Software Reliability Models: Assumptions, Limitations, and Applicability

AMRIT L. GOEL, MEMBER, IEEE

Abstract—**A number of analytical models have been proposed during the past 15 years for assessing the reliability of a software system. In this paper we present an overview of the key modeling approaches, provide a critical analysis of the underlying assumptions, and assess the limitations and applicability of these models during the software development cycle. We also propose a step-by-step procedure for fitting a model and illustrate it via an analysis of failure data from a medium-sized real-time command and control software system.**

Index Terms—**Estimation, failure count models, fault seeding, input domain models, model fitting, NHPP, software reliability, times between failures.**

Introduction and Background

An important quality attribute of a computer system is the degree to which it can be relied upon to perform its intended function. Evaluation, prediction, and improvement of this attribute have been of concern to designers and users of computers from the early days of their evolution. Until the late 1960's, attention was almost solely on the hardware related performance of the system. In the early 1970's, software also became a matter of concern, primarily due to a continuing increase in the cost of software relative to hardware, in both the development and the operational phases of the system.

Software is essentially an instrument for transforming a discrete set of inputs into a discrete set of outputs. It comprises of a set of coded statements whose function may be to evaluate an expression and store the result in a temporary or permanent location, decide which statement to execute next, or to perform input/output operations.

Since, to a large extent, software is produced by humans, the finished product is often imperfect. It is imperfect in the sense that a discrepancy exists between what the software can do versus what the user or the computing environment wants it to do. The computing environment refers to the physical machine, operating system, compiler and translator, utilities, etc. These discrepancies are what we call software faults. Basically, software faults can be attributed to an ignorance of the user requirements, ignorance of the rules of the computing environment, and to poor communication of software requirements between the user and the programmer or poor documentation of the software by the programmer. Even if we know that software contains faults, we generally do not know their exact identity.

Currently, there are two approaches available for indicating the existence of software faults, viz. program proving, and program testing. Program proving is formal and mathematical while program testing is more practical and heuristic. The approach taken in program proving is to construct a finite sequence of logical statements ending in the statement, usually the output specification statement, to be proved. Each of the logical statements is an axiom or is a statement derived from earlier statements by the application of an inference rule. Program proving by using inference rules is known as the inductive assertion method. This method was mainly advocated by Floyd, Hoare, Dijkstra, and recently Reynolds [39]. Other work on program proving is on the symbolic execution method. This method is the basis of some automatic program verifiers. Despite the formalism and mathematical exactness of program proving, it is still an imperfect tool for verifying program correctness. Gerhart and Yelowitz [10] showed several programs which were proved to be correct but still contained faults. However, the faults were due to failures in defining what exactly to prove and were not failures of the machanics of the proof itself.

Program testing is the symbolic or physical execution of a set of test cases with the intent of exposing embedded faults in the program. Like program proving, program testing remains an imperfect tool for assuring program correctness. A given testing strategy may be good for exposing certain kinds of faults but not for all possible kinds of faults in a program. An advantage of testing is that it can provide useful information about a program's actual behavior in its intended computing environment, while proving is limited to conclusions about the program's behavior in a postulated environment.

In practice neither proving nor testing can guarantee complete confidence in the correctness of a program. Each has its advantages and limitations and should not be viewed as competing tools. They are, in fact, complementary methods for decreasing the likelihood of program failure.

Due to the imperfectness of these approaches in assuring a correct program, a metric is needed which reflects the degree of program correctness and which can be used

Manuscript received February 4, 1985; revised July 31, 1985 and September 30, 1985. This work was supported in part by Rome Air Development Center, GAFB, and by the Computer Applications and Software Engineering (CASE) Center at Syracuse University.

The author is with the Department of Electrical and Computer Engineering and the School of Computer and Information Science, Syracuse University, Syracuse, NY 13244.

0098-5589/85/1200-1411$01.00 © 1985 IEEE

in planning and controlling additional resources needed for enhancing software quality. One such quantifiable metric of quality that is commonly used in software engineering practice is software reliability. This measure has attracted considerable attention during the last 15 years and continues to be employed as a useful metric. A commonly used approach for measuring software reliability is via an analytical model whose parameters are generally estimated from available data on software failures. Reliability and other relevant measures are then computed from the fitted model.

Even though such models have been in use for some time, the realism of many of the underlying assumptions and the applicability of these models for assessing software reliability continue to be questioned. It is the purpose of this paper to evaluate the current state-of-the-art related to this issue. Specifically, the key modeling approaches are briefly discussed and a critical analysis of their underlying assumptions, limitations, and applicability during the software development cycle is presented.

It should be pointed out that the emphasis of this paper is on software reliability modeling approaches and several related but important issues are only briefly mentioned. Examples of such issues are the practical and theoretical difficulties of parametric estimation, statistical properties of estimators, unification of models via generalized formulations or via, say, a Bayesian interpretation, validation and comparison of models, and determination of optimum release time. For a discussion of these issues, the reader is referred to Goel [19].

The term software reliability is discussed in Section II along with a classification of the various modeling approaches. The key models are briefly described in Sections III, IV, and V. An assessment of the main assumptions underlying the models is presented in Section VI and the applicability of these models during the software development cycle is discussed in Section VII. A step-by-step procedure for fitting a model is given in Section VIII and is illustrated via an analysis of software failure data from a medium-sized command and control system. A summary of some related work and concluding remarks are presented in Section IX.

II. Meaning and Measurement of Software Reliability

There are a number of views as to what software reliability is and how it should be quantified. Some people believe that this measure should be binary in nature so that an imperfect program would have zero reliability while a perfect one would have a reliability value of one. This view parallels that of program proving whereby the program is either correct or incorrect. Others, however, feel that software reliability should be defined as the relative frequency of the times that the program works as intended by the user. This view is similar to that taken in testing where a percentage of the successful cases is used as a measure of program quality.

According to the latter viewpoint, software reliability is a probabilistic measure and can be defined as the probability that software faults do not cause a failure during a specified exposure period in a specified use environment. The probabilistic nature of this measure is due to the uncertainty in the usage of the various software functions and the specified exposure period here may mean a single run, a number of runs, or time expressed in calendar or execution time units. To illustrate this view of software reliability, suppose that a user executes a software product several times according to its usage profile and finds that the results are acceptable 95 percent of the time. Then the software is said to be 95 percent reliable for that user.

A more precise definition of software reliability which captures the points mentioned above is as follows [30]. Let F be a class of faults, defined arbitrarily, and T be a measure of relevant time, the units of which are dictated by the application at hand. Then the reliability of the software package with respect to the class of faults F and with respect to the metric T, is the probability that no fault of the class occurs during the execution of the program for a prespecified period of relevant time.

Assuming that software reliability can somehow be measured, a logical question is what purpose does it serve. Software reliability is a useful measure in planning and controlling resources during the development process so that high quality software can be developed. It is also a useful measure for giving the user confidence about software correctness. Planning and controlling the testing resources via the software reliability measure can be done by balancing the additional cost of testing and the corresponding improvement in software reliability. As more and more faults are exposed by the testing and verification process, the additional cost of exposing the remaining faults generally rises very quickly. Thus, there is a point beyond which continuation of testing to further improve the quality of software can be justified only if such improvement is cost effective. An objective measure like software reliability can be used to study such a tradeoff.

Current approaches for measuring software reliability basically parallel those used for hardware reliability assessment with appropriate modifications to account for the inherent differences between software and hardware. For example, hardware exhibits mixtures of decreasing and increasing failure rates. The decreasing failure rate is seen due to the fact that, as test or use time on the hardware system accumulates, failures, most likely due to design errors, are encountered and their causes are fixed. The increasing failure rate is primarily due to hardware component wearout or aging. There is no such thing as wearout in software. It is true that software may become obsolete because of changes in the user and computing environment, but once we modify the software to reflect these changes, we no longer talk of the same software but of an enhanced or a modified version. Like hardware, software exhibits a decreasing failure rate (improvement in quality) as the usage time on the system accumulates and faults, say, due to design and coding, are fixed. It should also be noted that an assessed value of the software

reliability measure is always relative to a given use environment. Two users exercising two different sets of paths in the same software are likely to have different values of software reliability.

A number of analytical models have been proposed to address the problem of software reliability measurement. These approaches are based mainly on the failure history of software and can be classified according to the nature of the failure process studied as indicated below.

Times Between Failures Models: In this class of models the process under study is the time between failures. The most common approach is to assume that the time between, say, the $(i-1)$st and the ith failures, follows a distribution whose parameters depend on the number of faults remaining in the program during this interval. Estimates of the prarameters are obtained from the observed values of times between failures and estimates of software reliability, mean time to next failure, etc., are then obtained from the fitted model. Another approach is to treat the failure times as realizations of a stochastic process and use an appropriate time-series model to describe the underlying failure process.

Failure Count Models: The interest of this class of models is in the number of faults or failures in specified time intervals rather than in times between failures. The failure counts are assumed to follow a known stochastic process with a time dependent discrete or continuous failure rate. Parameters of the failure rate can be estimated from the observed values of failure counts or from failure times. Estimates of software reliability, mean tme to next failure, etc., can again be obtained from the relevant equations.

Fault Seeding Models: The basic approach in this class of models is to "seed" a known number of faults in a program which is assumed to have an unknown number of indigenous faults. The program is tested and the observed number of seeded and indigenous faults are counted. From these, an estimate of the fault content of the program prior to seeding is obtained and used to assess software reliability and other relevant measures.

Input Domain Based Models: The basic approach taken here is to generate a set of test cases from an input distribution which is assumed to be representative of the operational usage of the program. Because of the difficulty in obtaining this distribution, the input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate of program reliability is obtained from the failures observed during physical or symbolic execution of the test cases sampled from the input domain.

III. Times Between Failures Models

This is one of the earliest classes of models proposed for software reliability assessment. When interest is in modeling times between failures, it is expected that the successive failure times will get longer as faults are removed from the software system. For a given set of observed values, this may not be exactly so due to the fact that failure times are random variables and observed values are subject to statistical fluctuations.

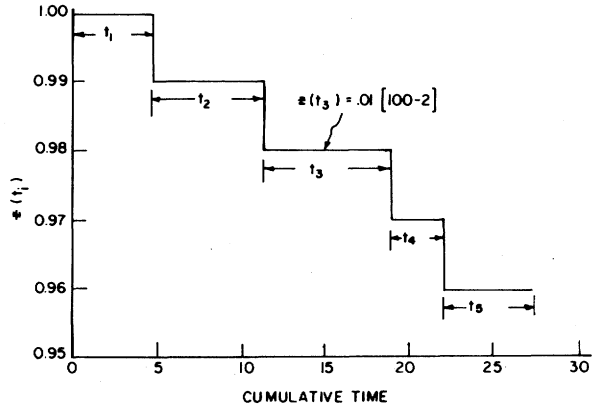

Fig. 1. A typical plot of $Z(t_i)$ for the JM model ($N = 100$, $\phi = 0.01$).

A number of models have been proposed to describe such failures. Let a random variable T_i denote the time between the $(i-1)$st and the ith failures. Basically, the models assume that T_i follows a known distribution whose parameters depend on the number of faults remaining in the system after the $(i-1)$st failure. The assumed distribution is supposed to reflect the improvement in software quality as faults are detected and removed from the system. The key models in this class are described below.

Jelinski and Moranda (JM) De-Eutrophication Model

This is one of the earliest and probably the most commonly used model for assessing software reliability [20]. It assumes that there are N software faults at the start of testing, each is independent of others and is equally likely to cause a failure during testing. A detected fault is removed with certainty in a negligible time and no new faults are introduced during the debugging process. The software failure rate, or the hazard function, at any time is assumed to be proportional to the current fault content of the program. In other words, the hazard function during t_i, the time between the $(i-1)$st and ith failures, is given by

$$Z(t_i) = \phi[N - (i-1)],$$

where ϕ is a proportionality constant. Note that this hazard function is constant between failures but decreases in steps of size ϕ following the removal of each fault. A typical plot of the hazard function for $N = 100$ and $\phi = 0.01$ is shown in Fig. 1.

A variation of the above model was proposed by Moranda [29] to describe testing situations where faults are not removed until the occurrence of a fatal one at which time the accumulated group of faults is removed. In such a situation, the hazard function after a restart can be assumed to be a fraction of the rate which attained when the system crashed. For this model, called the geometric de-eutrophication model, the hazard function during the ith testing interval is given by

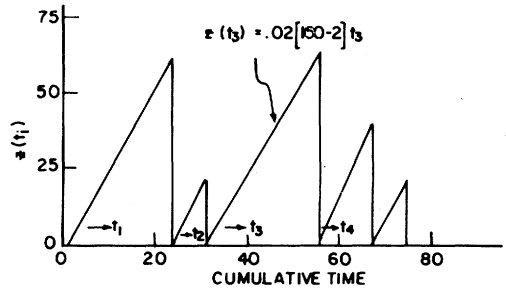

Fig. 2. A typical plot of the hazard function for the SW model ($N = 150$, $\phi = 0.02$).

$$Z(t_i) = Dk^{i-1},$$

where D is the fault detection rate during the first interval and k is a constant ($0 < k < 1$).

Schick and Wolverton (SW) Model

This model is based on the same assumptions as the JM model except that the hazard function is assumed to be proportional to the current fault content of the program as well as to the time elapsed since the last failure [40] is given by

$$Z(t_i) = \phi\{(N - (i - 1))\}\, t_i$$

where the various quantities are as defined above. Note that in some papers t_i has been taken to be the cumulative time from the beginning of testing. That interpretation of t_i seems to be inconsistent with the interpretation in the original paper, see, e.g., Goel [15].

We note that the above hazard rate is linear with time within each failure interval, returns to zero at the occurrence of a failure and increases linearly again but at a reduced slope, the decrease in slope being proportional to ϕ. A typical behavior of $Z(t_i)$ for $N = 150$ and $\phi = 0.02$ is shown in Fig. 2.

A modification of the above model was proposed in [41] whereby the hazard function is assumed to be parabolic in test time and is given by

$$Z(t_i) = \phi[N - (i - 1)]\,(-at_i^2 + bt_i + c)$$

where a, b, c are constants and the other quantities are as defined above. This function consists of two components. The first is basically the hazard function of the JM model and the superimposition of the second term indicates that the likelihood of a failure occurring increases rapidly as the test time accumulates within a testing interval. At failure times ($t_i = 0$), the hazard function is proportional to that of the JM model.

Goel and Okumoto Imperfect Debugging Model

The above models assume that the faults are removed with certainty when detected. However, in practice [47] that is not always the case. To overcome this limitation, Goel and Okumoto [11], [13] proposed an imperfect debugging model which is basically an extension of the JM model. In this model, the number of faults in the system at time t, $X(t)$, is treated as a Markov process whose transition probabilities are governed by the probability of imperfect debugging. Times between the transitions of $X(t)$ are taken to be exponentially distributed with rates dependent on the current fault content of the system. The hazard function during the interval between the $(i - 1)$st and the ith failures is given by

$$Z(t_i) = [N - p(i - i)]\lambda.$$

where N is the initial fault content of the system, p is the probability of imperfect debugging, and λ is the failure rate per fault.

Littlewood–Verrall Bayesian Model

Littlewood and Verall [25], [26] took a different approach to the development of a model for times between failures. They argued that software reliability should not be specified in terms of the number of errors in the program. Specifically, in their model, the times between failures are assumed to follow an exponential distribution but the parameter of this distribution is treated as a random variable with a gamma distribution, viz.

$$f(t_i|\lambda_i) = \lambda_i\, e^{-\lambda_i t_i}$$

and

$$f(\lambda_i|\alpha,\, \psi(i)) = \frac{[\psi(i)]^{\alpha}\, \lambda_i^{\alpha-1}\, e^{-\psi(i)\lambda_i}}{\Gamma\alpha}$$

In the above, $\psi(i)$ describes the quality of the programmer and the difficulty of the programming task. It is claimed that the failure phenomena in different environments can be explained by this model by taking different forms for the parameter $\psi(i)$.

IV. Fault Count Models

This class of models is concerned with modeling the number of failures seen or faults detected in given testing intervals. As faults are removed from the system, it is expected that the observed number of failures per unit time will decrease. If this is so, then the cumulative number of failures versus time curve will eventually level off. Note that time here can be calander time, CPU time, number of test cases run or some other relevant metric. In this setup, the time intervals may be fixed *a priori* and the observed number of failures in each interval is treated as a random variable.

Several models have been suggested to describe such failure phenomena. The basic idea behind most of these models is that of a Poisson distribution whose parameter takes different forms for different models. It should be noted that Poisson distribution has been found to be an excellent model in many fields of application where interest is in the number of occurrences.

One of the earliest models in this category was proposed by Shooman [43]. Taking a somewhat similar approach, Musa [31] later proposed another failure count model based on execution time. Schneidewind [42] took a differ-

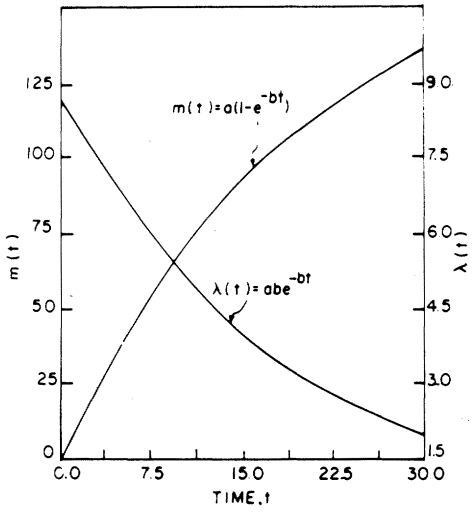

Fig. 3. A typical plot of the $m(t)$ and $\lambda(t)$ functions for the Goel–Okumoto NHPP model ($a = 175$, $b = 0.05$).

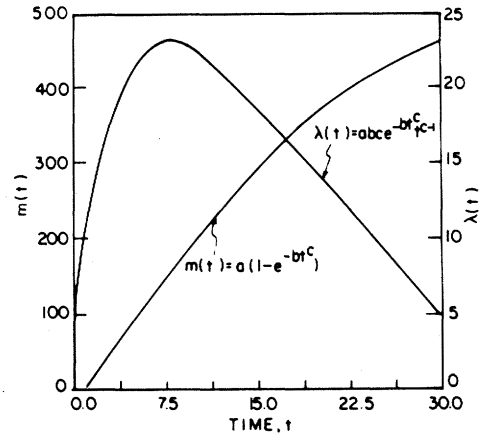

Fig. 4. A typical plot of the $m(t)$ and $\lambda(t)$ functions for the Goel generalized NHPP model ($a = 500$, $b = 0.015$, $c = 1.5$).

ent approach and studied the fault counts over a series of time intervals. Goel and Okumoto [11] introduced a time dependent failure rate of the underlying Poisson process and developed the necessary analytical details of the models. A generalization of this model was proposed by Goel [16]. These and some other models in this class are described below.

Goel–Okumoto Nonhomogeneous Poission Process Model

In this model Goel and Okumoto [12] assumed that a software system is subject to failures at random times caused by faults present in the system. Letting $N(t)$ be the cumulative number of failures observed by time t, they proposed that $N(t)$ can be modeled as a nonhomogeneous Poisson process, i.e., as a Poisson process with a time dependent failure rate. Based on their study of actual failure data from many systems, they proposed the following form of the model

$$P\{N(t) = y\} = \frac{(m(t))^y}{y!} e^{-m(t)}, \quad y = 0, 1, 2, \cdots$$

where

$$m(t) = a(1 - e^{-bt}),$$

and

$$\lambda(t) \equiv m'(t) = abe^{-bt}.$$

Here $m(t)$ is the expected number of failures observed by time t and $\lambda(t)$ is the failure rate. Typical plots of the $m(t)$ and $\lambda(t)$ functions are shown in Fig. 3.

In this model a is the expected number of failures to be observed eventually and b is the fault detection rate per fault. It should be noted that here the number of faults to be detected is treated as a random variable whose observed value depends on the test and other environmental factors. This is a fundamental departure from the other models which treat the number of faults to be a fixed unknown constant.

In some environments a different form of the $m(t)$ function might be more suitable than the one given above, see, e.g., Ohba [36] and Yamada *et al.* [48].

Using a somewhat different approach than described above, Schneidewind [42] had earlier studied the number of faults detected during a time interval and failure counts over a series of time intervals. He assumed that the failure process is a nonhomogeneous Poisson process with an exponentially decaying intensity function given by

$$d(i) = \alpha e^{-\beta i}, \ \alpha, \beta > 0, \ i = 1, 2, \cdots$$

where α and β are the parameters of the model.

Goel Generalized Nonhomogeneous Poisson Process Model

Most of the times between failures and failure count models assume that a software system exhibits a decreasing failure rate pattern during testing. In other words, they assume that software quality continues to improve as testing progresses. In practice, it has been observed that in many testing situations, the failure rate first increases and then decreases. In order to model this increasing/decreasing failure rate process, Goel [16], [17] proposed the following generalization of the Goel–Okumoto NHPP model.

$$P\{N(t) = y\} = \frac{(m(t))^y}{y!} e^{-m(t)}, \quad y = 0, 1, 2, \cdots,$$

$$m(t) = a(1 - e^{-bt^c}),$$

where a is expected number of faults to be eventually detected, and b and c are constants that reflect the quality of testing. The failure rate for the model is given by

$$\lambda(t) \equiv m'(t) = abc\, e^{-bt^c} t^{c-1}.$$

Typical plots of the $m(t)$ and $\lambda(t)$ functions are shown in Fig. 4.

Musa Execution Time Model

In this model Musa [31] makes assumptions that are similar to those of the JM model except that the process

modelled is the number of failures in specified execution time intervals. The hazard function for this model is given by

$$z(\tau) = \phi f(N - n_c)$$

where τ is the execution time utilized in executing the program up to the present, f is the linear execution frequency (average instruction execution rate divided by the number of instructions in the program), ϕ is a proportionality constant, which is a fault exposure ratio that relates fault exposure frequency to the linear execution frequency, and n_c is the number of faults corrected during $(0, \tau)$.

One of the main features of this model is that it explicitly emphasizes the dependence of the hazard function on execution time. Musa also provides a systematic approach for converting the model so that it can be applicable for calendar time as well.

Shooman Exponential Model

This model is essentially similar to the JM model. For this model the hazard function [43], [44] is of the following form

$$z(t) = k\left[\frac{N}{I} - n_c(\tau)\right]$$

where t is the operating time of the system measured from its initial activation, I is the total number of instructions in the program, τ is the debugging time since the start of system integration, $n_c(\tau)$ is the total number of faults corrected during τ, normalized with respect to I, and k is a proportionality constant.

Generalized Poisson Model

This is a variation of the NHPP model of Goel and Okumoto and assumes a mean value function [1] of the following form.

$$m(t_i) = \phi(N - M_{i-1})\, t_i^{\alpha}$$

where M_{i-1} is the total number of faults removed up to the end of the $(i - 1)$st debugging interval, ϕ is a constant of proportionality, and α is a constant used to rescale time t_i.

IBM Binomial and Poisson Models

In these models Brooks and Motley [6] consider the fault detection process during software testing to be a discrete process, following a binomial or a Poisson distribution. The software system is assumed to be developed and tested incrementally. They claim that both models can be applied at the module or the system level.

Musa–Okumoto Logarithmic Poisson Execution Time Model

In this model [33] the observed number of failures by some time τ is assumed to be a NHPP, similar to the Goel–Okumoto model, but with a mean value function which is a function of τ, viz.

$$\mu(\tau) = \frac{1}{\theta} \cdot \ln(\lambda_0 \theta \tau + 1),$$

where λ_0 and θ represent the initial failure intensity and the rate of reduction in the normalized failure intensity per failure, respectively. This model is also closely related to Moranda's geometric de-eutrophication model [29] and can be viewed as a continuous version of this model.

V. Fault Seeding and Input Domain Based Models

In this section we give a brief description of a few time-independent models that have been proposed for assessing software reliability. As mentioned earlier, the two approaches proposed for this class of models are fault seeding and input domain analysis.

In fault seeding models, a known number of faults is seeded (planted) in the program. After testing, the numbers of exposed seeded and indigenous faults are counted. Using combinatorics and maximum likelihood estimation, the number of indigenous faults in the program and the reliability of the software can be estimated.

The basic approach in the input domain based models is to generate a set of test cases from an input (operational) distribution. Because of the difficulty in estimating the input distribution, the various models in this group partition the input domain into a set of equivalence classes. An equivalence class is usually associated with a program path. The reliability measure is calculated from the number of failures observed during symbolic or physical execution of the sampled test cases.

Mills Seeding Model

The most popular and most basic fault seeding model is Mills' Hypergeometric model [27]. This model requires that a number of known faults be randomly seeded in the program to be tested. The program is then tested for some amount of time. The number of original indigenous faults can be estimated from the numbers of indigenous and seeded faults uncovered during the test by using the hypergeometric distribution. The procedure adopted in this model is similar to the one used for estimating population of fish in a pond or for estimating wildlife. These models are also referred to as tagging models since a given fault is tagged as seeded or indigenous.

Lipow [23] modified this problem by taking into consideration the probability of finding a fault, of either kind, in any test of the software. Then, for statistically independent tests, the probability of finding given numbers of indigenous and seeded faults can be calculated. In another modification, Basin [2] suggested a two stage procedure with the use of two programmers which can be used to estimate the number of indigenous faults in the program.

Nelson Model

In this input domain based model [35], the reliability of the software is measured by running the software for a sample of n inputs. The n inputs are randomly chosen from

the input domain set $E = (E_i : i = 1, \cdots, N)$ where each E_i is the set of data values needed to make a run. The random sampling of n inputs is done according to a probability distribution P_i; the set $(P_i : i = 1, \cdots N)$ is the operational profile or simply the user input distribution. If n_e is the number of inputs that resulted in execution failures, then an unbiased estimate of software reliability $\hat{R}_1$ is $\{1 - (n_e/n)\}$. However, it may be the case that the test set used during the verification phase may not be representative of the expected operational usage. Brown and Lipow [7] suggested an alternative formula for $\hat{R}$ which is

$$\hat{R}_2 = 1 - \sum_{j=1}^{N} \left(\frac{f_j}{n_j}\right) p(E_j)$$

where n_j is the number of runs sampled from input subdomain E_j and f_j is the number of failures observed out of n_j runs.

The main difference between Nelson's $\hat{R}_1$ and Brown and Lipow's $\hat{R}_2$ is that the former explicitly incorporates the usage distribution or the test case distribution while the latter implicitly assumes that the accomplished testing is representative of the expected usage distribution. Both models assume prior knowledge of the operational usage distribution.

Ramamoorthy and Bastani Model

In this input domain based model, the authors are concerned with the reliability of critical, real-time, process control programs. In such systems no failures should be detected during the reliability estimation phase, so that the reliability estimate is one. Hence, the important metric of concern is the confidence in the reliability estimate. This model provides an estimate of the conditional probability that the program is correct for all possible inputs given that it is correct for a specified set of inputs. The basic assumption is that the outcome of each test case provides at least some stochastic information about the behavior of the program for other points which are close to the test point. The specific model is discussed in [3], [38]. A main result of this model is

$$P\{\text{program is correct for all points in } [a, a + V] \mid \text{it is correct for test cases having successive distances } x_j, j = 1, \cdots, n-1\} = e^{-\lambda V} \prod_{j=1}^{n-1} \left[\frac{2}{1 + e^{-\lambda x_j}}\right],$$

where λ is a parameter which is deduced from some measure of the complexity of the source code.

Unlike other sampling models, this approach allows any test case selection strategy to be used. Hence, the testing effort can be minimized by choosing test cases which exercise error-prone constructs. However, the model concerning the parameter λ needs to be validated experimentally.

A related model based on fuzzy set theory is discussed in [4].

VI. Model Assumptions and Limitations

In this section we evaluate the implications of the various assumptions underlying the models described above. The main purpose of the following discussion is to focus attention on the framework within which the existing models have been developed. The applicability of such models during the software development cycle will be discussed in the next section.

Before proceeding further, it is helpful to note that a precise, unambiguous statement of the underlying assumptions is necessary to develop a mathematical model. The physical process being modeled, the software failure phenomenon in our case, can hardly be expected to be so precise. It is, therefore, necessary to have a clear understanding of the statement as well as the intent of an assumption.

In the following discussion, the assumptions are evaluated one at a time. Not all of the assumptions discussed here are relevant to any given model but, as a totality, they provide an insight into the kind of limitations imposed by them on the use of the software reliability models. It should be pointed out that the arguments presented here are not likely to be universally applicable because the software development process is very environment dependent. What holds true in one environment may not be true in another. Because of this, even assumptions that seem reasonable, e.g., during the testing of one function or system, may not hold true in subsequent testing of the same function or system. The ultimate decision about the appropriateness of the underlying assumptions and the applicability of the models will have to be made by the user of a model. What is presented here should be helpful in determining whether the assumptions associated with a given model are representative of the user's development environment and in deciding which model, if any, to use.

Times Between Failures Are Independent

This assumption is used in all times between failure models and requires that successive failure times be independent of each other. In general, this would be the case if successive test cases were chosen randomly. However, testing, especially functional testing, is not based on independent test cases, so that the test process is not likely to be random. The time, or the additional number of test cases, to the next failure may very well depend on the nature or time of the previous fault. If a critical fault is uncovered, the tester may decide to intensify the testing process and look for more potential critical faults. This in turn may mean shorter time to the next failure. Although strict adherence to this assumption is unlikely, care should be taken in ensuring some degree of independence in data points.

A Detected Fault Is Immediately Corrected

The models that require this assumption assume that the software system goes through a purification process as testing uncovers faults. An argument can be made that

this assumption is at least implicitly satisfied in many testing situations. Sometimes, when a fault is encountered, testing can proceed without removing it. In that case, the future fault detection process can be assumed to behave as if the fault had been physically removed. If, however, the fault is in the path that must be tested further, this assumption would be satisfied only if the fault is removed prior to proceeding with the remainder of the test bucket or if new test cases were generated to get around it.

No New Faults Are Introduced During the Fault Removal Process

The purpose of this assumption is to ensure that the modeled failure process does have a monotonic pattern. That is, the subsequent faults are exposed from a system that has less faults than before. In general, this may not be true if faults are debugged after each occurrence because other paths may be affected during debugging, leading to additional faults in the system. It is generally considered to be a restrictive assumption in reliability models. The only way to strictly satisfy this is to ensure that the correction process does not introduce new faults. If, however, the additional faults introduced constitute a very small fraction of the fault population, the practical effect on model results would be minimal.

Failure Rate Decreases with Test Time

This assumption implies that the software gets better with testing in a statistical sense. This seems to be a reasonable assumption in most cases and can be justified as follows. As testing proceeds, faults are detected. They are either removed before testing continues or they are not removed and testing is shifted to other parts of the program. In the former case, the subsequent failure rate decreases explicitly. In the latter case, the failure rate (based upon the entire program) decreases implicitly since a smaller portion of the code is subjected to subsequent testing.

Failure Rate Is Proportional to the Number of Remaining Faults

This assumption implies that each remaining fault has the same chance of being detected in a given testing interval between failures. It is a reasonable assumption if the test cases are chosen to ensure equal probability of executing all portions of the code. However, if one set of paths is executed more thoroughly than another, more faults in the former are likely to be detected than in the latter. Faults residing in the unreachable or never tested portion of the code will obviously have a low, or zero, probability of being detected.

Reliability Is a Function of the Number of Remaining Faults

This assumption implies that all remaining faults are equally likely to appear during the operation of a system and is used when reliability estimates are based on the number of remaining faults. If the usage is uniform, then this is clearly a reasonable assumption. If, however, some portions are more likely to be executed than others, this assumption will not hold. However, the reliability of the system can be recomputed by incorporating information about differences in usage. In other words, a reliability measure conditioned on usage rather than based on the number of remaining faults, would be more suitable. If, however, such information is not available, the assumption of uniform usage is the only reasonable one to make. In that case, the estimated reliability value should be interpreted with caution.

Time Is Used as a Basis for Failure Rate

Most models use time as a basis for determining changes in failure rate. This usage assumes that testing effort is proportional to either calendar time or execution time. Also, time is generally easy to measure and most testing records are kept in terms of time. Another argument in favor of this measure is that time tends to smooth out differences in test effort.

If, however, testing is not proportional to time, the models are equally valid for any other relevant measure. Some examples of such measures are lines of code tested, number of functions tested, and number of test cases executed.

Failure Rate Increases Between Failures

This assumption implies that the likelihood of finding a fault increases as the testing time increases within a given failure interval. This would be a justifiable assumption if software were assumed to be subject to wearout within the interval. But, generally, this is not the case with software systems. Another situation where such an assumption might be justifiable is where testing intensity increases within the interval in the same fashion as does the failure rate.

Testing Is Representative of the Operational Usage

This assumption is necessary when a reliability estimate based on testing is projected into the operational phase. It is used primarily in input domain based models. The times between failures and fault count models would also need this assumption if they are used to assess operational reliability.

Test cases are generally chosen to ensure that the functional requirements of the system are correctly met. A given user of the system, however, may not use the functions in the same proportion as done during testing. In that case, testing will not reflect the operational usage. If information about usage pattern is available, testing effort can be modified to be representative of the use profile.

VII. Applicability of Software Reliability Models

In this section we suggest the classes of models that might be applicable in various phases of the software development process. Some of the general comments made in the beginning of Section VI about the importance and

TABLE I
LIST OF KEY ASSUMPTIONS BY MODEL CATEGORY

Times Between Failures (TBF) Models
- Independent times between failures.
- Equal probability of the exposure of each fault.
- Embedded faults are independent of each other.
- Faults are removed after each occurrence.
- No new faults introduced during correction, i.e., perfect fault removal.

Fault Count (FC) Models
- Testing intervals are independent of each other.
- Testing during intervals is reasonably homogeneous.
- Numbers of faults detected during nonoverlapping intervals are independent of each other.

Fault Seeding (FS) Models
- Seeded faults are randomly distributed in the program.
- Indigeneous and seeded faults have equal probabilities of being detected.

Input Domain Based (IDB) Models
- Input profile distribution is known.
- Random testing is used.
- Input domain can be partitioned into equivalent classes.

interpretation of assumptions are also applicable to the discussion here. In particular, note that a precise statement of assumptions is necessary for modeling even though the development process being modeled is extremely unlikely to be that precise. A partial explanation for this apparent inconsistency lies in the fact that a model is, simply, an attempt to summarize the complexity of the real process in order to understand it and possibly control it. In order to be useful, a software reliability model, thus, has to be simple and cannot capture in detail every facet of the modeled failure process. A realization of such constrants imposed on a mathematical model would be helpful in choosing one which can adequately represent the environment within a given development phase.

In the following discussion, we consider the four classes of software reliability models (see Table I) and assess their applicability during the design, unit testing, integration testing, and operational phases.

Design Phase

During the design phase, faults may be detected visually or by other formal or informal procedures. Existing software reliability models are not applicable during this phase because the test cases needed to expose faults as required by fault seeding and input domain based models do not exist, and the failure history required by time dependent models is not available.

Unit Testing

The typical environment during module coding and unit testing phase is such that the test cases generated from the module input domain do not form a representative sample of the operational usage distribution. Further, times between exposures of module faults are not random since the test strategy employed may not be random testing. In fact, test cases are usually executed in a deterministic fashion. Given these conditions, it seems that the fault seeding models are applicable provided it can be assumed that the indigenous and seeded fault have equal probabilities of being detected. However, a difficulty could arise if the programmer is also the tester in this phase. The input domain based models seem to be applicable, except that matching the test profile to operational usage distribution could be difficult. Due to these difficulties, such models, although applicable, may not be usable.

The time dependent models, especially the time between failures models, do not seem to be applicable in this environment since the independent times between failures assumption is seriously violated.

Integration Testing

A typical environment during integration testing is that the modules are integrated into partial or whole systems and test cases are generated to verify the correctness of the integrated system. Test cases for this purpose may be generated randomly from an input distribution or may be generated deterministically using a reliable test strategy, the latter being probably more effective. The exposed faults are corrected and there is a strong possibility that the removal of exposed faults may introduce new faults.

Under such testing conditions, fault seeding models are theoretically applicable since we still have the luxury of seeding faults into the system. Input domain based models based on an explicit test profile distribution are also applicable. The difficulty in applying them at this point is the very large number of logic paths generated by the whole system.

If deterministic testing (e.g., boundary value analysis, path testing) is used, times between failures models may not be appropriate because of the violation of the independence of interfailure times assumption. Fault count models may be applicable if sets of test cases are independent of each other, even if the tests within a set are chosen deterministically. This is so because in such models the system failure rate is assumed to decrease as a result of executing a set of test cases and not at every failure.

If random testing is performed according to an assumed input profile distribution, then most of the existing software reliability models are applicable. Input domain based models, if used, should utilize a test profile distribution which is statistically equivalent to the operational profile distribution. Fault seeding models are applicable likewise, since faults can be seeded and the equal probability of fault detection assumption may not be seriously violated. This is due to the random nature of the test generation process.

Times between failures and failure count models are most applicable with random testing. The next question could be about choosing a specific model from a given class. Some people prefer to try a couple of these models on the same failure history and then choose one. However, because of different underlying assumptions of these models, there are subtle distinctions among them. Therefore, as far as possible, the choice of a specific model

should be based on the development environment considerations.

Acceptance Testing

During acceptance testing, inputs based on operational usage are generated to verify software acceptability. In this phase, seeding of faults is not practical and the exposed faults are not usually immediately corrected. The fault seeding and times between failures models are thus not applicable. Many other considerations here are similar to those of intergration testing so that the fault count and input domain based models are generally applicable.

Operational Phase

When the reliability of the software as perceived by the developer or the "friendly users" is already acceptable, the software is released for operational use. During the operational phase, the user inputs may not be random. This is because the user may use the same software function or path on a routine basis. Inputs may also be correlated (e.g., in real-time systems), thus losing their randomness. Furthermore, faults are not always immediately corrected. In this environment, fault-count models are likely to be most applicable and could be used for monitoring software failure rate or for determining the optimum time for installing a new release.

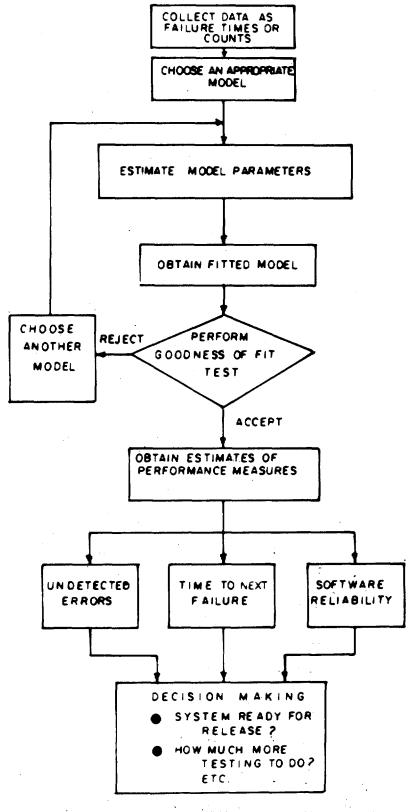

Fig. 5. Flowchart for software reliability modeling and decision making.

VIII. Development and Use of a Model

A step-by-step procedure for fitting a model to software failure data is presented in this section. The procedure is illustrated via analyses of data from a medium size, real-time command and control system. The use of the fitted model for computing reliability and other performance measures, as well as for decision-making, is also explained.

Modeling Procedure

The various steps of the model fitting and decision making process are shown in Fig. 5 and are described below.

Step 1—Study Software Failure Data: The models discussed in this paper require that failure data be available. For most of the models, such data should be in the form of either times between failures or as failure counts. The first step in developing a model is to carefully study such data in order to gain an insight into the nature of the process being modeled.

It is highly desirable to plot the data as a function of, say, calendar time, execution time, or number of test cases executed. The objective of such plots is to try to determine the appropriate variables to use in the model. For example, based on the data and information about the development environment, a model of failures versus unique test cases run may be more important and relevant than a model of failures versus, say, calendar time. Sometimes it is desirable to model several such combinations and then use the fitted models for answering a variety of questions about the failure process. Occasionally, it may be necessary to normalize the data to, for example, account for changes in system size during testing.

Step 2—Choose a Reliability Model: The next step is to choose an appropriate model based upon an understanding of the testing process and of the assumptions underlying the models discussed earlier. The data and plots from Step 1 are likely to be very helpful in this process. A check about the "goodness" of the chosen model for the data at hand will be made in Step 5.

Step 3—Obtain Estimates of Model Parameters: Different methods are generally required depending upon the nature of available data. The most commonly used one is the method of maximum likelihood because it has very good statistical properties. However, sometimes, the method of least squares or some other method may be preferred.

Step 4—Obtain the Fitted Model: The fitted model is obtained by substituting the estimated values of the parameters in the chosen model. At this stage, we have a fitted model based on the available failure data and the chosen model form.

Step 5—Perform Goodness-of-Fit Test: Before proceeding further, it is advisable to conduct the Kolmogorov–Smirnov, or some other suitable goodness-of-fit test to check the model fit. If the model fits, i.e., if it is a satisfactory descriptor of the observed failure process, we can move ahead. However, if the model does not fit, we have to collect additional data or seek a better, more appropriate model. There is no easy answer to either how much data to collect or how to look for a better model. Decisions

on these issues are very much problem dependent and require a clear understanding of the models and the software development environment.

Step 6—Obtain Estimates of Performance Measures: At this stage, we can compute various quantitative measures to assess the performance of the software system. Some useful measures are shown in Fig. 5. Confidence bounds can also be obtained for these measures to evaluate the degree of uncertainty in the computed values of the performance measures.

Step 7—Decision Making: The ultimate objective of developing a model is to use it for making some decisions about the software system, e.g., whether to release the system or continue testing. Such decisions are made at this stage of the modeling process based on the information developed in the previous steps.

An Example of Software Reliability Modeling

We now employ the above procedure to illustrate the development of a software reliability model based on failure data from a real-time, command and control system. The delivered number of object instructions for this system was 21 700 and it was developed by Bell Laboratories. The data were reported by Musa [32] and represent the failures observed during system testing for 25 hours of CPU time.

For purposes of this illustration, we employ the NHPP model of Goel and Okumoto [12]. We do so because of its simplicity and applicability over a wide range of testing situations as also noted by Misra [28], who successfully used this model to predict the number of remaining faults in a space shuttle software subsystem.

Step 1: The original data were reported as times between failures. To overcome the possible lack of independence among these values, we summarized the data into numbers of failures per hour of execution time. The summarized data are given in Table II. A plot of the hourly data is shown in Fig. 6 and a plot of $N(t)$, the cumulative number of failures by t, is shown in Fig. 7. Some other plots shown in Fig. 7 will be discussed later.

Step 2: A study of the data in Table II and of the plot in Fig. 6 indicates that the failure rate (number of failures per hour) seems to be decreasing with test time. This means that an NHPP with a mean value function $m(t) = a(1 - e^{-bt})$ should be a reasonable model to describe the failure process.

Step 3: For the above model, two parameters, a and b, are to be estimated from the failure data. We chose to use the method of maximum likelihood for this purpose [14], [16]. The estimated values for the two parameters are $\hat{a} = 142.32$ and $\hat{b} = 0.1246$. Recal that $\hat{a}$ is an estimate of the expected total number of faults likely to be detected and $\hat{b}$ represents the number of faults detected per fault per hour.

Step 4: The fitted model based on the data of Table II and the parameters estimated in Step 3 is

$$\hat{m}(t) = 142.32\ (1 - e^{-0.1246t})$$

TABLE II
FAILURES IN 1 HOUR (EXECUTION TIME) INTERVALS AND CUMULATIVE FAILURES

Hour	Number of Failures	Cumulative Failures
1	27	27
2	16	43
3	11	54
4	10	64
5	11	75
6	7	82
7	2	84
8	5	89
9	3	92
10	1	93
11	4	97
12	7	104
13	2	106
14	5	111
15	5	116
16	6	122
17	0	122
18	5	127
19	1	128
20	1	129
21	2	131
22	1	132
23	2	134
24	1	135
25	1	136

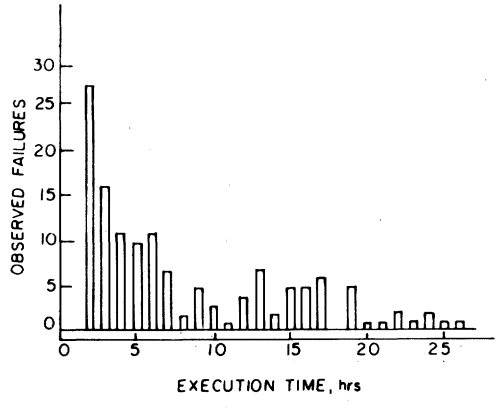

Fig. 6. Plot of the number of failures per hour.

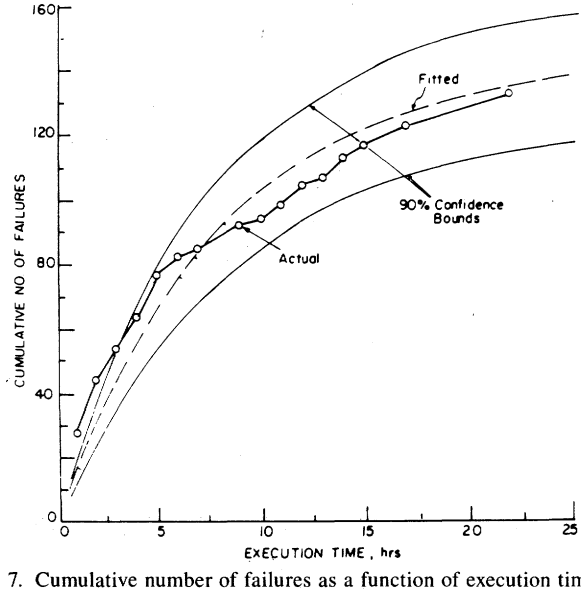

Fig. 7. Cumulative number of failures as a function of execution time and confidence bounds.

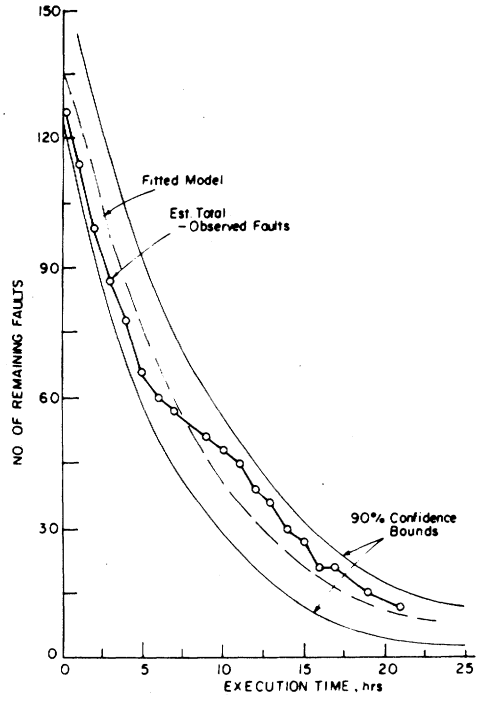

Fig. 8. Estimated remaining number of faults and confidence bounds.

and

$$\hat{\lambda}(t) = 17.73 \cdot e^{-0.1246t}.$$

Step 5: In this case, we used the Kolmogorov–Smirnov goodness-of-fit test for checking the adequacy of the model. For details of this test, see Goel [17]. Basically, the test provides a statistical comparison between the actual data and the model chosen in Step 2. The fitted model in Step 4 passed this test so that it could be considered to be a good descriptor of the data in Table II. The plots in Fig. 7 also provide a visual check of the goodness-of-fit of the model.

Step 6: For illustration purposes, we computed only one performance measure, the expected number of remaining faults, at various testing times. A plot of these values is shown in Fig. 8.

Plots of the confidence bounds for the expected cumulative number of failures, and the expected number of remaining faults are also shown in Figs. 7 and 8, respectively. A study of these plots indicates that the chosen NHPP model provides an excellent fit to the data and can be used for purposes of describing the failure behavior as well as for prediction of the future failure process. The information available from this can be used for planning, scheduling, and other management decisions as indicated below.

Step 7: The model developed above can be used for answering a variety of questions about the failure process and for determining the additional test effort required until the system is ready for release. This type of information can be sought at various points of time and one does not have to wait until the end of testing. For illustrative purposes suppose that failure data through only 16 hours of testing are available, and a total of 122 failures (see Table II) have been observed. Based on these data, the fitted model is

$$\hat{m}(t) = 138.37\ (1 - e^{-0.133t}).$$

An estimate of the remaining number of faults is 16.37 with a 90 percent confidence interval of (4.64–28.11). Also, the estimated one hour ahead reliability is 0.165 and the corresponding 90 percent confidence interval is (0.019–0.310).

Now, suppose that a decision to release software for operational use is to be based on the number of remaining faults. Specifically, suppose that we would release the system if the expected number of remaining faults is less than or equal to 10. In the above analysis we saw that the best estimate of this quantity at present is 16.37, which means that we should continue testing in the hope that additional faults can be detected and removed. If we were to carry on a similar analysis after each additional hour of testing, the expected number of remaining faults after 20 hours would be 9.85 so that the above release criterion would be met.

The above simple example was meant to illustrate the kind of information that can be obtained from a software reliability model. In practice, determination of release time, additional testing effort, etc. are based on much more elaborate considerations than remaining faults. The results from models such as the ones developed here can be used as inputs into the decision-making process.

IX. Concluding Remarks

In this paper, we have provided a review and an evaluation of software reliability models. Four classes of analytical models, along with their underlying assumptions and limitations, were described. The use and applicability of such models during software development and operational phases was also discussed. A methodology for developing a model from failure data was proposed and illustrated via an example. The objective was to provide a model user an insight into the usefulness and limitations of such models that would be helpful in determining which model, if any, to use in a given software development environment.

The material presented here primarily dealt with the development and use of a software reliability model. Several related, but important aspects [19] that were not addressed are model unification issues [22], optimum release time determination [9], parametric estimation problems, comparisons of models [1], [15], [41], [46], and alternate approaches to such models [8], [21].

It should be noted that the above analytical models are primarily useful in estimating and monitoring software reliability, viewed as a measure of software quality. Since they treat the software product and the development process as a blackbox, they cannot be explicitly used for assessing the role of various tools and techniques in determining software quality. For this purpose, more detailed

models will be needed that explicitly incorporate information about the software system and the development process.

ACKNOWLEDGMENT

P. Valdes and A. Deb provided valuable help on an earlier version of this paper. Constructive comments by P. Moranda and F. Bastani were very helpful in improving the quality of the paper. The author is grateful to all of them.

REFERENCES

[1] J. E. Angus, R. E. Schafer, and A. Sukert, "Software reliability model validation," in *Proc. Annu. Reliability and Maintainability Symp.*, San Francisco, CA, Jan. 1980, pp. 191–193.
[2] S. L. Basin, "Estimation of software error rate via capture–recapture sampling," Science Applications, Inc., Palo Alto, CA, 1974.
[3] F. B. Bastani, "An input domain based theory of software reliability and its application," Ph.D. dissertation, Univ. California, Berkeley, 1980.
[4] ——, "On the uncertainty in the correctness of computer programs," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 857–864, Sept. 1985.
[5] B. W. Boehm, J. R. Brown, M. Lipow, "Quantitative evaluation of software quality," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976, pp. 592–605.
[6] W. D. Brooks and R. W. Motley, "Analysis of discrete software reliability models," Rep. RADC-TR-80-84, Apr. 1980.
[7] J. R. Brown and M. Lipow, "Testing for software reliability," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975, pp. 518–527.
[8] L. H. Crow and N. D. Singpurwalla, "An empirically developed Fourier series model for describing software failures," *IEEE Trans. Rel.*, vol. R-33, pp. 175–183, June 1984.
[9] E. H. Forman and N. D. Singpurwalla, "An empirical stopping rule for debugging and testing computer software," *J. Amer. Statist. Ass.*, vol. 72, no. 360, pp. 750–757, 1977.
[10] S. Gerhart and L. Yelowitz, "Observations of fallibility in applications of modern programming methodologies," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 195–207, May 1976.
[11] A. L. Goel and K. Okumoto, "An analysis of recurrent software failures in a real-time control system," in *Proc. ACM Annu. Tech. Conf.*, ACM, Washington, DC, 1978, pp. 496–500.
[12] ——, "A time dependent error detection rate model for software reliability and other performance measures," *IEEE Trans. Rel.*, vol. R-28, pp. 206–211, 1979.
[13] ——, "A Markovian model for reliability and other performance measures of software systems," in *Proc. Nat. Comput. Conf.*, New York, vol. 48, 1979, pp. 769–774.
[14] A. L. Goel, "A software error detection model with applications," *J. Syst. Software*, vol. 1, pp. 243–249, 1980.
[15] ——, "A summary of the discussion on an analysis of competing software reliability models," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 501–502, 1980.
[16] ——, "A guidebook for software reliability assessment," Rep. RADC-TR-83-176, Aug. 1983.
[17] ——, "Software reliability modelling and estimation techniques," Rep. RADC-TR-82-263, Oct. 1982.
[18] A. L. Goel, V. R. Basili, and P. M. Valdes, "When and how to use a software reliability model," in *Proc. 7th Software Eng. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.
[19] A. L. Goel, "Software reliability modeling and related topics: A survey," Dep. of Elec. and Comput. Eng., Syracuse Univ., Syracuse, NY, Tech. Rep., Oct. 1985.
[20] Z. Jelinski and P. Moranda, "Software reliability research," In *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic, 1972, pp. 465–484.
[21] W. Kremer, "Birth–death and bug counting," *IEEE Trans. Rel.*, vol. R-32, pp. 27–47, Apr. 1983.
[22] N. Langberg and N. D. Singpurwalla, "Unification of some software reliability models via the Bayesian approach," *SIAM J. Sci. Statist. Comput.*, vol. 6., pp. 781–790, July 1985.
[23] M. Lipow, "Estimation of software package residual errors," TRW, Redondo Beach, CA, Software Series Rep. TRW-SS-72-09, 1972.
[24] ——, "Maximum likelihood estimation of parameters of a software time-to-failure distribution," TRW, Redondo Beach, CA, Systems Group Rep. 2260.1.9.-73B-15, 1972.
[25] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *Appl. Statist.*, vol. 22, pp. 332–346, 1973.
[26] B. Littlewood, "Theories of software reliability: How good are they and how can they be improved?" *IEEE Trans. Software Eng.*, vol. SE-6, pp. 489–500, 1980.
[27] H. D. Mills, "On the statistical validation of computer programs," IBM Federal Syst. Div., Gaithersburg, MD, Rep. 72-6015, 1972.
[28] P. N. Misra, "Software reliability analysis," *IBM Syst. J.*, vol. 22, no. 3, pp. 262–270, 1983.
[29] P. B. Moranda, "Prediction of software reliability during debugging," in *Proc. Annu. Reliability and Maintainability Symp.*, Washington, DC, Jan. 1975, pp. 327–332.
[30] ——, private communication, 1982.
[31] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312–327, 1971.
[32] ——, "Software Reliability Data," DACS, RADC, New York, 1980.
[33] J. D. Musa and K. Okumoto, "A logarithmic Poisson execution time model for software reliability measurement," in *Proc. 7th Int. Conf. Software Eng.*, Orlando, FL, Mar. 1983, pp. 230–237.
[34] G. J. Myers, *Software Reliability, Principles and Practices*. New York: Wiley, 1976.
[35] E. Nelson, "Estimating software reliability from test data," *Microelectron. Rel.*, vol. 17, pp. 67–74, 1978.
[36] M. Ohba, "Software reliability analysis models," *IBM J. Res. Develop.*, vol. 28, pp. 428–443, July 1984.
[37] K. Okumoto and A. L. Goel, "Availability and other performance measures of software systems under imperfect maintenance," in *Proc. COMPSAC*, Chicago IL, Nov. 1978, pp. 66–71.
[38] C. V. Ramamoorthy and F. B. Bastani, "Software reliability: Status and perspectives," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 359–371, July 1982.
[39] J. Reynolds, *The Craft of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[40] G. J. Schick and R. W. Wolverton, "Assessment of software reliability," presented at the 11th Annu. Meeting German Oper. Res. Soc., DGOR, Hamburg, Germany; also in *Proc. Oper. Res.*, Physica-Verlag, Wirzberg-Wien, 1973, pp. 395–422.
[41] G. J. Schick and R. W. Wolverton, "An analysis of computing software reliability models," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 104–120, 1978.
[42] N. F. Schneidewind, "Analysis of error processes in computer software," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975, pp. 337–346.
[43] M. L. Shooman, "Probabilistic models for software reliability prediction," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic, 1972, pp. 485–502.
[44] ——, "Software reliability measurement and models," in *Proc. Annu. Reliability and Maintainability Symp.*, Washington, DC, Jan. 1975, pp. 485–491.
[45] ——, "Structural models for software reliability and prediction," in *Proc. 2nd Int. Conf. Software Eng.*, San Francisco, CA, Oct. 1976, pp. 268–273.
[46] A. N. Sukert, "An investigation of software reliability models," in *Proc. Annu. Reliability and Maintainability Symp.*, Philadelphia, PA, Jan. 1977, pp. 478–484.
[47] T. A. Thayer, M. Lipow, and E. C. Nelson, "Software reliability study," Rep. RADC-TR-76-238, Aug. 1976.
[48] S. Yamada, M. Ohba, and S. Osaki, "S-shaped reliability growth modeling for software error detection," *IEEE Trans. Rel.*, vol. R-32, pp. 475–478, Dec. 1983.

Amrit L. Goel (M'75), for a photograph and biography, see this issue, p. 1410.