,
4) the prior distribution of parameters of \( g \) is assumed,
5) the testing process is similar to the operational environment.

**Discussion:** Although the model is moderately general, it has not been widely used. Some applications can be found in [53] and [61]. The model cannot utilize additional information on the relation between \( \lambda_j \) and \( \lambda_{j-1} \). For example, it

\[
\begin{array}{cccccc}
 j & n_j & n_{jf} & f_{jf} & \frac{1}{a} \\
\hline
1 & 1 & 1 & 1 & 2.0000 \\
2 & 1 & 1 & 20 & 1 & 4.0000 \\
3 & 1 & 1 & 20 & 2 & 3.6973 \\
4 & 1 & 1 & 40 & 2 & 3.3828 \\
5 & 2 & 1 & 200 & 1 & 3.4863 \\
6 & 5 & 1 & 200 & 2 & 3.9073 \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
 j & n_j & n_{jf} & f_{jf} & \frac{1}{a} \\
\hline
1 & 1 & 1 & 1 & 2.0000 \\
2 & 1 & 1 & 20 & 1 & 4.0000 \\
3 & 1 & 1 & 20 & 2 & 3.6973 \\
4 & 1 & 1 & 40 & 2 & 3.3828 \\
5 & 2 & 1 & 200 & 1 & 3.4863 \\
6 & 5 & 1 & 200 & 2 & 3.9073 \\
\hline
\end{array}
\]

Fig. 7. (a) Application of the stochastic model to the EPRI project data—PROGA. (b) Application of the stochastic model to the EPRI project data—PROGB.
cannot model the case where it is assured that no new errors are introduced, i.e., \( \lambda_j \leq \lambda_{j+1} \). Also, \( g(\lambda_{j+1}) \) includes a growth parameter which is ad hoc. Further, from the results indicated in [61] it appears as though the distribution of the time to next failure is too broad to yield a satisfactory indication of the current status of the software. A sharp distribution is preferred if the reliability indication is to be used in determining a stopping time for software testing.

D. Summary

We can view \( \lambda \) as a random walk process in the interval \((0, e)\). Each time the program is changed (due to error corrections or other modifications) \( \lambda \) changes. In the formulation of (3) and (4), \( \lambda_j \) denotes the state of \( \lambda \) after the \( j \)th change to the program. Let \( Z_j \) denote the time between failures after the \( j \)th change. \( Z_j \) is a random variable whose distribution depends on \( \lambda_j \). In all the above continuous (discrete) time models, we have assumed this distribution to be the exponential (geometric) distribution with parameter \( \lambda_j \), provided that \( f(T_j(\cdot)) = 1 \). We do not know anything about the random walk process of \( \lambda \) other than a sample of time between failures. Hence, one approach is to construct a model for \( \lambda \) and fit the parameters of the model to the sample data. Then we assume that the future behavior of \( \lambda \) can be predicted from the behavior of the model.

Some of the models for \( \lambda \) which have been developed are as follows.

**General Poisson Model [4]:** The set of possible states are \((0, e/N, 2e/N, \cdots, e)\); \( \lambda_j = (N - j)e/N; \) the parameters are \( e, N \); there is a finite number of possible states.

**Geometric De-Eutrophication Model [69]:** The set of possible states are \((e, ed, ed^2, ed^3, \cdots)\), where \( d < 1; \lambda_j = ed^j \); the parameters are \( e, d \); there is an infinite (although countable) number of states.

**Stochastic (Input Domain) Model:** The state space is continuous over the interval \((0, e)\); \( \lambda_j = \lambda_{j-1} + \Delta_j \), where \( \Delta_j \sim \lambda_j-1 X, X \sim \beta(t, s) \); the parameters are \( r \) and \( s \); note the \( e = 1 \) in (3).

An alternative approach is the Bayesian approach advocated by Littlewood [57]. Here we postulate a prior distribution for each of \( \lambda_1, \lambda_2, \cdots, \lambda_j \). Then based on the sample data, we compute the posterior distribution of \( \lambda_{j+1} \). Some additional discussions appear in [85].

In this section we have discussed some of the theoretical issues of modeling the software reliability growth process. Additional discussions can be found in [50], [54], [57], [67], [70], [71], [97], [103], and [107]. Many other software reliability growth models have been proposed. Some of these are discussed in [21], [37], [89], [92], [93], and [107]. The data requirements of these models have been investigated in [26]. Trivedi and Shooman [108] have developed a model which includes the debugging and error correction time. Reliability growth models have been used for other purposes. Some examples are: code reading [48], estimating the stopping time for testing a program [33], [34], [77], [85], estimating the number of executable paths given that the inputs are selected randomly [72], and estimation of hardware reliability growth [73].

In the following section we discuss models which estimate the reliability of the program based on the results of testing performed in its validation phase. In this phase no changes are made to the software even if new errors are discovered.

V. Validation Phase

As we have already noted, the validity of most software reliability growth models increases with the size of the software since personnel peculiarities are averaged out. However, software used for highly critical, real-time purposes, as in nuclear power plant safety control systems, are of medium size (approximately a few thousand lines of some high level Language code). The programs must be shown to possess very high reliability. Ideally, we would like to prove that the software meets its specification. However, existing proof techniques cannot tackle programs of this size and complexity. For this reason, after the software has been debugged to the satisfaction of the developer(s) it enters a validation phase. During this phase the software is thoroughly tested, specifically for estimating its reliability. If any errors are discovered during this phase, they are not corrected. In fact, the software may be rejected if even a single new error is discovered. In this section we discuss methods of estimating software reliability during the validation phase. We first discuss Nelson's method [63], [79], [109]. Then we develop a model for estimating the reliability and the correctness probability of the program based on its input domain.

A. The Nelson Model

This model [63], [109] is based on the definition given in Section II. Test cases are selected randomly according to the operational distribution. Then

\[
\tilde{R}(1) = 1 - \frac{n_f}{n}
\]

where

\[
n = \text{total number of test cases,} \\
n_f = \text{number of failures out of these } n \text{ runs.}
\]

Fig. 8 shows the size of the remaining errors \( Ve \) in the program when inputs are selected according to the operational distribution. \( Ve \) is an unknown quantity which is related to the reliability \( R(1) \). Specifically

\[
Ve = 1 - \tilde{R}(1).
\]

In Section IV-C we observed that if inputs are selected independently from the input domain, then

number of runs to next failure \( \sim \text{geometric}(Ve) \).

Hence, if \( n_f \) is the number of failures out of \( n \) runs, the MLE of \( Ve \) is

\[
\hat{Ve} = \frac{n_f}{n}.
\]

This is true since no changes are made to the program even if errors are detected, so that \( Ve \) remains constant. This is the difference with the models discussed in Section IV which consider the case where \( Ve \) changes due to debugging actions.
The Nelson model is the only model whose theoretical foundations are sound. However, it suffers from a number of practical drawbacks.

1) In order to have a high confidence in the reliability estimate, a large number of test cases must be used.

2) It does not take into account "continuity" in the input domain. For example, if the program is correct for a test case, then it is likely that it is correct for all test cases executing the same sequence of statements.

3) It assumes random sampling of the input domain. Thus, it cannot take advantage of testing strategies which have a higher probability of detecting errors, e.g., boundary value testing, etc. Further, for most real-time control systems the successive inputs are correlated if the inputs are sensor readings of physical quantities, like temperature, which cannot change rapidly. In these cases we cannot perform random testing.

4) It does not consider any complexity measure of the program, e.g., number of paths, statements, etc. Generally, a complex program should be tested more that a simple program for the same confidence in the reliability estimate.

In order to overcome these drawbacks, the model has been extended [79] as follows.

The input domain is divided into several equivalence classes. The division can be based on paths or some other criteria when the number of paths is too large (e.g., program subfunctions). It is assumed that there is some continuity over an equivalence class, i.e., if the program executes correctly for an input from the jth equivalence class, then it will execute correctly for any randomly selected input from the same equivalence class with probability 1 - \( \epsilon_j \), where \( \epsilon_j < 1 \). Then

\[
R(1) = \sum_{j=1}^{m} P_j (1 - \epsilon_j)
\]

where

- \( m \) = number of equivalence classes,
- \( P_j \) = probability of selecting an input from the jth equivalence class during actual operation.

**Discussion:** This model is a big improvement over the original form. Some comments are as follows.

1) The assignment of values to \( \epsilon_j \) is ad hoc; no theoretical justification is given for the assignment [79].

2) The model uses only one type of complexity measure, namely, number of paths, functions, etc. However, it does not consider the relative complexity of each path, function, etc.

Many other interesting aspects of the Nelson model are discussed in [109].

**B. Input Domain Based Model**

This model is discussed in detail in [82]. It removes most of the objections to the Nelson model. The price is the increased complexity of the model. However, it is perfectly suited for medium size programs such as those for real-time control systems.

In Section II we defined the operational reliability of a program by the following equation:

\[
R = 1 - \lim_{n \to \infty} \frac{n_f}{n}
\]

where \( n_f \) is the number of failures in \( n \) runs. This is the time domain definition of software reliability. In Section IV we discussed the concept of error size and found that

\[
\hat{R} = 1 - \hat{V}_e,
\]

where \( \hat{V}_e \) is the estimated remaining error size. This is the input domain definition of software reliability. \( \hat{V}_e \) can be determined by testing the program and locating and estimating the size of errors found. An obvious advantage of this approach is that it permits any testing strategy to be used. An accompanying parameter is the correctness probability of the program. This requires the concept of probabilistic equivalence classes defined as follows: \( E \) is a probabilistic equivalence class if \( E \subseteq I \), where \( I \) is the input domain of the program \( P \), and \( P \) is correct for all elements in \( E \), with probability \( P(X_1, \cdots, X_d) \), if \( P \) is correct for each \( X_i \in E, i = 1, \cdots, d \). Then \( P(I \mid X) \) is the correctness probability of \( P \) based on the set of test cases \( X \). (Obviously, the program must be correct for each element in \( X \).) Probabilistic equivalence classes are derived from the requirements specification and the program source code in order to minimize control flow errors. A suggested selection criterion [82] is as follows.

Let \( E \) be a probabilistic equivalence class; \( X \in E \) if an error in the program which affects any element in \( E \) can affect \( X \), and vice versa. The results of this classification scheme are as follows:

1) it includes all paths without loops since distinct paths differ in at least one statement,

2) multiple conditions are treated separately since an error in one condition need not affect the other conditions,

3) loops are restricted to a finite number of repetitions.
In order to further minimize control flow errors, these classes should be intersected with classes derived from the requirements specification [112].

Finally, we can estimate the correctness probability of the program using the continuity assumption, i.e., closely related points in the input domain are "correlated" with respect to the implementation of a function. This is true in general for algebraic programs where errors usually affect an interval of nearby points. These regions correspond to high probability equivalence classes, such as those formed on the basis of program paths.

At first we assume that the program has a single input variable. We further assume that the correctness probability of a point depends at most on the correctness of its neighbors. If the equivalence class is discrete, say \( (x_1, x_2, \ldots, x_n) \), then this means that

\[
P\{x_i \text{ correct} | x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \text{ correct} \}
= P\{x_i \text{ correct} | x_{i-1}, x_{i+1} \text{ correct} \}.
\]

This leads to a mathematically tractable derivation of the correctness probability.

Let \( E_i = [a, a + \lambda] \) denote a continuous equivalence class of a program \( P \), selected using the above selection criterion. Fig. 9 shows the interpretation of the nearest neighbor dependency assumption for \( E_i \). Fig. 9(a) illustrates the case where a single test case is used. It means that the distribution of a probabilistic equivalence class containing the test case is the exponential distribution. In principle, any other distribution can be considered. However, the derivation of the correctness probability may be intractable. Fig. 9(b) considers the case where two test cases are used. The region between these two points is validated by both of them.

In general, it can be shown that

\[
P\{E_i \text{ is correct} | 1 \text{ test case} \} = e^{-\lambda V}
\]

\[
P\{E_i \text{ is correct} | n \text{ test cases having successive distances } x_j, j = 1, \ldots, n-1 \} = e^{-\lambda V} \prod_{j=1}^{n-1} \left[ \frac{2}{1 + e^{-\lambda x_j}} \right].
\]

Here \( \lambda \) is a parameter of the equivalence class. In particular, the mean length of \( E_i \) is \( 1/\lambda \). \( \lambda \) can range from 0 to \( \infty \). A value of 0 means that \( E_i \) is an equivalence class with finite degree. (The degree of an equivalence class is the number of distinct test cases which completely validate it.) For example, \( f(x) = \text{constant} \) can be verified by one test case.

The above analysis can be easily generalized to \( m \) dimensions by assuming independent behavior along each coordinate. However, for equivalence classes of finite degrees we can adopt a more satisfactory approach. If \( D \) is the degree of the equivalence class then it can be shown that

\[
\lambda V \approx \frac{D - 1}{N}
\]

where \( N \) is the number of elements in the class (due to the finite word length of the computer used). This means that more testing should be performed when using computers of small word lengths, which is intuitive.

**Fig. 9. Interpretation for continuous equivalence classes.** (a) Single test case. (b) Two test cases.

\( D \) can be related to the testing complexity of the program. For example, for a program computing a polynomial

\[
a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n.
\]

\( D \) is \( (n + 1) \) since \( (n + 1) \) distinct test cases completely validate it. More generally, for algebraic programs we can estimate \( D \) by viewing each expression as a group of multinomials and function calls. Thus

\[
x^3 z + y \cos \left( x^2 y + \frac{z^w}{z} \right)
\]

can be written as

\[
a_1 = xw
\]

\[
a_2 = z
\]

\[
a_3 = \frac{a_1}{a_2}
\]

\[
a_4 = x^2 y + a_3
\]

\[
a_5 = \cos (a_4)
\]

\[
a_6 = x^3 z + y a_5.
\]

Each subexpression is either a multinomial or a function call. (Division and exponential are considered as functions with two input parameters.) For a function call with \( n \) input parameters, the complexity is defined to be \( n \) + the complexity of the function. If the function has been validated, its complexity is 1. For a multinomial in \( k \) variables having highest degree \( n \), the complexity is defined to be

\[
\prod_{i=1}^{k} \left[ \frac{n + i}{i} \right].
\]

This is equal to the maximum number of terms in a multinomial in \( k \) variables having highest degree \( n \). When \( k \) is large, this complexity measure will be very large. In these cases the complexity measure can be taken to be equal to the number of terms in the expression. This is acceptable since code reading would detect any extra or missing terms. Thus, in the above example, the complexity is \( 6 + 2 + 3 + 20 + 2 + 70 = 103 \). For such general expression, we estimate \( D \) by equating it to the
testing complexity of the equivalence class. Thus
\[ \lambda V \approx \frac{D - 1}{N}. \]

Assumption: The distribution of the equivalence classes has to be assumed; this includes prior estimation of the parameters of the distribution.

Discussion: The advantages of this model are as follows.
1) Any test case selection strategy can be used; this will minimize the testing effort since we can choose test cases which exercise error-prone constructs.
2) It does not assume random sampling.
3) It takes into account the complexity of the program: a simple program is tested less than a complicated program for the same correctness probability. The model also yields the optimal testing strategy to be used. Specifically, for algebraic programs the test cases should be spread out over the input domain for higher correctness probability.

The disadvantages of the model are as follows.
1) It is relatively expensive to determine the equivalence classes and their complexity.
2) The assumption has to be justified by experience.

C. Summary

The models discussed in this section are especially attractive for medium size programs whose reliability cannot be accurately estimated by using reliability growth models. These models also have the advantage of considering the structure of the program. This enables the joint use of program proving and testing in order to validate the program and assess its reliability [62]. Another model which considers the program structure is discussed in [100].

The correctness probability measure discussed above is an example of criteria which determine how well the program has been tested. A similar criterion based on fuzzy set theory is discussed in [7].

As evident from the discussions in Section III, no specific software reliability model has found wide acceptance. This is partly due to the cost involved in gathering failure data and partly because of the difficulty in modeling the testing process. In the following, we outline a method combining well-established proof procedures with software reliability estimation methods. It is particularly suitable for critical control systems.

1) During the testing and debugging phase two different software reliability growth models are used, primarily for determining the stopping time. This is a measure of the amount of testing needed in order to reach a desired reliability goal. The latter is stringent since software for critical systems may be rejected if even a single error is found after this phase.

2) During the validation phase the equivalence classes are determined based on program paths using the selection criterion discussed in Section V-B. Boundary value and range testing are performed in order to ensure that the classes are properly chosen.

3) If the path corresponding to each equivalence class can be verified (e.g., by using symbolic execution [83]), then the correctness probability of the class is 1.

4) If the correctness of the path cannot be verified, then the degree of the equivalence class is estimated. Next, as many test cases as necessary are used in order to achieve a desired correctness probability estimate. It can be shown that for a given number of test cases, the correctness probability estimate is maximized if the test cases are spread evenly over the equivalence class [82].

In the next section we discuss an experimental approach in determining the correctness probability of the program. This can be used as a verification of the theoretical estimate.

VI. Error Seeding

The method of "error seeding" is a powerful experimental approach to evaluating software development tools and techniques. It was originally used by Mills [36], [92] to estimate the number of errors remaining in the program. In Section VI-A we show that error seeding provides an independent estimate of the correctness probability of a program. Section VI-B discusses its application to evaluating the efficiency of different testing techniques and the reliability of different sets of test cases.

A. Correctness Probability

The program is seeded with \( m \) errors (one at a time), and for each error all the test cases are run until the error is detected or the set of test cases is exhausted. Then

\[ P\{ \text{correctness probability based on input set } X \} \]
\[ = P\{I|X\} \]
\[ = P\{\text{no error in program|program works for } X\} \]
\[ = P\{\text{any error in the program is detected by } X\} \]
\[ = \frac{m_d}{m} \]

where \( m_d \) is the number of seeded errors detected by the set of test cases \( X \). The size of the errors seeded in the program provides a natural confidence level in the resulting estimate of the correctness probability.

The error seeding approach is not always practical since the program may have to be tested \( m|X| \) times and this can be very large. For example, if \( m = 1000, |X| = 10^6 \), then the program may have to be tested \( 10^8 \) times! Thus, a model for estimating the correctness probability—such as developed in Section V-B—is essential. Another factor in favor of the theoretical model is the auxiliary derivations regarding optimal testing, dependency of the correctness probability on the program complexity, machine word length, etc. [82]. These results are reasonable. In particular, the spread of test cases over the input domain has been suggested as a good testing practice by Howden and is used as a criteria of how well the program has been tested by Ehrenberger [14].

B. Evaluation of Test Cases

An important application of error seeding is to the evaluation of different testing strategies and sets of test cases. We first discuss the efficiency of a testing strategy and then we discuss the reliability of test cases.

1) Testing Efficiency: The efficiency of a test case selec-
tion strategy is defined as the probability that test cases selected randomly using the strategy will detect an error, i.e., it is the effective error size viewed by the testing strategy. The efficiency of a test case depends on various factors, especially the number of errors remaining in the program and the type of the errors. If we consider the relative efficiency of different testing strategies, then the primary factor is the type of errors in the program. For example, initially static analysis techniques can detect many errors, but their efficiency falls as syntax errors, missing initialization, and other similar errors are eliminated. Similarly, some errors are difficult to detect by testing or dynamic analysis, although they are easily detected by code review. An example is missing control flow type of errors which are extremely difficult to detect by testing the code, since the missing code may affect a very small part of the input domain, i.e., have a very small error size. However, code review and static analysis can expose these errors. Further studies on the nature and frequency of software errors can be found in [2], [27], [90], and [99].

The efficiency of a testing strategy can be estimated by error seeding. We seed errors in the program according to various types and sizes and then apply different testing strategies. The results indicate the efficiency of the strategies and the types of errors for which they are mainly effective.

Experiment: The following experiment was performed in the EPRI project [62]. Sixteen errors were seeded in the program. One was caught during code review. Eight were caught using data flow analysis tools. The errors which escaped detection by these techniques were of the type of missing assignment statement and bad expressions. These were detected by testing.

2) Reliability of Test Cases: A data selection criterion was originally defined by Goodenough and Gerhart [38] to be reliable if it ensured the selection of tests that are consistent in their ability to reveal errors. Here we define the reliability of a set of test cases, irrespective of the test selection strategy, to be some measure of the confidence in the correctness of the program if it works for the given test cases. That is, the probability of test cases is a measure of our belief in the a posteriori correctness of the software. The correctness probability (Section IV-B) and correctness possibility [7] are measures of the reliability of test cases. Other measures are: 1) the coverage of program structure, its functions, and its input domain [14]; these are due to Ehrenberger; 2) representativeness of the set of test cases [15]; and 3) statistical confidence in the reliability estimate [109]. Also, software complexity measures can be used to guide the amount of testing [17], [29], [40], [64].

An interesting experimental approach is program mutation due to DeMillo and Lipton [22]. However, this technique is expensive since the number of mutations is combinatorially explosive for programs of realistic sizes. The majority of the mutations have large error sizes and these are easily detected. A practical solution is to seed the program with errors, such that the size of the errors is controlled. We now discuss one such experiment.

Experiment: The error seeding technique was used to assess the reliability of the set of test cases used in the first phase of the EPRI project [62]. Eight errors were seeded. Errors in expressions were detected by almost all the test cases, implying that arithmetic errors usually have a large size. However, three errors escaped detection. These were of the boundary value and missing control flow type of errors. This indicates that further test cases exercising the ranges of the variables and boundary conditions are necessary.

 VII. Conclusion

We have addressed only a few topics in software reliability. The bulk of this paper is devoted to software reliability growth models. We have developed a general framework for these models using the concepts of error size and testing process. We distinguished between error-counting and nonerror-counting models. If only the reliability estimate is required, then the nonerror-counting models are preferable since it is easy to extend them to include situations where new errors are introduced into the software as a result of changes. Error-counting models should be used when an estimate of the number of remaining errors is needed. This may be required if resources have to be allocated for the maintenance phase (assuming that the average resource per error correction is known). It is also possible to estimate the number of errors remaining by using error seeding techniques. (Another approach is to develop a random walk model for the number of errors remaining in the program.)

Finally, we briefly discussed the Nelson model and its extension, and the input domain based model for the validation phase. We also mentioned the error seeding technique in evaluating testing techniques, sets of test cases, and the correctness probability of programs.

Some aspects which we have not discussed here are: 1) operational reliability [18], [51], [55], [58], 2) reliability estimate after perturbation during the maintenance phase (discussion on maintenance (growth dynamics) can be found in [8], [9], [10], [30], [52], [104], and [105]), 3) the combination of hardware and software reliability estimates to get the overall system reliability estimate [16], [49], [50], [107], the nature and frequency of software errors [2], [27], [90], [99], 4) methods of achieving reliable software: there are several references on this aspect, some of which are [3], [5], [11], [28], [32], [35], [39], [42], [43], [45], [65], [80], [81], [83], [87], and [88], 5) other metrics, besides software reliability, which reflect the quality of software systems are discussed in [13], [20], [110], and [111], 6) finally, there have been a few case studies on the efficacy of these techniques [1], [21], [46], [76], [113].

Acknowledgment

The authors wish to thank B. Littlewood, E. Nelson, G. Schick, and R. Wolverton for their careful review of an earlier draft of the paper. Their detailed comments have greatly improved the quality of the paper.

References


[2] F. Akiiyama, "An example of software system debugging," in


[59] ——, private communication.


[70] ——, “The (sad) status of: (unapproached) limits to; and (manifolds) alternatives of software measurement techniques,” in Proc. COMPCON 1978, San Francisco, CA, Mar. 1978, pp. 353-354.


Weak Mutation Testing and Completeness of Test Sets

WILLIAM E. HOWDEN

Abstract—Different approaches to the generation of test data are described. Error-based approaches depend on the definition of classes of commonly occurring program errors. They generate tests which are specifically designed to determine if particular classes of errors occur in a program. An error-based method called weak mutation testing is described. In this method, tests are constructed which are guaranteed to force program statements which contain certain classes of errors to act incorrectly during the execution of the program over those tests. The method is systematic, and a tool can be built to help the user apply the method. It is extensible in the sense that it can be extended to cover additional classes of errors. Its relationship to other software testing methods is discussed. Examples are included.

Different approaches to testing involve different concepts of the adequacy or completeness of a set of tests. A formalism for characterizing the completeness of test sets that are generated by error-based methods such as weak mutation testing as well as the test sets generated by other testing methods is introduced. Error-based, functional, and structural testing emphasize different approaches to the test data generation problem. The formalism which is introduced in the paper can be used to describe their common basis and their differences.

Index Terms—Complete, effective, mutations, testing.

INTRODUCTION

In functional testing, the programmer identifies the functions which are supposed to be implemented by his program, and then tests the code against the specifications for those functions [1]-[3]. The functions to be tested may be described in either requirements or design specifications [4]. Design functions may correspond to parts of a program. Guidelines have been developed both for the identification of and for the construction of data for testing functions.

Manuscript received May 1, 1980; revised June 19, 1981. This paper is a revised version of the paper “Completeness Criteria for Testing Elementary Program Functions,” presented at the 5th International Conference on Software Engineering, 1981.

The author is with the Department of Electrical Engineering and Computer Science, University of California at San Diego, La Jolla, CA 92037.

C. V. Ramamoorthy (M’57–SM’76–F’78) received the undergraduate degrees in physics and technology from the University of Madras, Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, MA.

He was associated with Honeywell’s Electronic Data Processing Division from 1956 to 1971, last as Senior Staff Scientist. He was a Professor in the Department of Electrical Engineering and Computer Sciences at the University of Texas, Austin. Currently, he is a Professor in the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

Dr. Ramamoorthy was Chairman of the Education Committee of the IEEE Computer Society and Chairman of the Committee to develop the Accreditation Guidelines for Computer Science and Engineering Degree Programs. He also was the Chairman of the AFIPS Education Committee, a member of the Science and Technology Advisory Group of the U.S. Air Force, and a member of the Technology Advisory Panel of Ballistic Missile Defense (U.S. Army). Currently, he is Vice President of the IEEE Computer Society for Educational Activities.

Farokh B. Bastani (M’82) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Bombay, India, in 1977, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California, Berkeley, in 1978 and 1980, respectively.

He joined the University of Houston, Houston, TX, in 1980, where he is currently Assistant Professor of Computer Science. His research interests include developing a design methodology and quality assessment techniques for large-scale computer systems.