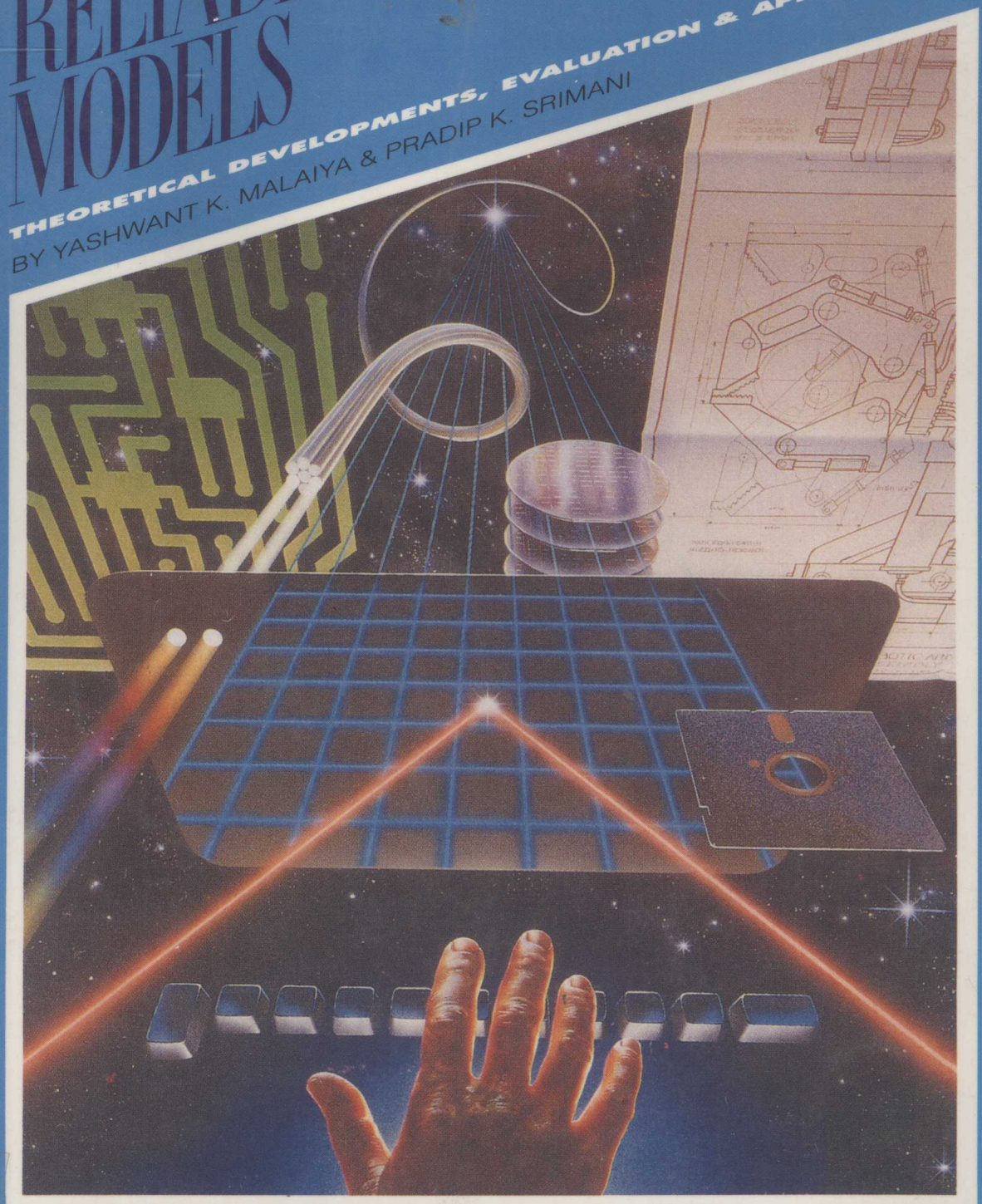
SOFTWARE RELIABILITY MODELS

THEORETICAL DEVELOPMENTS, EVALUATION & APPLICATIONS

BY YASHWANT K. MALAIYA & PRADIP K. SRIMANI



31
36

1951-1991
40 YEARS OF SERVICE
IEEE COMPUTER SOCIETY

The Institute of Electrical and Electronics Engineers, Inc.

TP31
M236

9261760

Software Reliability Models: Theoretical Developments, Evaluation and Applications

IEEE Computer Society Press Technology Series

Edited by

Yashwant K. Malaiya *Pradip K. Srimani*

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

Library of Congress Cataloging-in-Publication Data

Software reliability models : theoretical development, evaluation, and applications / edited by Yashwant K. Malaiya, Pradip K. Srimani.
p. cm.
Includes bibliographical references.
ISBN 0-8186-9110-7. -- ISBN 0-8186-2110-9 (paper). -- ISBN 0-8186-6110-0 (microfiche)
1. Computer software--Reliability. I. Malaiya, Yashwant K. II. Srimani, Pradip K.
QA76.76.R44S662 1991
005--dc20 90-27247
CIP

Published by

IEEE Computer Society Press
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

Copyright © 1990 by the Institute of Electrical and Electronics Engineers, Inc.

Cover designed by Alex Torres

Printed in United States of America

Copyright and Reprint Permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of U.S. copyright law for private use of patrons those articles in this volume that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress Street, Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint or republication permission, write to Director, Publishing Services, IEEE, 345 East 47th Street, New York, NY 10017. All rights reserved.

IEEE Computer Society Press Order Number 2110
Library of Congress Number 90-56332
IEEE Catalog Number EH0329-3
ISBN 0-8186-2110-9 (paper)
ISBN 0-8186-6110-0 (microfiche)
ISBN 0-8186-9110-7 (case)
SAN 264-620X

Additional copies can be ordered from:

IEEE Computer Society Press
Customer Service Center
10662 Los Vaqueros Circle
P.O. Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society
13, Avenue de l'Aquilon
B-1200 Brussels
BELGIUM

IEEE Computer Society
Ooshima Building
2-19-1 Minami-Aoyama,
Minato-Ku
Tokyo 107, JAPAN

IEEE Service Center
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

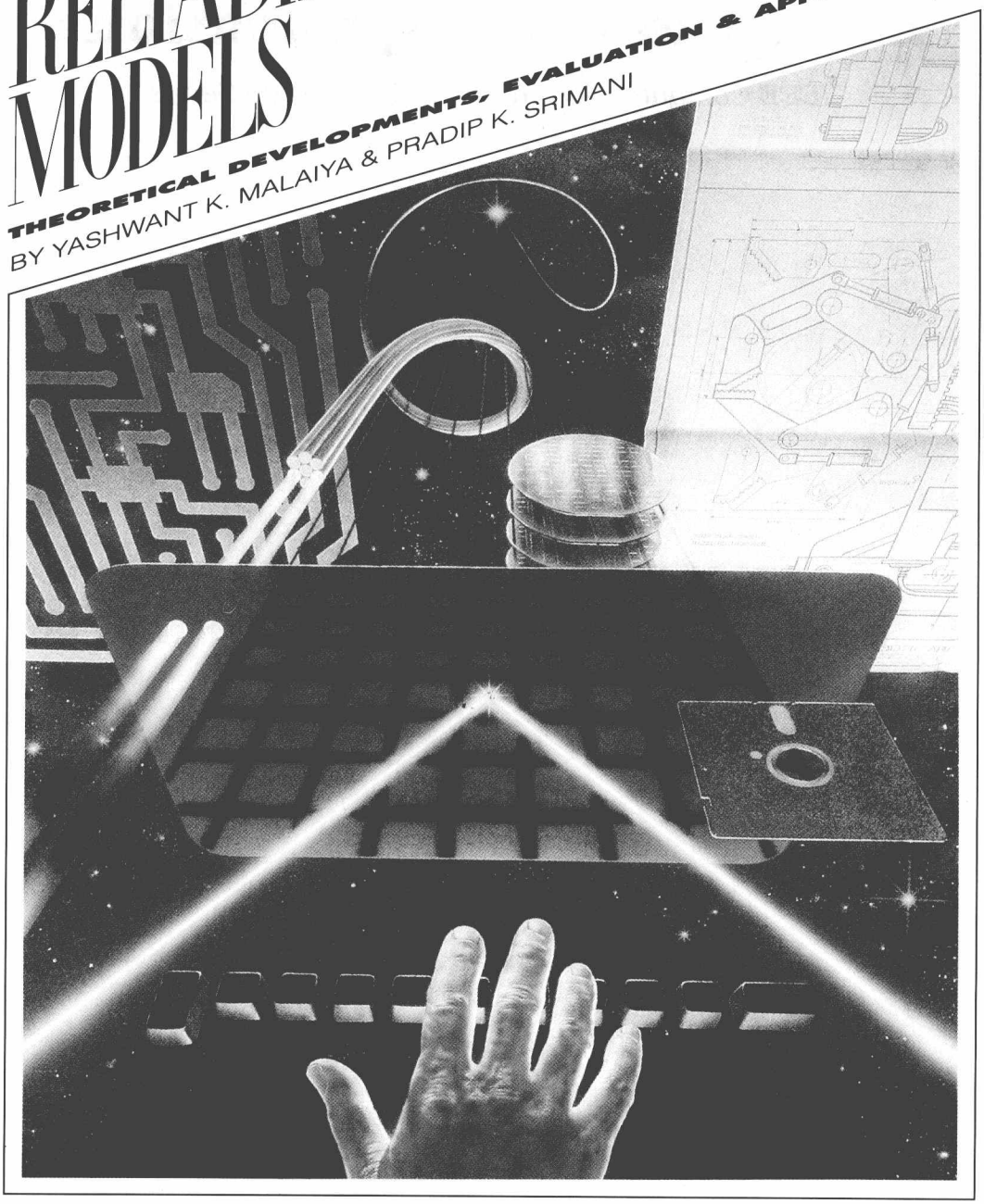
Software Reliability Models: Theoretical Developments, Evaluation and Applications

IEEE Computer Society Press Technology Series

SOFTWARE RELIABILITY MODELS

THEORETICAL DEVELOPMENTS, EVALUATION & APPLICATIONS

BY YASHWANT K. MALAIYA & PRADIP K. SRIMANI



1951-1991
40 YEARS OF SERVICE
IEEE COMPUTER SOCIETY

The Institute of Electrical and Electronics Engineers, Inc.

试读结束，需要全本PDF请购买 www.ertongbook.com

Introduction

Software often constitutes the more expensive part of the computer solution of any problem, and the demand for more and better software is ever increasing. Applications are becoming more sophisticated, demanding high quality reliable and cost effective software. The software designer has to ensure an acceptable level of quality and reliability before a product is released. So it is essential for the designers and the users to have an effective tool to evaluate the reliability of software. Software reliability is defined as the probability of a failure free operation of a computer program for a specified time in a given environment [Mus87]. A failure is simply any deviation in the program behavior from its requirements. The best approach to evaluate software reliability or failure intensity in any program quantitatively is to use some software reliability growth model. A software reliability model (SRM) is a mathematical model that represents failures as a *random process* that is characterized by either times of failures or numbers of failures at fixed times.

These reliability models can be used to evaluate development status during testing of a software project as well as to evaluate the software engineering technology (tools) quantitatively. These models are also used to monitor operational performance of software and to control addition/alteration of features during maintenance of a product. A quantitative understanding of software quality and the factors affecting it also increase insight into the entire software development process. In recent years, the use of software reliability models has become widely accepted and they are being used at AT&T, HP, IBM, Cray and other companies.

1 Software Reliability Models

Modeling of software reliability is conceptually done in three broad stages. First, some assumptions are made to include the *environmental* conditions where the software will run. Second, mathematical formulas are developed to estimate and/or predict useful system parameters like failure intensity, number of failures in an interval, probability distribution of failure intervals and so on. These parameters are estimated from real-life data using some statistical methods like maximum likelihood estimation, least square estimation, or some Bayesian methods. Finally, these parameters are used in predicting future behavior of the software which helps in further planning, maintenance and upgrading of the software.

The specification of an SRM usually involves specification of the failure process as a function of time. For example the failure can be modeled as a birth/death Markov process or as a Poisson process which must be nonhomogeneous if software faults are corrected during the data collection process. In almost all models the failures are assumed to be independent of each other; this assumption is generally found to be reasonable in real life [Mus79, Mus84]. The time component can be specified in two ways: the execution time or the CPU time and the calender time that has elapsed. These two can vary significantly on multiprogrammed or multiprocessor environments. Execution time describes the software behavior more accurately while the calender time is more understandable and hence more acceptable to a system manager.

A large number of such growth models have been proposed, studied, and compared over the last fifteen or so years. Any model must make some assumptions about the development and test environment and potentially there can be infinitely many environments. The environment can change depending on the software system, the design phase as well as the practices and capabilities of the design team. There can also be large variations in data collection and parameter estimation procedures. No one model has been found to be superior to all the others in all possible situations. Some SRMs have good predictability in some situations but work poorly in others. The effectiveness of the models may depend on specific circumstances as well as the validity of the assumptions. Hence it is essential for software designers and users to be reasonably familiar with all the relevant major models to make informed decisions about the quality of any software product. It is equally important for system managers as well as for people engaged in research in software quality and software engineering in general.

2 The Selected Papers

The present volume is a collection of a few representative papers from the rapidly growing literature dealing with the different aspects of software reliability analysis. All the papers included in this book are from the decade of eighties. Early well known classical models [Mus75, Lit73, Goe79, JM75] have been amply studied and compared in the two articles by Goel and by Yamada and Osaki. Our primary objective in making the selection was to give the reader an overview of the current approaches and flavor of the recent directions of research in modeling. It is also to be noted that almost all the recent models are in some way or other based upon some of the early basic models. The papers included here present the major aspects of the theoretical framework; some of the papers also show how the models are being used in actual practice.

Thirteen papers have been included in the present volume. The first eight of them discuss various aspects of SRMs, which can be used with dynamic test data (time between failures or the number of failures is consecutive test periods). Major models are discussed, computational approaches are given, and models are compared. In addition some essential (perhaps questionable) assumptions are examined in detail. The rest of the papers use some different approaches for reliability analysis. Static analysis, which uses static metrics like program size, is sometimes used to supplement SRMs in early phases. The paper by Scott, Gault and McAllister examines evaluation of software fault-tolerance. Two of the papers by Singpurwalla and Soyer and by Weiss and Weyuker present intriguing alternatives to SRMs. What follows is a brief introduction to each of the selected papers.

2.1 Dynamic SRMs

The first paper by Goel presents an excellent comparative overview of all the major earlier models of software reliability. A critical assessment of the applicability and limitations of the underlying assumptions during the software life cycle are also presented. It explains how to fit a given model in

a given application via an appropriate sized real life example. It includes an extensive bibliography of early papers. The second paper by Littlewood challenges a common assumption in reliability modeling that failure rate of a program is a constant multiple of the remaining bugs. It proposes an alternative Baysian approach that results in significant improvement of the model performance and develops compact mathematical expressions for time required and number of bug-fixes required for a given target reliability. In the third paper, Musa and Okumoto present a new variation of nonhomogeneous Poisson type failure process and compare it by using actual data with other models. The paper by Yamada and Osaki is another excellent critical summary of existing models. It also illustrates application examples of the exponential and delayed-S-shaped models by using actual software error data observed during testing.

Although it is assumed that errors are completely removed from software when they are detected, Ohba and Chou show in their paper that this assumption is neither realistic nor needed for the validity of most of the exponential type growth models.

In the next paper, Brocklehurst, Chan, Littlewood, and Snell address the issue of non applicability of any model universally in all situations and propose a new mechanism called recalibration that attempts to help the user choose the right model for the application at hand. Malaiya, Karunanithi, and Verma, in their paper, present experimental results on predictive powers of different reliability models. They have shown that some models tend to work better in several cases. In the eighth paper, Kruger describes his two years' experience with a reliability model in a real-life industrial situation. The paper reinforces the belief that reliability models will be increasingly important and useful in deciding project schedules and in estimating final product quality.

2.2 *Alternative Approaches*

The ninth paper, by Takahashi and Kamayachi, describes a different model based on factors like frequency of program specification change, programmers' skill and volume of design documentation and shows that the model predicts future errors better than those based on program size alone. In the next paper, Singpurwalla and Soyer take a different approach. They introduce a random coefficient autoregressive process of order 1 to describe reliability growth. By varying the structure of dependence between stages of modification, they consider different models and develop interesting mathematical results about likelihood of the models' predictive distributions. Ehrlich and Emerson, in the next paper, investigate the relationship between applicability of any Poisson type growth model and the nature of the system test process and raise some theoretical questions about validity of assuming certain statistical properties of failure occurrence during testing irrespective of the testing process.

Weiss and Weyuker, in the next paper, redefine reliability as a generalization of the probability of the correctness of the software. They introduce a tolerance function and show that, by varying the tolerance function, many natural good reliability models can be designed. In the last paper Scott et al. discuss the reliability of fault-tolerant software. They present models for three different mechanisms of writing fault tolerant programs, present experimental results to validate the models,

and introduce a simple model to evaluate relative costs for increasing reliability in the three types of fault-tolerant programming.

We hope that the present volume will prove to be a handy reference to software engineers, serious users, and researchers in the area of software reliability modeling. For additional reading, one can look into [Mus87, Lit87, SE-11, SE-12] and the references contained therein.

3 Acknowledgement

The work of Y. K. Malaiya was partly supported by an SDIO/IST contract monitored by ONR.

4 References

[Goe79] A.L. Goel and K. Okumoto, "Time Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Trans. Reliability*, Vol. R-28, pp. 206-211, 1979.

[JM75] Z. Jelinski and P.B. Moranda, "Software Reliability Research," (W. Frieberger, editor), *Statistical Computer Performance Evaluation*, Academic Press, Orlando, FL., pp. 465-484, 1972.

[Lit73] B. Littlewood and J.L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," *Applied Statistics*, Vol. 22, pp. 332-346, 1973.

[Lit87] B. Littlewood (editor), *Software Reliability : Achievement and Assessment,* Blackwell Scientific Publication, Oxford, England, 1987.

[Mus75] J.D. Musa, "A Theory of Software Reliability and Its Application," *IEEE Trans. Software Engineering*, Vol. SE-1, pp. 312-327, 1975.

[Mus79] J.D. Musa, "Validity of Execution Time Theory of Software Reliability," *IEEE Trans. Reliability*, Vol. R-28, pp. 239-249, 1979.

[Mus84] J.D. Musa and K. Okumoto, "A Comparison of Time Domains for Software Reliability Models," *Journal of Systems and Software*, Vol. 4, pp. 277-287, 1984.

[Mus87] J.D. Musa, A. Iannino and K. Okumoto, *Software Reliability: Measurement, Prediction, Application,* McGraw-Hill, New York, 1987.

[SE-11] Special Issue on Sofware Reliability – Part I, *IEEE Trans. Software Engineering*, Vol. SE-11, December 1985.

[SE-12] Special Issue on Sofware Reliability – Part II, *IEEE Trans. Software Engineering*, Vol. SE-12, January 1986.

Table of Contents

试读结束，需要全本PDF请购买 www.ertongbook.com

Reprinted from *IEEE Transactions on Software Engineering*, Volume SE-11, Number 12, December 1985, pages 1411–1423. Copyright © 1985 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

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.

EH0329-3/90/0000/0001/$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.

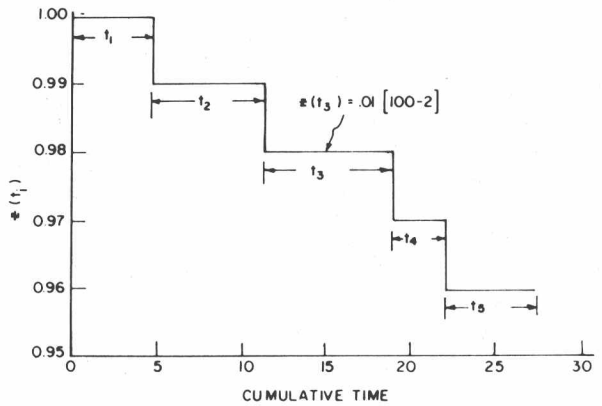


Fig. 1. A typical plot of $Z(t_i)$ for the JM model ($N = 100$, $\phi = 0.01$).

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

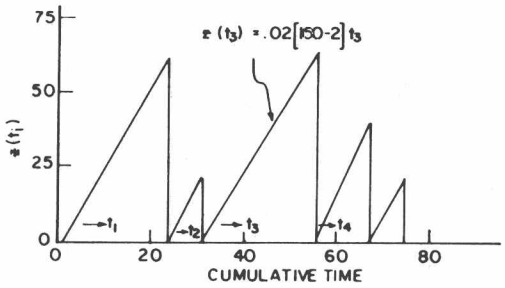


Fig. 2. A typical plot of the hazard function for the SW model ($N = 150$, $\phi = 0.02$).

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

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-

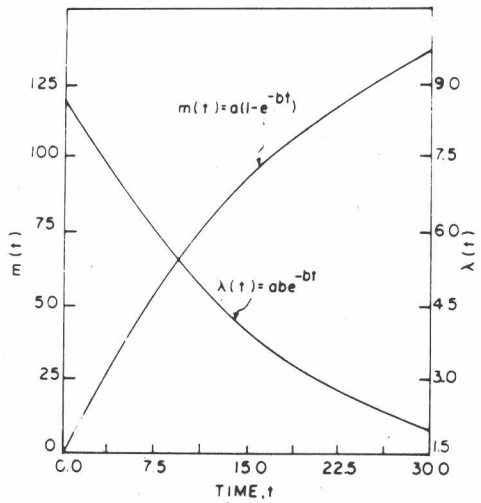


Fig. 3. A typical plot of the $m(t)$ and $\lambda(t)$ functions for the Goel–Okumoto NHPP model ($a = 175$, $b = 0.05$).

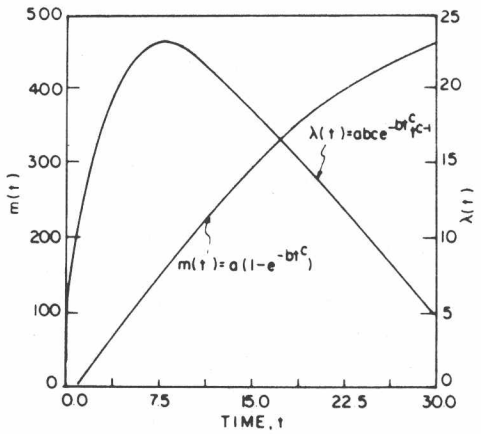


Fig. 4. A typical plot of the $m(t)$ and $\lambda(t)$ functions for the Goel generalized NHPP model ($a = 500$, $b = 0.015$, $c = 1.5$).

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

试读结束，需要全本PDF请购买 www.ertongbook.com