IEEE

IEEE Recommended Practice on Software Reliability

IEEE Reliability Society

Sponsored by the
Standards Committee

1633™

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

27 June 2008

IEEE Std 1633™-2008

IEEE Recommended Practice on Software Reliability

Sponsor

Standards Committee
of the
IEEE Reliability Society

Approved 27 March 2008

IEEE-SA Standards Board

Abstract: The methods for assessing and predicting the reliability of software, based on a life-cycle approach to software reliability engineering, are prescribed in this recommended practice. It provides information necessary for the application of software reliability (SR) measurement to a project, lays a foundation for building consistent methods, and establishes the basic principle for collecting the data needed to assess and predict the reliability of software. The recommended practice prescribes how any user can participate in SR assessments and predictions.
Keywords: software reliability

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-7381-5406-0 STD95776
Print: ISBN 978-0-7381-5407-7 STDPD95776

Introduction

This introduction is not part of IEEE Std 1633-2008, IEEE Recommended Practice on Software Reliability.

Software reliability engineering (SRE) is an established discipline that can help organizations improve the reliability of their products and processes. The American Institute of Aeronautics and Astronautics (AIAA) defines SRE as "the application of statistical techniques to data collected during system development and operation to specify, predict, estimate, and assess the reliability of software-based systems." This recommended practice is a composite of models and tools and describes the "what and how" of SRE. It is important for an organization to have a disciplined process if it is to produce high reliability software. The process is described and enhanced to include a life-cycle approach to SRE that takes into account the risk to reliability due to requirements changes. A requirements change may induce ambiguity and uncertainty in the development process that cause errors in implementing the changes. Subsequently, these errors may propagate through later phases of development and maintenance. These errors may result in significant risks associated with implementing the requirements. For example, reliability risk (i.e., risk of faults and failures induced by changes in requirements) may be incurred by deficiencies in the process (e.g., lack of precision in requirements). Figure a shows the overall SRE closed-loop holistic process. The arrows signify the flow of software product, information, and/or failure data.

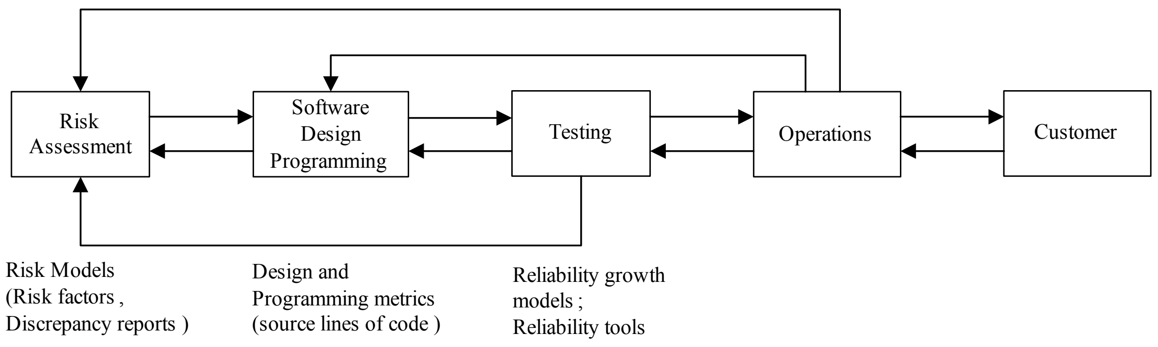

Risk Assessment
Software Design Programming
Testing
Operations
Customer
Risk Models (Risk factors , Discrepancy reports)
Design and Programming metrics (source lines of code)
Reliability growth models ; Reliability tools

Figure a—SRE process

The scope of this recommended practice is to address software reliability (SR) methodologies and tools. This recommended practice does not address systems reliability, software safety, nor software security. The recommended practice only briefly addresses software quality. This recommended practice provides a common baseline for discussion and prescribes methods for assessing and predicting the reliability of software. The recommended practice is intended to be used in support of designing, developing, and testing software and to provide a foundation on which practitioners and researchers can build consistent methods for assessing the reliability of software. It is intended to meet the needs of software practitioners and users who are confronted with varying terminology for reliability measurement and a plethora of models and data collection methods. This recommended practice contains information necessary for the application of SR measurement to a project. This includes SR activities throughout the software life cycle (SLC) starting at requirements generation by identifying the application, specifying requirements, and analyzing requirements and continuing into the implementation phases. It also serves as a reference for research on the subject of SR.

It includes guidance on the following:

— Common terminology.

— Assessment of SR risk.

— Determining whether previously applied software processes are likely to produce code that satisfies a given SR requirement.

— Software design process improvement evaluation and software quality.

— SR assessment procedures (i.e., measure current software reliability).

— Test selection and model selection.

— Data collection procedures to support SR estimation and prediction.

— Determining when to release a software system, or to stop testing the software and implement corrections.

— SR prediction (i.e., predict future software reliability). This includes the calculation of the probability of occurrence of the next failure for a software system, and other reliability metrics.

— Identify elements in a software system that are leading candidates for redesign to improve reliability.

Revisions to the document and notes

This document is a revision of AIAA R-013-1992, Recommended Practice on Software Reliability. The following changes and additions are designed to enhance its usability:

— "Recommended models" has been changed to "initial models" to reflect the fact that this document is a recommended practice. The meaning of "initial models" is that models in this category are designated for initial use. If none of these models is satisfactory for the user's application, the models described in Annex A can be considered.

— Life-cycle approach to SRE.

— SRE process diagram (see Figure a).

— Inclusion of reliability requirements risk assessment.

— Additions to 5.1.

— Identify application.

— Specify the reliability requirement.

— Allocate the requirement.

— Correct designation of informative clauses.

— Simplified and clarified language.

— Elimination of mathematics that do not support the objectives of the recommended practice.

— Correct use of "shall," "should," and "may" to indicate conformance with the recommended practice.

— Correction of errors.

— Upgrade of Schneidewind initial model.

— Addition of John Musa's latest book [B55][a] as a reference.

— Deletion of Assumption 1 in the Musa/Okumoto model.

— The Littlewood/Verrall model has been moved from "initial models" to Annex A because many of its terms were not defined.

[a] The numbers in brackets correspond to those of the bibliography in Annex G.

Structure of the recommended practice

This recommended practice contains five clauses and seven annexes as follows:

— Clause 1 is the introduction, including scope, purpose, audience, and relationships between hardware and SR.
— Clause 2 contains definitions of terms used in this recommended practice.
— Clause 3 gives an overview of SRE.
— Clause 4 provides a list of activities that should be carried out in order to conform with this recommended practice.
— Clause 5 provides information on recommended SR prediction models.
— Annex A provides information on additional SR prediction models.
— Annex B describes methods for combining hardware and SR predictions into a system reliability prediction.
— Annex C provides information on using the recommended practice to help obtain system reliability (including both hardware and SR).
— Annex D contains Table D.1, which lists the two most popular SR measurement tools.
— Annex E contains a comparison of constant, linearly decreasing, and exponentially decreasing models.
— Annex F provides SR prediction tools prior to testing during the early design phases.
— Annex G contains a bibliography of over 70 papers and books about SRE.

Clause 1, Clause 2, and Clause 3 should be read by all users. Clause 3, Annex A, and Annex C provide the basis for establishing the process and the potential uses of the process. Subclause 5.6.1 provides the foundation for establishing an SR data collection program, as well as what information needs to be collected to support the recommended models, which is also described in Annex A. Annex D identifies tools that support the reliability database, as well as the recommended models and the analysis techniques described in Clause 4, Annex A, and Annex C. Finally, to improve the state of the art in SRE continuously, recommended practice users should typically review Clause 1, Clause 2, and Clause 3 and begin applying the techniques described in 5.5, and concluding with Annex D and Annex F on reliability tools.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association Web site at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA Web site at http://standards.ieee.org.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this recommended practice may require use of subject matter covered by patent rights. By publication of this recommended practice, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this recommended practice are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this recommended practice was submitted to the IEEE-SA Standards Board for approval, the Software Reliability Working Group had the following membership:

Norman F. Schneidewind, *Chair*
Louis J. Gullo, *Sponsor Chair and Final Editor*
Kadir Alpaslan Demir, *Editor*

William Everett
William Farr
Herbert Hecht
Michael Hinchey
Samuel Keene
Theodore Keller
Michael Lyu
John Musa
Allen Nikora
Martin Shooman
George Stark
Jeff Voas
Mladen Vouk
Dolores Wallace
Linda Wilbanks

The following members of the individual balloting committee voted on this recommended practice. Balloters may have voted for approval, disapproval, or abstention.

Ahmed Abdelhalim
William J. Ackerman
S. Aggarwal
Ali Al Awazi
Bakul Banerjee
Charles Barest
Hugh Barrass
Thomas Basso
Steven Bezner
Gregory M. Bowen
Juan Carreon
Lanna Carruthers
Norbert N. Carte
Lawrence Catchpole
Weijen Chen
Keith Chow
Raul Colcher
Tommy Cooper
Paul Croll
Geoffrey Darnton
Matthew Davis
Kadir Alpaslan Demir
Bostjan K. Derganc
Antonio Doria
Neal Dowling
Scott P. Duncan
Sourav Dutta
Kameshwar Eranki
Carla Ewart
Harriet Feldman
John Fendrich
Andrew Fieldsend
Kenneth Fodero
Andre Fournier
Avraham Freedman
Michael Geipel
Gregg Giesler
Allan Gillard
Colin M. Glanville
H. Glickenstein
John Garth Glynn
Ron Greenthaler
Randall Groves
C .Guy
Ajit Gwal
John Harauz
Mark Henley
Rutger A. Heunks
Werner Hoelzl
Robert Holibaugh
Chi T. Hon
Peter Hung
Mark Jaeger
James Jones
Lars Juhlin
Piotr Karocki
Samuel Keene
Robert B. Kelsey
Mark Knight
Thomas Kurihara
George Kyle
Marc Lacroix
Daniel Lindberg
G. Luri
L. J. Tajerian Martinez
Richard A. McBride
Michael McCaffrey
Jonathon McLendon
Earl Meiers
Gary Michel
William Milam
James Moore
Thomas Mullins
Jerry Murphy
Rajesh Murthy
John Musa
Michael S. Newman
Mark Paulk
Miroslav Pavlovic
Robert Peterson
Peter Petrov
Ulrich Pohl
Iulian Profir
Annette Reilly
Michael Roberts
Robert Robinson
Terence Rout
Michael Rush
Randall Safier
James Sanders
Helmut H. Sandmayr
Bartien Sayogo
Robert Schaaf
Hans Schaefer
Norman F. Schneidewind
David J. Schultz
Stephen Schwarm
Lynn Simms
Carl Singer
James Sivak
David Smith
Steven Smith
Luca Spotorno
Manikantan Srinivasan
Thomas Starai
James St.Clair
Walter Struppler
Gerald Stueve
Mark Sturza
Noriaki Takenoue
Joseph Tardo
John Thywissen
Thomas Tullia
Vincent J. Tume
Mladen Vouk
John Walz
Robert Wilkens
David Willow
Don Wright
Oren Yuen
Janusz Zalewsk

Contents

IEEE Recommended Practice on Software Reliability

1. Overview

1.1 Scope

Software reliability (SR) models have been evaluated and ranked for their applicability to various situations. Many improvements have been made in SR modeling and prediction since 1992. This revised recommended practice reflects those advances in SR since 1992, including modeling and prediction for distributed and network systems. Situation specific usage guidance was refined and updated. The methodologies and tools included in this recommended practice are extended over the software life cycle (SLC).

1.2 Purpose

The recommended practice promotes a systems approach to SR prediction. Although there are some distinctive characteristics of aerospace software, the principles of reliability are generic, and the results can be beneficial to practitioners in any industry.

1.3 Intended audience

This recommended practice is intended for use by both practitioners (e.g., managers, software developers/engineers, software acquisition personnel, technical managers, and software quality and SR personnel) and researchers. Researchers are considered to be academics in universities and personnel doing research work in government and industry laboratories.

It is assumed that users of this recommended practice have a basic understanding of the SLC and an understanding of statistical concepts.

1.4 Applications of software reliability engineering

The techniques and methodologies presented in this recommended practice have been successfully applied to software projects by industry practitioners in order to do the following:

- Assess SR risk
- Indicate whether a previously applied software process is likely to produce code that satisfies a given SR requirement
- Provide a measure for software design process improvement evaluation and software quality
- SR assessment procedures (i.e., measure current software reliability)
- Data collection procedures to support SR estimation and prediction
- Determine when to release a software system, or to stop testing the software and make improvements
- Calculate the probability of occurrence of the next failure for a software system, and other reliability metrics
- Identify elements in a software system that are leading candidates for redesign to improve reliability
- Indicate software maintenance effort by assessing SR
- Assist software safety certification

1.5 Relationship to hardware reliability

The creation of software and hardware products is alike in many ways and can be similarly managed throughout design and development. While the management techniques may be similar, there are genuine differences between hardware and software (see Kline [B36], Lipow and Shooman [B39][1]). Examples are as follows:

- Changes to hardware require a series of important and time-consuming steps: capital equipment acquisition, component procurement, fabrication, assembly, inspection, test, and documentation. Changing software is frequently more feasible (although effects of the changes are not always clear) and often requires only testing and documentation.
- Software has no physical existence. It includes data as well as logic. Any item in a file can be a source of failure.
- One important difference is that hardware is constrained by physical law. One effect is that testing is simplified, because it is possible to conduct limited testing and use knowledge of the physics of the device to interpolate behavior that was not explicitly tested. This is not possible with software, since a minor change can cause failure.
- Software does not wear out. Furthermore, failures attributable to software faults come without advance warning and often provide no indication they have occurred. Hardware, on the other hand, often provides a period of graceful degradation.

[1] The numbers in brackets correspond to those of the bibliography in Annex G.

- — Software may be more complex than hardware, although exact software copies can be produced, whereas manufacturing limitations affect hardware.
- — Repair generally restores hardware to its previous state. Correction of a software fault always changes the software to a new state.
- — Redundancy and fault tolerance for hardware are common practice. These concepts are only beginning to be practiced in software.
- — Hardware reliability is expressed in wall clock time. SR may be expressed in execution, elapsed, or calendar time.

Despite the previously listed differences, hardware and SR must be managed as an integrated system attribute. However, these differences must be acknowledged and accommodated by the techniques applied to each of these two types of subsystems in reliability analyses.

2. Definitions

For the purposes of this recommended practice, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* [B22] and IEEE Std 610.12™-1990 [B23] should be referenced for terms not defined in this clause.

2.1 assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process).

NOTE—The formulation of test strategies is also part of assessment. Test strategy formulation involves the determination of priority, duration, and completion date of testing, allocation of personnel, and allocation of computer resources to testing.[2]

2.2 calendar time: Chronological time, including time during which a computer may not be running.

2.3 clock time: Elapsed wall clock time from the start of program execution to the end of program execution.

2.4 error: (A) A discrepancy between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition. **(B)** Human action that results in software containing a fault. A coding error may be considered a defect, but it may or may not be considered a fault. Usually, any error or defect that is detected by the system and or observed by the user is deemed a fault. All faults are serious errors that should be corrected. Not all errors are faults.

NOTE—Examples include omission or misinterpretation of user requirements in a software specification and incorrect translation or omission of a requirement in the design specification.

2.5 execution time: (A) The amount of actual or central processor time used in executing a program. **(B)** The period of time during which a program is executing.

2.6 failure: (A) The inability of a system or system component to perform a required function within specified limits. **(B)** The termination of the ability of a functional unit to perform its required function. **(C)** A departure of program operation from program requirements.

NOTE—A failure may be produced when a fault is encountered and a loss of the expected service to the user results.

[2] Notes in text, tables, and figures of a standard are given for information only and do not contain requirements needed to implement this recommended practice.

2.7 failure rate: (A) The ratio of the number of failures of a given category or severity to a given period of time; for example, failures per second of execution time, failures per month. Synonymous with failure intensity. **(B)** The ratio of the number of failures to a given unit of measure, such as failures per unit of time, failures per number of transactions, failures per number of computer runs.

2.8 failure severity: A rating system for the impact of every recognized credible software failure mode.

NOTE—The following is an example of a rating system:

— Severity #1: Loss of life or system

— Severity #2: Affects ability to complete mission objectives

— Severity #3: Workaround available, therefore minimal effects on procedures (mission objectives met)

— Severity #4: Insignificant violation of requirements or recommended practices, not visible to user in operational use

— Severity #5: Cosmetic issue that should be addressed or tracked for future action, but not necessarily a present problem.

2.9 fault: (A) A defect in the code that can be the cause of one or more failures. **(B)** An accidental condition that causes a functional unit to fail to perform its required function. A fault is synonymous with a bug. A fault is an error that should be fixed with a software design change.

2.10 fault tolerance: The survival attribute of a system that allows it to deliver the required service after faults have manifested themselves within the system.

2.11 integration: The process of combining software elements, hardware elements, or both into an overall system.

2.12 maximum likelihood estimation: A form of parameter estimation in which selected parameters maximize the probability that observed data could have occurred.

2.13 module: (A) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, input to or output from an assembler, compiler, linkage editor, or executive routine. **(B)** A logically separable part of a program.

2.14 operational: Pertaining to the status given a software product once it has entered the operation and maintenance phase.

2.15 parameter: A variable or arbitrary constant appearing in a mathematical expression, each value of which restricts or determines the specific form of the expression.

2.16 reliability risk: The probability that requirements changes will decrease reliability.

2.17 software quality: (A) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs, such as conforming to specifications. **(B)** The degree to which software possesses a desired combination of attributes. **(C)** The degree to which a customer or user perceives that software meets his or her composite expectations. **(D)** The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

2.18 software reliability (SR): (A) The probability that software will not cause the failure of a system for a specified time under specified conditions. **(B)** The ability of a program to perform a required function under stated conditions for a stated period of time.

NOTE—For definition **(A)**, the probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered.

2.19 software reliability engineering (SRE): The application of statistical techniques to data collected during system development and operation to specify, estimate, or assess the reliability of software-based systems.

2.20 software reliability estimation: The application of statistical techniques to observed failure data collected during system testing and operation to assess the reliability of the software.

2.21 software reliability model: A mathematical expression that specifies the general form of the software failure process as a function of factors such as fault introduction, fault removal, and the operational environment.

2.22 software reliability prediction: A forecast or assessment of the reliability of the software based on parameters associated with the software product and its development environment.

3. Software reliability modeling—Overview, concepts, and advantages

3.1 General

Software is a complex intellectual product. Inevitably, some errors are made during requirements formulation as well as during designing, coding, and testing the product. The development process for high-quality software includes measures that are intended to discover and correct faults resulting from these errors, including reviews, audits, screening by language-dependent tools, and several levels of test. Managing these errors involves describing the errors, classifying the severity and criticality of their effects, and modeling the effects of the remaining faults in the delivered product, and thereby working with designers to reduce their number of errors and their criticality.

NOTE—The IEEE standard for classifying errors and other anomalies is IEEE Std 1044™-1993 [B25].

Dealing with faults costs money. It also impacts development schedules and system performance (through increased use of computer resources such as memory, CPU time, and peripherals requirements). Consequently, there can be too much as well as too little effort spent dealing with faults. The system engineer (along with management) can use reliability estimation and assessment to understand the current status of the system and make tradeoff decisions.

3.2 Basic concepts

Clause 3 describes the basic concepts involved in SRE and addresses the advantages and limitations of SR prediction and estimation. The basic concept of reliability predictions and assessments of electronic systems and equipment is described in IEEE Std 1413™-1998 [B28]. The objective of IEEE Std 1413-1998 [B28] is to identify required elements for an understandable, credible reliability prediction, which will provide the users of the prediction sufficient information to evaluate the effective use of the prediction results. A reliability prediction should have sufficient information concerning inputs, assumptions, and uncertainty, such that the risk associated with using the prediction results would be understood.

There are at least two significant differences between hardware reliability and SR. First, software does not fatigue or wear out. Second, due to the accessibility of software instructions within computer memories, any line of code can contain a fault that, upon execution, is capable of producing a failure.

An SR model specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment. The failure rate (failures per unit time) of a software system is generally decreasing due to fault identification and removal, as shown in Figure 1. At a particular time, it is possible to observe a history of the failure rate of the software. SR modeling is done to estimate the form of the curve of the failure rate by statistically estimating the parameters associated with the selected model. The purpose of this measure is twofold: 1) to estimate the extra execution time during test required to meet a specified reliability objective and 2) to identify the expected reliability of the software when the product is released.

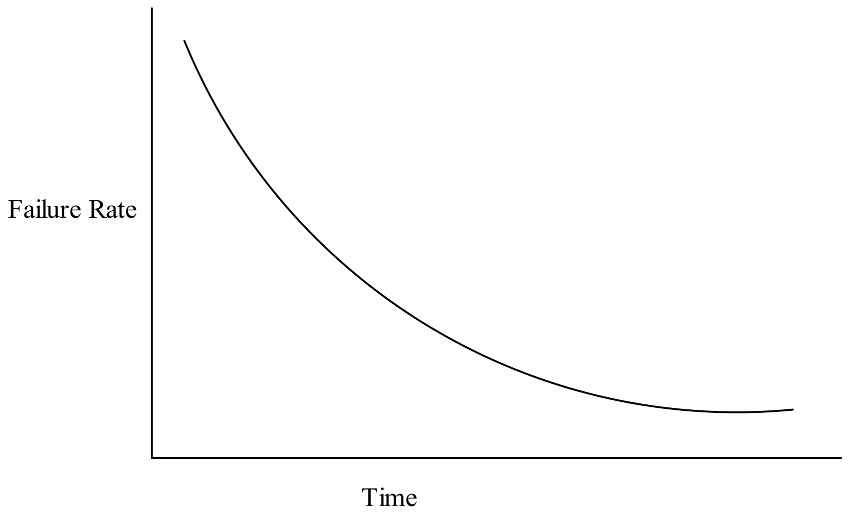


Figure 1—SR failure rate curve

3.3 Limitations of software reliability assessment and prediction

SR models can both assess and predict reliability. The former deals with measuring past and current reliabilities. The latter provides forecasts of future reliability. The word "prediction" is not intended to be used in the common dictionary sense of foretelling future events, particularly failures, but instead as an estimate of the probabilities of future events. Both assessment and prediction need good data if they are to yield good forecasts. Good data implies accuracy (that data is accurately recorded at the time the events occurred) and pertinence (that data relates to an environment that is equivalent to the environment for which the forecast is to be valid). A negative example with respect to accuracy is the restricting of failure report counts to those, which are completely filled out. This is negative because they may represent a biased sample of the total reports. A negative example with respect to pertinence would be the use of data from early test runs at an uncontrolled workload to forecast the results of a later test executed under a highly controlled workload.

3.4 Prediction model advantages/limitations

The premise of most prediction models is that the failure rate is a direct function of the number of faults in the program and that the failure rate will be reduced (reliability will be increased) as faults are detected and eliminated during test or operations. This premise is reasonable for the typical test environment and it has been shown to give credible results when correctly applied. However, the results of prediction models will be adversely affected by the following factors:

— Change in failure criteria
— Significant changes in the code under test
— Significant changes in the computing environment.

All of these factors will require the estimation of a new set of reliability model parameters. Until these can be established, the effectiveness of the model will be impaired. Estimation of new parameters depends on the measurement of several execution time intervals between failures or failure counts in intervals.

Major changes can occur with respect to several of the previously listed factors when software becomes operational. In the operational environment, the failure rate is a function of the fault content of the program, of the variability of input and computer states, and of software maintenance policies. The latter two factors are under management control and are utilized to assess an expected or desired range of values for the failure rate or the downtime due to software causes. Examples of management action that decrease the failure rate include avoidance of high workloads and avoidance of data combinations that have caused previous failures (see Gifford and Spector [B16], Iyer and Velardi [B30]). Software in the operational environment may not exhibit the reduction in failure rate with execution time that is an implicit assumption in most estimation models (see Hecht and Hecht [B19]). Knowledge of the management policies is therefore essential for selection of an SR estimation procedure for the operational environment. Thus, the prediction of operational reliability from data obtained during test may not hold true during operations.

Another limitation of SR prediction models is their use in verifying ultra-high reliability requirements. For example, if a program executes successfully for *x* hours, there is a 0.5 probability that it will survive the next *x* hours without failing (see Fenton and Littlewood [B14]). Thus, to have the kind of confidence needed to verify a 10^{-9} requirement would require that the software execute failure-free for several billion hours. Clearly, even if the software had achieved such reliability, one could never assure that the requirement was met. The most reasonable verifiable requirement is in the 10^{-3} to 10^{-4} range.

Many ultra-reliable applications are implemented on relatively small, slow, inexpensive computers. Ultra-reliable applications, such as critical programs could be small (less than 1000 source lines of code) and execute infrequently during an actual mission. With this knowledge, it may be feasible to test the critical program segment on several faster machines, considerably reducing the required test time.

Furthermore, where very high reliability requirements are stated (failure probabilities $<10^{-6}$), they frequently are applicable to a software-controlled process together with its protective and mitigating facilities, and therefore they tend to be overstated if applicable to the process alone. An example of a protective facility is an automatic cutoff system for the primary process and reversion to analog or manual control. An example of a mitigation facility is an automatic sprinkler system that significantly reduces the probability of fire damage in case the software-controlled process generates excessive heat. If the basic requirement is that the probability of extensive fire damage should not exceed 10^{-6} probability of failure per day, and if both protecting and mitigating facilities are in place, it is quite likely that further analysis will show the maximum allowable failure rate for the software controlled process to be on the order of 10^{-3} probability of failure per day, and hence, within the range of current reliability estimation methods. Where the requirements for the software-controlled process still exceed the capabilities of the estimation methodology after allowing for protective and mitigating facilities, fault tolerance techniques may be applied. These may involve fault tolerance (see Hecht and Hecht [B19]) or functional diversity. An example of the latter is to control both temperature and pressure of steam generation, such that neither one of them can exceed safety criteria. The reduction in failure probability that can be achieved by software fault tolerance depends in a large measure on the independence of failure mechanisms for the diverse implementations. It is generally easier to demonstrate the independence of two diverse functions than it is to demonstrate the independence of two computer programs, and hence functional diversity is frequently preferred.

There is an exception to the premise that failure rate gets better with time and upgrades, as shown with the curve in Figure 1. This curve implies that developed and deployed software keeps getting better with upgrades without consideration for the state of the design and development environment from which the software originated. If the development platforms, software processes, and design environment remain at a constant high level of maturity, or the environment continuously improves in generating quality software, then the actual failure rate performance is expected to follow the curve in Figure 1. If this is not the case and the development environment deteriorates, and the development team loses capability and expertise over time, then the deployed software eventually reaches a point of diminishing return in which the failure rate increases over time where it is no longer possible to correct errors, or correcting errors leads to other errors. At this point, the software is no longer economical to maintain and must be redesigned or recoded.

4. Software reliability assessment and prediction procedure

4.1 General

This clause provides a recommended practice for the practitioner on how to do SR assessment and prediction and what types of analysis can be performed using the technique. It defines a step-by-step procedure for executing SR estimation and describes possible analysis using the results of the estimation procedure.

4.2 Software reliability procedure

The following 13-step procedure for assessing and predicting SR should be executed. Each step of the procedure should be tailored to the project and the current life-cycle phase.

a) Identify application
b) Specify the requirement
c) Allocate the requirement
d) Make a reliability risk assessment
e) Define errors, faults, and failures
f) Characterize the operational environment
g) Select tests
h) Select models
i) Collect data
j) Estimate parameters
k) Validate the model
l) Perform assessment and prediction analysis
m) Forecast additional test duration

4.2.1 Identify the application

All components of the software application (software product) that are subject to SR assessment or prediction should be identified.

NOTE 1—Example: A diesel generator contains a single computer with a single process that controls the generator. There is only the one component. The goal is to assess the reliability during operation.

NOTE 2—Example: The U.S. Space Shuttle contains many systems. A typical application may contain the following software systems: solid rocket boosters, external tank, orbit processing, navigation, control, launch processing, and reentry processing. Furthermore, each system consists of seven software components. The goal is to specify the reliability of each system and to allocate the reliability to each component.

NOTE 3—This step may be implemented within the Software Requirements Analysis Process in 7.1.2 of IEEE Std 12207™-2008 [B29].

4.2.2 Specify the reliability requirement

The reliability specification should be stated. Components may be combined (for reliability) in series, in parallel, or in combination. For the most conservative analysis and assessment of reliability, a series connection should be used. The reason is that if any component fails, the system fails. Note that this approach is used for prediction. Recognize that the actual connection of software components may not be in succession.

NOTE—This step may be implemented within the Software Requirements Analysis Process in 7.1.2 of IEEE Std 12207-2008 [B29].

4.2.3 Allocate the reliability requirement

Reliability requirements for each component should be allocated. A conservative allocation is a series connection of components. If such is chosen, the Equation (1), Equation (1.1), Equation (1.2), and Equation (1.3) should be carried out, using the following definitions:

i	system identification
R_T	*total* reliability of system
R_i	system i reliability (assumed equal for each system)
n	number of components in a system
c_i	criticality of system i
t_i	specified execution time of system i
T_i	mean time that system i would survive
ΔT_i	mean time that system i fails to meet execution time specification

The *total* reliability of system i with n components in series is given by Equation (1) as follows:

$$R_T = \prod_{1}^{n} R_i \tag{1}$$

Using Equation (1) and assuming that the system reliabilities R_i are equal, the reliability of each system is given by Equation (1.1) as follows:

$$R_i = R_T^{(1/n)} \tag{1.1}$$

After making the prediction provided by Equation (1.1), estimate the mean time that system i could survive from Equation (1.2) as follows:

$$T_i = R_i\, t_i = R_T^{(1/n)}\, t_i \tag{1.2}$$

Furthermore, account for the extent to which the prediction given by Equation (1.2) fails to meet the requirement specification for mean survival time of components. This calculation is provided by Equation (1.3) as follows:

$$\Delta T_i = t_i - T_i \tag{1.3}$$

Consider a hypothetical system consisting of $n = 7$ software components, with criticality c_i and execution time t_i listed in Table 1. The systems have the *total* reliability requirements, R_T, respectively, listed in Table 1. In this example, the *total* reliability R_T is given. In other examples, the objective could be to calculate R_T from Equation (1).

Table 1 also shows two results of the reliability allocation exercise. Namely, the assessed reliability of each system R_i and *the mean time that component i fails to meet execution time requirement* ΔT_i.

Table 1—Hypothetical system example

i	R_T	c_i	t_i (h)	R_i	ΔT_i (h)
1	0.90	5	12	0.9851	0.1793
2	0.91	4	15	0.9866	0.2007
3	0.92	3	20	0.9882	0.2368
4	0.93	2	30	0.9897	0.3094
5	0.94	1	60	0.9912	0.5280
6	0.95	6	10	0.9927	0.0730
7	0.96	7	9	0.9942	0.0523

It is important to see whether there is a "correct relationship" between ΔT_i and the criticality c_i of system i. For example, looking at Figure 2, ΔT_i decreases as c_i increases. This is the correct relationship. If the plot had been otherwise, the software developer should consider examining the SRE process to determine why the correct relationship was not obtained.

NOTE—This step may be implemented within the Software Architectural Design Process in 7.1.3 of IEEE Std 12207-2008 [B29].

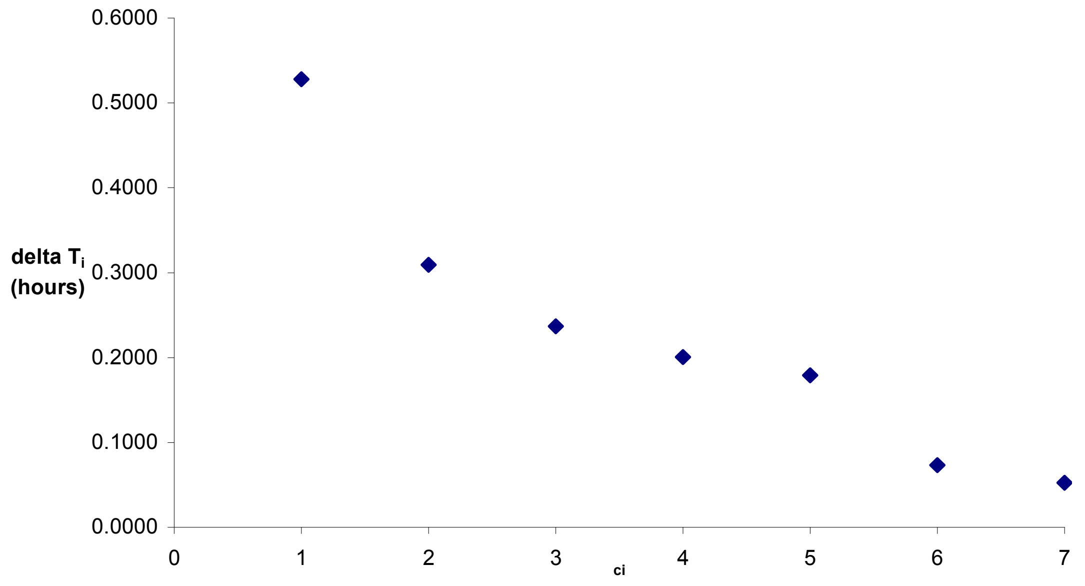


Figure 2—Time that system fails to meet specified execution time requirement, delta versus system criticality c_i

4.2.4 Make a reliability risk assessment

A reliability risk assessment should be based on the risk to reliability due to software defects or errors caused by requirements and requirements changes. The method to ascertain risk based on the number of requirements and the impact of changes to requirements is inexact, but nevertheless, necessary for early design assessments of large scale systems. It is probably not necessary for smaller systems and modules. Risk assessment is performed by calculating two quantities of the risk, the risk for remaining failures, and risk for the time to next failure. The methods to calculate these two risk metrics for risk assessment are provided in 5.3.1.12.2.

NOTE—This step may be implemented within the Risk Management Process in 6.3.4 of IEEE Std 12207-2008 [B29].

4.2.5 Define errors, faults, and failures

Project specific definitions for error, failures, and faults should be provided within the constraints of the definitions in 2.4, 2.6, and 2.9, respectively. These definitions are usually negotiated by the testers, developers, and users. These definitions should be agreed upon prior to the beginning of test. There are often commonalities in the definitions among similar products (i.e., most people agree that a software fault that stops all processing is a failure). The important consideration is that the definitions be consistent over the life of the project. There are a number of considerations relating to the interpretation of these definitions. The analyst must determine the answers to the following questions as they relate to errors, faults, and failures:

- Is an error, failure, or fault counted if it is decided not to seek out and remove the cause of the error, failure, or fault?
- Are repeated errors, failures, or faults counted? If so, how are they counted (e.g., one pattern failure represents several failure occurrences of the same failure type)?
- What is an error, failure, or fault in a fault-tolerant system?
- Is a series of errors, failures, or faults counted if it is triggered by data degradation?

NOTE 1—A discussion of each of these considerations is provided in pp. 77–85 in *Software Reliability: Measurement, Prediction, Application* [B54].

NOTE 2—Projects need to classify failures by their severity. An example classification is provided in the NOTE of 2.8. The classes are usually distinguished by the criticality of the failure to system safety and/or cost of fault correction. It is desirable to consider failure severity by type.

NOTE 3—For some projects, there appears to be relative consistency of failures. For example, if 10% of the failures occurring early in test fall in a particular severity class, about the same percentage will be expected to be found in that class late in test.

NOTE 4—Time measurement based on CPU time or failure counts based on CPU time for failure data are preferred. However, there are approximating techniques, if the direct measurement of CPU time is not available (see pp. 156–158 in *Software Reliability: Measurement, Prediction, Application* [B54]). Failure times should be recorded in execution time or failure counts per execution time interval. However, should execution time not be available, elapsed clock time is a satisfactory approximation. When failure times are collected from multiple machines operating simultaneously, intervals between failures should be counted by considering all machines operating simultaneously. If the machines have different average instruction execution rates, execution times (reciprocal of execution rate) should be adjusted by using the average of the average execution rate.

NOTE 5—This step may be implemented within the Measurement Process in 6.3.7 of IEEE Std 12207-2008 [B29].

4.2.6 Characterize the operational environment

The operational environment should be characterized. This characterization should be based on the following three aspects:

- System configuration—the arrangement of the system's components. Software-based systems include hardware as well as software components.
- System evolution.
- System operational profile.

NOTE 1—The purpose of determining the system configuration is twofold:

- To determine how to allocate system reliability to component reliabilities
- To determine how to combine component reliabilities to establish system reliability

NOTE 2—In modeling SR, it is necessary to recognize that systems frequently evolve as they are tested. That is, new code or even new components are added. Special techniques for dealing with evolution are provided in pp. 166–176 in *Software Reliability: Measurement, Prediction, Application* [B54].

NOTE 3—A system may have multiple operational profiles or operating modes. For example, a space vehicle may have ascent, on-orbit, and descent operating modes. Operating modes may be related to the time of operation, installation location, and customer or market segment. Reliability should be assessed separately for different modes if they are significantly different.

NOTE 4—This step may be implemented within the Stakeholder Requirements Definition Process in 6.4.1 of IEEE Std 12207-2008 [B29].

4.2.7 Select tests

Tests and test cases should be selected to reflect how the system will actually be used. Test selection criteria and guidance on the appropriateness of one test over another depending on certain situations (test conditions or use conditions) may be useful.

The tester should select one of the following approaches:

- The test duplicates the actual operational environments (as closely as possible).
- Testing is conducted under severe conditions for extended periods of time.
- The reliability modeling effort must take into account the approach taken by the test team to expose faults so that accurate assessments can be made.

NOTE—This step may be implemented within the System Qualification Testing Process in 6.4.6 of IEEE Std 12207-2008 [B29].

4.2.8 Select models

One or more SR models should be selected and applied. Model selection criteria and guidance on the appropriateness of one model over another depending on certain situations may be useful.

NOTE—This step may be implemented within the Decision Management and Measurement Processes in 6.3.3 and 6.3.7, respectively, of IEEE Std 12207-2008 [B29].

The models described in Clause 5 provided proven accurate results in specific environments, but there is no guarantee that these models will be suitable in all environments; therefore, it is necessary that users compare several models applicable to the user's environment prior to final selection.

4.2.9 Collect data

Data collection should be sufficient to determine if the objectives of the SR effort have been met.

In setting up a reliability program, the following goals should be achieved:

- Clearly defined data collection objectives
- Collection of appropriate data (i.e., data geared to making reliability assessments and predictions)
- Review of the collected data promptly to see ensure that it meets the objectives

A process should be established addressing each of the following steps, which are further described in 5.6.1:

a) Establish the objectives
b) Set up a plan for the data collection process
c) Apply tools
d) Provide training
e) Perform trial run
f) Implement the plan
g) Monitor data collection
h) Evaluate the data as the process continues
i) Provide feedback to all parties

NOTE—This step may be implemented within the Measurement Process in 6.3.7 of IEEE Std 12207-2008 [B29].

4.2.10 Estimate parameters

A parameter estimation method should be selected. Three common methods of parameter estimation exist: method of moments, least squares, and maximum likelihood. Each of these methods has useful attributes. Maximum likelihood estimation is the recommended approach.

NOTE 1—This step may be implemented within the Measurement Process in 6.3.7 of IEEE Std 12207-2008 [B29].

NOTE 2—A full treatment of parameter estimation is provided by Farr [B11] and in *Software Reliability: Measurement, Prediction, Application* [B54] and *Software Engineering: Design, Reliability, and Management* [B69]. All of the SRE tools described in Annex B perform parameter estimation, using one or more of these methods.

4.2.11 Validate the model

The chosen model or models should be validated. The model validation should also include checking the data collection and parameter estimation methods.

NOTE 1—Several considerations are involved in properly validating a model for use on a given production project. First, it is necessary to deal with the assumptions of the model under evaluation. Choosing appropriate failure data items and relating specific failures to particular intervals of the life-cycle or change increments often facilitate this task (see Schneidewind and Keller [B59]). Depending on the progress of the *production* project, the model validation data source should be selected from the following, listed in the order of preference:

a) Production project failure history (if project has progressed sufficiently to produce failures)

b) Prototype project employing similar products and processes as the production project

c) Prior project employing similar products and processes as the production project (reference Annex B)

NOTE 2—Using one of these data sources, the analyst should execute the model several times within the failure history period and then compare the model output to the actual subsequent failure history using one of the following:

a) Predictive validity criteria (5.2.2)

b) A traditional statistical goodness-of-fit test [e.g., chi-square or Kolmogorov−Smirnov (K-S)]

NOTE 3—It is important that a model be continuously rechecked for validation, even after selection and application, to ensure that the fit to the observed failure history is still satisfactory. In the event that a degraded model fit is experienced, alternate candidate models should be evaluated using the procedure of 4.2.7 through 4.2.10.

NOTE 4—This step may be implemented within the Measurement Process in 6.3.7 of IEEE Std 12207-2008 [B29].

4.2.12 Perform assessment and prediction analysis

Once the data has been collected and the model parameters estimated, the analyst is ready to perform the appropriate analysis. This analysis should be to assess the current reliability of the software, predict the number of failures remaining in the code, or predict a test completion date.

4.2.12.1 Recommended analysis practice

Subclauses 4.2.12.2 and 4.2.12.3 provide details of analysis procedures that are supported by SRE technology.

4.2.12.2 Assess current reliability

Reliability assessments in test and operational phases basically follow the same procedure. However, there is a difference. During the test phase, software faults should be removed as soon as the corresponding software failures are detected. As a result, reliability growth (e.g., decreasing failure rate over time) should be observed. However, in the operational phase, correcting a software fault may involve changes in multiple software copies in the customers' sites, which, unless the failure is catastrophic, is not always done until the next software release. Therefore, the software failure rate may not be decreasing.

4.2.12.3 Predict achievement of a reliability goal

The date at which a given reliability goal can be achieved is obtainable from the SR modeling process illustrated in Figure 3. As achievement of the reliability target approaches, the adherence of the actual data to the model predictions should be reviewed and the model corrected, if necessary. Refer to Annex C.

NOTE—This step may be implemented within the Decision Management and Measurement Processes in 6.3.3 and 6.3.7, respectively, of IEEE Std 12207-2008 [B29].

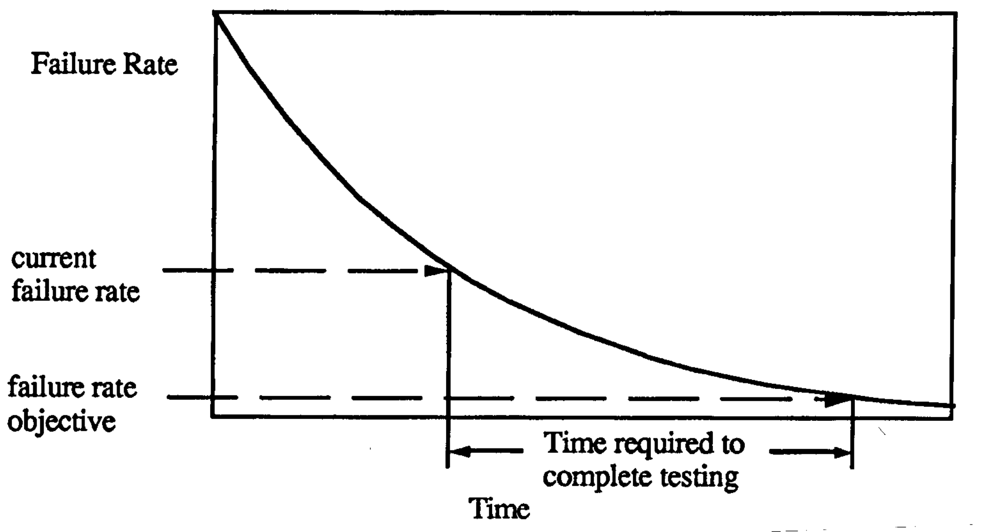


Figure 3—Example SR measurement application

4.2.13 Forecast additional test duration

Additional test duration should be predicted if the initial and objective failure intensities and the parameters of the model are known. (These are identified for each model in Clause 5.)

For the Musa basic exponential model, Equation (2) follows:

$$\Delta t = \frac{\nu_0}{\lambda_0} \ln\left(\frac{\lambda_0}{\lambda_F}\right) \tag{2}$$

where

Δt	is the test duration in CPU hours
ν_0	is the total failures parameter of the model
λ_0	is the initial failure intensity
λ_F	is the objective failure intensity

The equivalent formula for the Musa/Okumoto logarithmic Poisson model is given in Equation (3) as follows:

$$\Delta t = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_0}\right) \tag{3}$$

where

θ	is the failure intensity decay parameter
λ_0	is the initial failure intensity
λ_F	is the objective failure intensity

5. Software reliability estimation models

5.1 Introduction

There are many ways to develop an SR model: a) describe it as a stochastic process; b) relate it to a Markov model; c) define the probability density or distribution function; or d) specify the hazard function. There are the following three general classes of SR prediction models: exponential non-homogeneous Poisson process (NHPP) models, non-exponential NHPP models, and Bayesian models. Subclauses 5.1.1, 5.1.2, and 5.1.3 describe the characteristics of each class.

5.1.1 Exponential NHPP models

Exponential NHPP models use the stochastic process and the hazard function approach. The hazard function, $z(t)$, is generally a function of the operational time, t. Several different derivations of $z(t)$ are given by Shooman [B71]. The probability of success as a function of time is the reliability function, $R(t)$, which is given by Equation (4) as follows:

$$R(t) = \exp\left[-\int_0^t z(y)dy\right] \tag{4}$$

Sometimes reliability is expressed in terms of a single parameter: mean time-to-failure (MTTF). MTTF is given by Equation (5) as follows:

$$\text{MTTF} = \int_0^\infty R(t)dt \tag{5}$$

On occasion the reliability function may be of such a form that MTTF is not defined. The hazard function (or failure intensity in pp. 11–18 of *Software Reliability: Measurement, Prediction, Application* [B54]) or the reliability function can be used in this case. The hazard function can be constant or can change with time.

Representative models in this class include Schneidewind's model (described in 5.3.1), Shooman's model, Musa's basic model, Jelinski and Moranda's model (described in A.4), and the generalized exponential model (described in 5.3.2). Model objectives, assumptions, parameter estimates, and considerations for using the model are described in the appropriate clause.

5.1.2 Non-exponential NHPP models

Non-exponential NHPP models also use the stochastic process and the hazard function approach. They are applicable after completion of testing. Early fault corrections have a larger effect on the failure intensity function than later ones.

Representative models in this class include Duane's model (described in A.2), Brooks and Motley's binomial and Poisson models, Yamada's s-shaped model (described in A.3), and Musa and Okumoto's logarithmic Poisson (described in 5.4). The assumptions and format of the model, its estimates for model fitting, and considerations for employing the model are described in the appropriate clause.

5.1.3 Bayesian models

NHPP models assume that the hazard function is directly proportional to the number of faults in the program, and hence the reliability is a function of this fault count. The Bayesian approach argues that a program can have many faults in unused sections of the code and exhibit a higher reliability than software with only one fault in a frequently exercised section of code. Representative models of this class are those developed by Littlewood [B41].

5.2 Criteria for model evaluation

The following criteria should be used for conducting an evaluation of SR models in support of a given project:

- Future predictive accuracy: Accuracy of the model in making predictions *beyond* the time span of the collected data (i.e., comparison of *future* predictions with *future* observed data).
- Historical predictive validity (i.e., comparison of retrospective predictions against past observed data).
- Generality: Ability of a model to make accurate predictions in a variety of operational settings (e.g,, real-time, Web applications).
- Insensitivity to noise: The ability of the model to produce accurate results in spite of errors in input data and parameter estimates.

5.2.1 Future predictive accuracy

Reliability prediction is primarily about the *future*, not the past. It is not possible to change the past; it is possible to influence the future. Many model validation methods are based in *retrospective* prediction, comparing historical failure data with predictions in the time span of the observed data. The only thing that can be proved by this method is that the past can be predicted. Observed data *is* important for estimating the parameters of the model. Once the parameters have been estimated, the model is used to predict the future reliability of the software. Then, the future predictions are compared with observed *future* data. Then, a decision is made as to whether the accuracy is satisfactory for the application using, for example, a goodness-of-fit test. If the accuracy is unsatisfactory, other models could be evaluated, using the previous procedure.

5.2.2 Model predictive validity

To compare a set of models against a given set of failure data, the *fitted* models are examined to determine which model is in best agreement with the observed data. This means that the model with the least difference between the retrospective prediction and the actual data is considered the best fit. Best fit can be measured by the criterion of minimum relative error.

NOTE 1—A *fitted* model is one that has had its parameters estimated from the observed data.

NOTE 2—One approach to determine if the *fitted* model *retrospectively* predicts the observed data within a reasonable relative error, such as $\frac{\text{actual - predicted}}{\text{predicted}} \leq 20\%$.

NOTE 3—A second method involves making a goodness-of-fit test between the prediction and the observed data. The literature on goodness-of-fit tests is quite extensive; the chi-square and K-S tests are two examples (see *Introduction to Mathematical Statistics* [B21]).

NOTE 4—In addition to these techniques for assessing model fit, the following three measures may be used to compare model predictions with a set of failure data:

- Accuracy (5.2.2.1)
- Bias (5.2.2.2)
- Trend (5.2.2.3)

5.2.2.1 Accuracy

One way to measure prediction accuracy is by the prequential likelihood (PL) function (see Littlewood et al. [B42]). Let the observed failure data be a sequence of times $t_1, t_2, \ldots, t_{i-1}$ between successive failures. The objective is to use the data to predict the future unobserved failure times T_i. More precisely, we want a good estimate of $F_i(t)$, defined as $\Pr(T_i < t)$, (i.e., the probability that T_i is less than a variable t). The prediction distribution $\tilde{F}_i(t)$ for T_i based on $t_1, t_2, \ldots, t_{i-1}$ will be assumed to have a pdf (probability density function).

$$\tilde{f}_i(t) = \frac{d}{dt}\tilde{F}_i(t) \tag{6}$$

For such one-step-ahead forecasts of $T_{i+1}, \cdots, T_{i+n}$, the PL is given by Equation (7) as follows:

$$\mathrm{PL}_n = \prod_{i=j+1}^{j+n} \tilde{f}_i(t_i) \tag{7}$$

Since this measure is usually very close to zero, its natural logarithm is frequently used for comparisons. Given two competing SR models A and B, the PL ratio is given by Equation (8) as follows:

$$\mathrm{PLR}_n = \frac{\ln \mathrm{PL}_n(A)}{\ln \mathrm{PL}_n(B)} \tag{8}$$

The ratio represents the likelihood that one model will give more accurate forecasts than the other model. If $\mathrm{PLR}_i \to \infty$ as $n \to \infty$, model A is favored over model B.

5.2.2.2 Bias

A model is considered *biased* if it predicts values that are consistently greater than the observed failure data, or consistently less than the observed data. To measure the amount of a model's bias, compute the maximum vertical distance (i.e., the K-S distance) (see *Introduction to Mathematical Statistics* [B21]) between cumulative distribution functions of the observed and predicted data. The criterion for goodness of fit is the maximum vertical distance between the plots of the two functions. For a good fit, this distance must be smaller than that given in the K-S tables for a given value of α.

5.2.2.3 Trend

Trend is measured by whether a prediction quantity r_i is monotonically increasing or decreasing over a series of time intervals i, as given by Equation (9):

$$T(i) = \frac{1}{i}\sum_{j=1}^{i} r_j \tag{9}$$

If $T(i)$ is increasing for a quantity like the time to next failure, it is indicative of reliability growth; on the other hand, if $T(i)$ is decreasing, it is indicative of a serious reliability problem whose cause must be investigated.

5.2.3 Quality of assumptions

The assumptions upon which an SR model is based should be stated.

NOTE—Common assumptions made in the SR models are as follows:

- Test inputs randomly encounter faults.
- The effects of all failures are independent.
- The test space covers the program use space.
- All failures are observed when they occur.
- Faults are immediately removed when failures are observed.
- The failure rate is proportional to the number of remaining faults.

If an assumption is testable, it should be supported by data to validate the assumption. If an assumption is not testable, it should be examined for reasonableness.

SR data may contain "noise" that impairs predictive accuracy. The most common source of noise is that software failure data is recorded in calendar time rather than in software execution time. Even when software failures are tracked carefully based on execution time, the software testing process may be inconsistent with the model assumptions (e.g., the software is not tested randomly). Therefore, a model should demonstrate its validity in real-word situations.

5.3 Initial models

The following models should be considered as initial models for SR prediction:

- The Schneidewind model
- The generalized exponential model
- The Musa/Okumoto logarithmic Poisson model

If these models cannot be validated (see 4.2.7*)* or do not meet the criteria defined in 5.2 for the project, alternative models that are described in Annex A can be considered. In addition, practitioners may also consider using a different set of initial models, if other models fit better due to specific considerations.

5.3.1 Schneidewind model

5.3.1.1 Schneidewind model objectives

The objectives of this model should be to predict the following software product attributes:

—	$F(t_1, t_2)$	Predicted failure count in the range $[t_1, t_2]$
—	$F(\infty)$	Predicted failure count in the range $[1, \infty]$; maximum failures over the life of the software
—	$F(t)$	Predicted failure count in the range $[1, t]$
—	$p(t)$	Fraction of remaining failures predicted at time t

— $Q(t)$	Operational quality predicted at time t; the complement of $p(t)$; the degree to which software is free of remaining faults (failures)
— $r(t)$	Remaining failures predicted at time t
— $r(t_t)$	Remaining failures predicted at total test time t_t
— t_t	Total test time predicted for given $r(t_t)$
— $T_F(t)$	Time to next failure(s) predicted at time t
— $T_F(t_t)$	Time to next failure predicted at total test time t_t

5.3.1.2 Parameters used in the predictions

The parameters are as follows:

— α	Failure rate at the beginning of interval S
— β	Negative of derivative of failure rate divided by failure rate (i.e., relative failure rate)
— r_c	Critical value of remaining failures; used in computing relative criticality metric (RCM) $r(t_t)$
— S	Starting interval for using observed failure data in parameter estimation
— t	Cumulative time in the range $[1, t]$; last interval of observed failure data; current interval
— T	Prediction time
— t_m	Mission duration (end time-start time); used in computing RCM $T_F(t_t)$

5.3.1.3 Observed quantities

The observed quantities are as follows:

— t_t	Total test time
— T_{ij}	Time since interval i to observe number of failures F_{ij} during interval j; used in computing MSE_T
— X_k	Number of observed failures in interval k
— X_i	Observed failure count in the range $[1, i]$
— X_{s-1}	Observed failure count in the range $[1, s-1]$
— $X_{s,l}$	Observed failure count in the range $[i, s-1]$
— $X_{s,i}$	Observed failure count in the range $[s, i]$
— $X_{s,t}$	Observed failure count in the range $[s, t]$
— $X_{s,t1}$	Observed failure count in the range $[s, t_1]$
— X_t	Observed failure count in the range $[1, t]$
— X_{t1}	Observed failure count in the range $[1, t_1]$

The basic philosophy of this model is that as testing proceeds with time, the failure detection process changes. Furthermore, recent failure counts are usually of more use than earlier counts in predicting the future. Three approaches can be employed in utilizing the failure count data, i.e., number of failures detected per unit of time. Suppose there are t intervals of testing and f_i failures were detected in the i^{th} interval, one of the following can be done:

— Utilize all of the failures for the t intervals.

- Ignore the failure counts completely from the first $s - 1$ time intervals ($1 \le s \le t$) and only use the data from intervals s through m.
- Use the cumulative failure count from intervals 1 through $s - 1$, i.e., $F_{s-1} = \sum_{i=1}^{s-1} f_i$.

The first approach should be used when it is determined that the failure counts from all of the intervals are useful in predicting future counts. The second approach should be used when it is determined that a significant change in the failure detection process has occurred and thus only the last $t - s + 1$ intervals are useful in future failure forecasts. The last approach is an intermediate one between the other two. Here, the combined failure counts from the first $s - 1$ intervals and the individual counts from the remaining are representative of the failure and detection behavior for future predictions.

5.3.1.4 Schneidewind model assumption

The assumption to the Schneidewind model is that the number of failures detected in one interval is independent of the failure count in another.

NOTE—In practice, this assumption has not proved to be a factor in obtaining prediction accuracy.

- Only new failures are counted.
- The fault correction rate is proportional to the number of faults to be corrected.
- The software is operated in a similar manner as the anticipated operational usage.
- The mean number of detected failures decreases from one interval to the next.
- The rate of failure detection is proportional to the number of failures within the program at the time of test. The failure detection process is assumed to be an NHPP with an exponentially decreasing failure detection rate. The rate is of the form $f(t) = \alpha e^{-\beta(t-s+1)}$ for the t^{th} interval where $\alpha > 0$ and $\beta > 0$ are the parameters of the model.

5.3.1.5 Schneidewind model structure

Two parameters are used in this model: α, which is the failure rate at time $t = 0$, and β, which is a proportionality constant that affects the failure rate over time (i.e., small β implies a large failure rate; large β implies a small failure rate). In these estimates, t is the last observed count interval, s is an index of time intervals, X_k is the number of observed failures in interval k, X_{s-1} is the number of failures observed from 1 through $s - 1$ intervals, $X_{s,\,t}$ is the number of observed failures from interval s through t, and $X_t = X_{s-1} + X_{s,\,t}$. The likelihood function is then developed as Equation (10) as follows:

$$\begin{aligned}\log L = X_t\left[\log X_t - 1 - \log\left(1-e^{-\beta t}\right)\right] \\ +X_{s-1}\left[\log\left(1-e^{-\beta(s-1)}\right)\right] \\ +X_{s,t}\left[\log\left(1-e^{-\beta}\right)\right] - \beta\sum_{k=0}^{t-s}(s+k-1)X_{s+k}\end{aligned} \tag{10}$$

This function is used to derive the equations for estimating α and β for each of the three approaches described earlier. In the equations that follow, α and β are estimates of the population parameters.

Parameter estimation: Approach 1

Use all of the failure counts from interval 1 through t (i.e., $s = 1$). Equation (11) and Equation (12) are used to estimate β and α, respectively, as follows:

$$\frac{1}{e^{\beta}-1}-\frac{t}{e^{\beta t}-1}=\sum_{k=0}^{t-1}k\frac{X_{k+1}}{X_t} \tag{11}$$

$$\alpha=\frac{\beta X_t}{1-e^{-\beta t}} \tag{12}$$

Parameter estimation: Approach 2

Use failure counts only in intervals *s* through *t* (i.e., $1 \le s \le t$). Equation (13) and Equation (14) are used to estimate β and α, respectively, as follows. (Note that Approach 2 is equivalent to Approach 1 for $s = 1$.)

$$\frac{1}{e^{\beta}-1}-\frac{t-s+1}{e^{\beta(t-s+1}-1}=\sum_{k=0}^{t-s}k\frac{X_{k+s}}{X_{s,t}} \tag{13}$$

$$\alpha=\frac{\beta X_{s,t}}{1-e^{-\beta(t-s+1)}} \tag{14}$$

Parameter estimation: Approach 3

Use cumulative failure counts in intervals 1 through $s - 1$ and individual failure counts in intervals *s* through *t* (i.e., $2 \le s \le t$). This approach is intermediate to Approach 1 that uses all of the data and Approach 2 that discards “old” data. Equation (15) and Equation (16) are used to estimate β and α, respectively, as follows. (Note that Approach 3 is equivalent to Approach 1 for $s = 2$.)

$$\frac{(s-1)X_{s-1}}{e^{\beta(s-1)}-1}+\frac{X_{s,t}}{e^{\beta}-1}-\frac{tX_t}{e^{\beta m}-1}=\sum_{k=0}^{t-s}(s+k-1)X_{s+k} \tag{15}$$

$$\alpha=\frac{\beta X_t}{1-e^{-\beta t}} \tag{16}$$

5.3.1.6 Criterion for optimally selecting failure data

The first step in identifying the optimal value of *s* (s^*) should be to estimate the parameters α and β for each value of *s* in the range $[1, t]$ where convergence can be obtained (see Schneidewind [B58], [B60], [B62]). Then the *mean square error* (MSE) criterion should be used to select s^*, the failure count interval that corresponds to the minimum MSE between predicted and actual failure counts (MSE_F), *time to next failure* (MSE_T), or *remaining failures* (MSE_r), depending on the type of prediction. Once α, β, and *s* are estimated from observed counts of failures, the predictions can be made. The reason MSE is used to evaluate which triple (α, β, *s*) is best in the range $[1, t]$ is that research has shown that because the product and process change over the life of the software, old failure data (i.e., $s = 1$) are not as representative of the current state of the product and process as the more recent failure data (i.e., $s > 1$).

Some examples of applying the MSE are given in 5.3.1.6.1, 5.3.1.6.2, and 5.3.1.6.3.

5.3.1.6.1 MSE criterion for remaining failures, maximum failures, and total test time [for *Method 2* and *Method 1* (*s = 1*)]

$$\mathrm{MSE}_s = \frac{\sum_{i=s}^{t} [F(i) - X_i]}{t - s + 1} \tag{17}$$

where

$$F(i) = \frac{\alpha}{\beta}\Big[1 - \exp\big(-\beta(i - s + 1)\big)\Big] + X_{s-1} \tag{18}$$

5.3.1.6.2 MSE criterion for time to next failure(s) [for *Method 2* and *Method 1* (*s = 1*)]

MSE criterion for *time to next failure*(s) is given in Equation (19) as follows:

$$\mathrm{MSE}_r = \frac{1}{J - s} \sum_{i=s}^{J-1} \left[\frac{\log\left(\dfrac{\alpha}{\alpha - \beta(X_{s,i} - F_{ij}} \right)}{\beta(i - s + 1)} - T_{ij} \right]^2 \tag{19}$$

5.3.1.6.3 MSE criterion for failure counts [for *Method 2* and *Method 1* (*s = 1*)]

$$\mathrm{MSE}_F = \frac{1}{t - s + 1} \sum_{i=s}^{t} \left[\frac{\alpha}{\beta\big(1 - \exp(-\beta(i - s + 1))\big)} - X_{s,i} \right]^2 \tag{20}$$

Thus, for each value of *s*, compute MSE using either Equation (17), Equation (19), or Equation (20), as appropriate. Choose *s* equal to the value for which MSE is smallest. The result is an optimal triple (α, β, *s*) for your data set. Then apply the appropriate approach to your data.

5.3.1.7 Schneidewind model limitations

The limitations of the model are the following:

— It does not account for the possibility that failures in different intervals may be related.

— It does not account for repetition of failures.

— It does not account for the possibility that failures can increase over time as the result of software modifications.

These limitations should be ameliorated by configuring the software into versions that represent the previous version plus modifications. Each version represents a different module for reliability prediction purposes: the model is used to predict reliability for each module. Then, the software system reliability is predicted by considering the *N* modules to be connected in series (i.e., worst-case situation), and computing the MTTF for *N* modules in series (see Schneidewind and Keller [B59]).

5.3.1.8 Schneidewind model data requirements

The only data requirements are the number of failures, f_i, $i = 1, \ldots, t$, per testing period.

A reliability database should be created for several reasons: Input data sets will be rerun, if necessary, to produce multiple predictions rather than relying on a single prediction; reliability predictions and assessments could be made for various projects; and predicted reliability could be compared with actual reliability for these projects. This database will allow the model user to perform several useful analyses: to see how well the model is performing; to compare reliability across projects to see whether there are development factors that contribute to reliability; and to see whether reliability is improving over time for a given project or across projects.

5.3.1.9 Schneidewind model applications

The major model applications are described as follows. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

- *Prediction:* Predicting future failures, fault corrections, and related quantities described in 5.3.1.6.1.
- *Control:* Comparing prediction results with predefined goals and flagging software that fails to meet those goals.
- *Assessment:* Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process). The formulation of test strategies is also part of assessment. Test strategy formulation involves the determination of priority, duration, and completion date of testing, allocation of personnel, and allocation of computer resources to testing.
- *Rank reliability:* Rank reliability on the basis of the parameters α and β, without making a prediction. Figure 4 through Figure 7 show four projects labeled IT35MSE-T. IT35MSE-F, I20, and I25. The first two are NASA Goddard Space Flight Center (GSFC) projects and the second two are NASA Space Shuttle operational increments (OI) (i.e., releases). For the four projects, the values of α and β are 4.408529, 3.276461, 0.534290, 0.221590, and 0.088474, 0.068619, 0.060985, 0.027882, respectively. The parameters α and β arethe initial value of the failure rate and the rate at which the failure rate changes, respectively. It is desirable to have the largest ratio of β to α for high reliability because this will yield the fastest decrease in failure rate combined with the smallest initial failure rate. For the four projects, these ratios are 0.0201, 0.0209, 0.1141, 0.1258, respectively, from the lowest to the highest reliability. Thus, after estimating α and β, using a tool such as Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) or Computer Aided Software Reliability Estimation (CASRE), rank reliability without or before making predictions. This procedure is useful for doing an initial reliability screening of projects to determine, for example, which projects might meet reliability specifications and which projects require reliability improvements. Figure 4 shows that OI25 starts with the lowest failure rate (α) and IT35MSE-T starts with the highest. In contrast, in Figure 5, OI25 has the lowest failure rate change (β) and IT35MSE-T has the highest. However, in this case, the lower initial failure rate outweighs the faster rate of change to produce the ordering of reliability shown in Figure 6 and Figure 7.

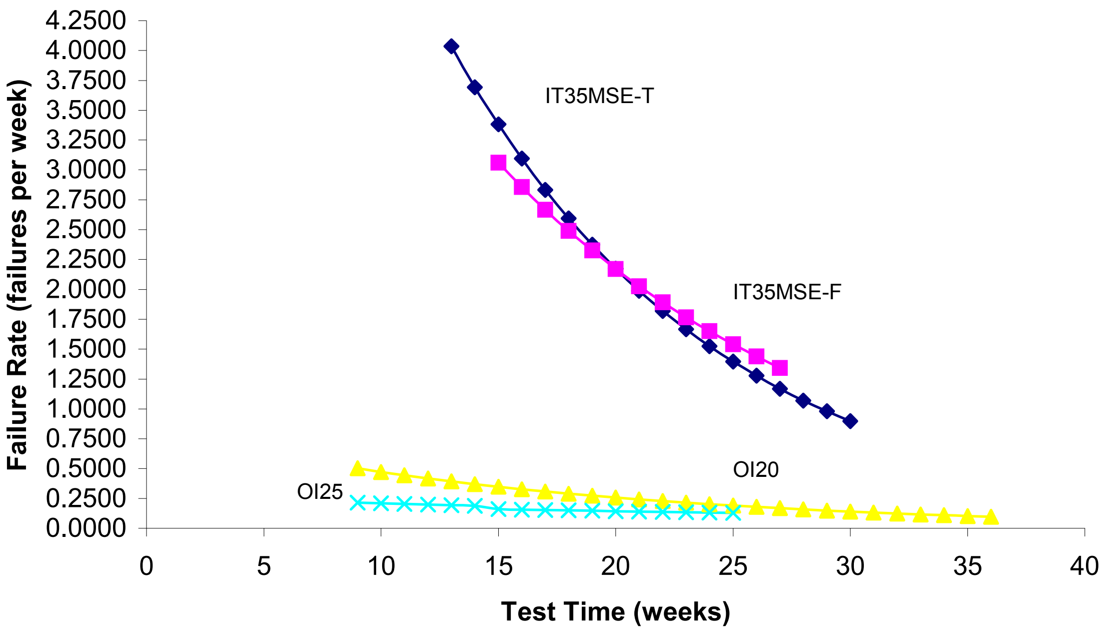


Figure 4—Failure rate for projects IT35MSE-T, IT35MSE-F, OI20, and OI25

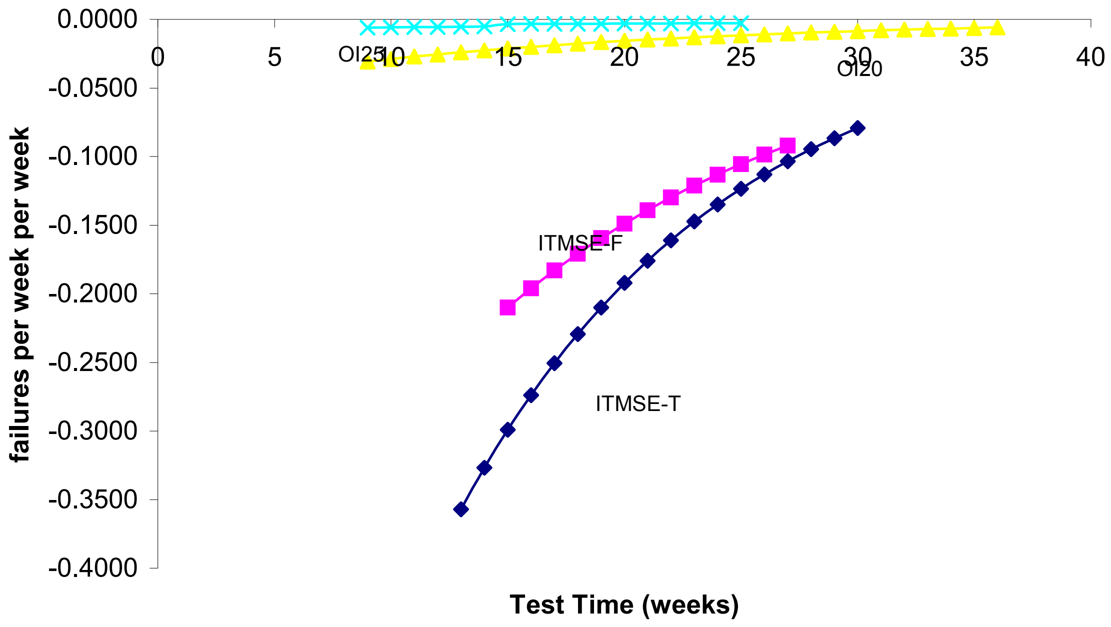


Figure 5—Rate of change in failure rate for projects ITMSE-T, ITMSE-F, OI20, and OI25

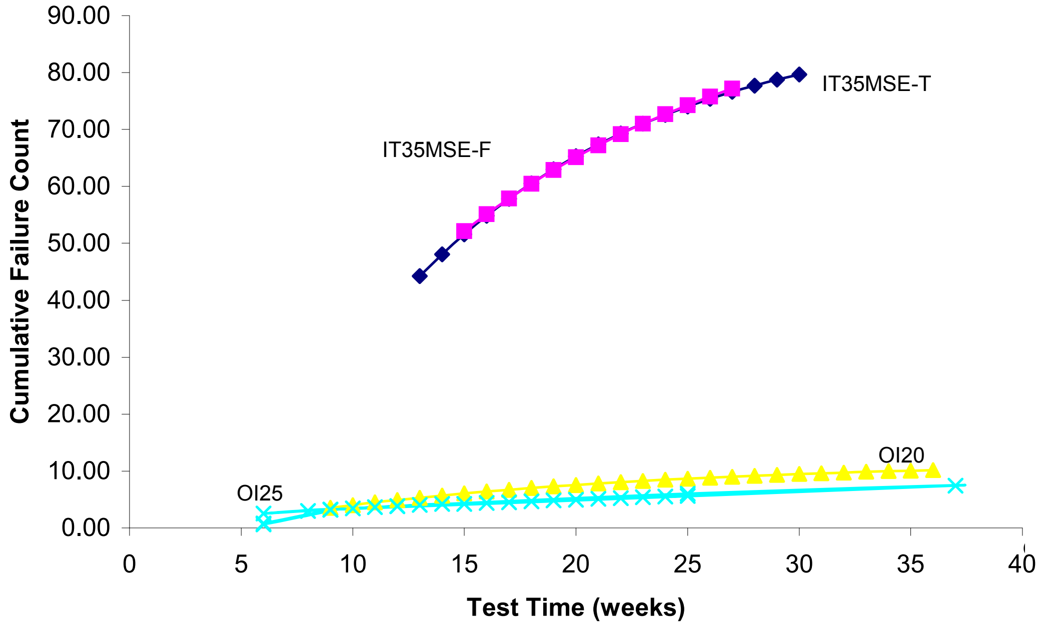


Figure 6—Cumulative failures versus test time for projects IT35MSE-T, IT35MSE-F, OI20, and OI25

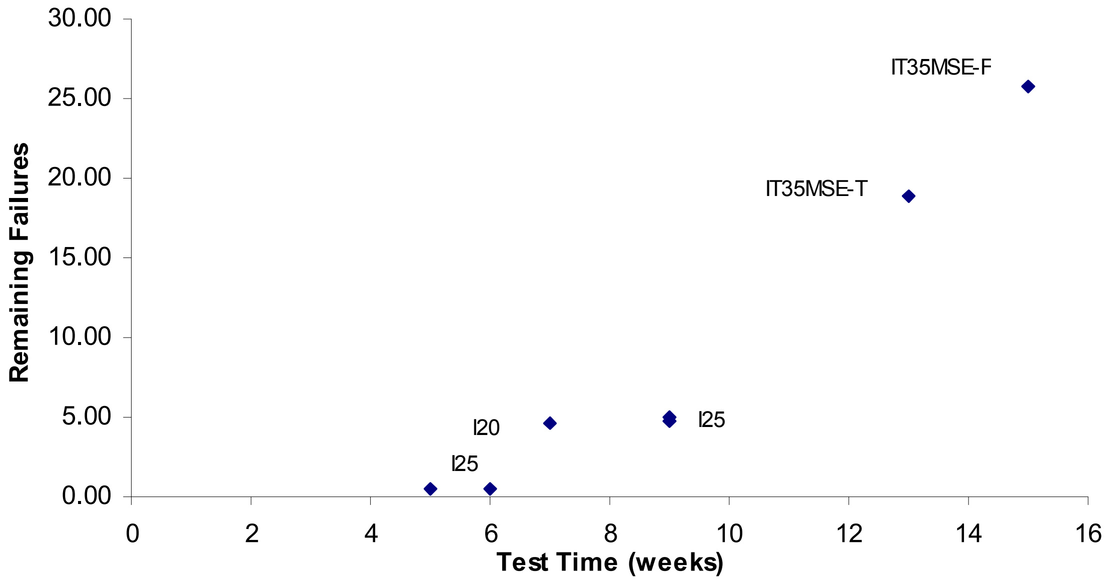


Figure 7—Remaining failures versus test time for projects IT35MSE-T, IT35MSE-F, OI20, and OI25

5.3.1.10 Reliability predictions

Using the optimal triple (α, β, s), which was given in 5.3.1.5, various reliability predictions should be computed. The Approach 2 equations [Equation (13) and Equation (14)] are given where t or T, as appropriate, is greater than or equal to s. For Approach 2, set $s = 1$.

Predict *time to detect a total of F_t failures* (i.e., time-to-next F_t failures) when the current time is t and $X_{s,t}$ failures have been observed as follows in Equation (21):

$$T_f(t) = \log\left[\frac{\alpha}{(\alpha - \beta(F_t + X_{s,t}))/\beta}\right] - (t - s + 1) \tag{21}$$

for $\alpha > \beta\,(F_t + X_{s,t})$.

Cumulative number of failures after time T

Predict

$$F(T) = (\alpha/\beta)\left[1 - e^{-\beta(T-s+1)}\right] + X_{s-1} \tag{22}$$

Maximum number of failures ($T = \infty$)

Predict

$$F(\infty) = \frac{\alpha}{\beta} + X_{s-1}\,. \tag{23}$$

Failures in an interval range

Predict *failure count in the range* $[t_1, t_2]$ as follows in Equation (24):

$$F(t_1, t_2) = \frac{\alpha}{\beta}\left[1 - \exp\left(-\beta(t_2 - s + 1)\right)\right] - X_{s,t1} \tag{24}$$

Maximum number of remaining failures, predicted at time t, after X(t) failures have been observed

Predict

$$RF(t) = \frac{\alpha}{\beta} + X_{s-1} - X(t) \tag{25}$$

Remaining failures

Predict *remaining failures* as a function of *total test time t_t* as follows in Equation (26):

$$r(t_t) = \frac{\alpha}{\beta}\left[\exp\left(-\beta(t_t - (s-1))\right)\right] \tag{26}$$

Fraction of remaining failures

Compute *fraction of remaining failures* predicted at time t as follows in Equation (27):

$$p(t) = \frac{r(t)}{F(\infty)} \tag{27}$$

Operational quality

Compute *operational quality* predicted at time *t* as follows in Equation (28):

$$Q(t) = 1 - p(t) \tag{28}$$

Total test time to achieve specified remaining failures

Predict *total test time* required to achieve a specified *number of remaining failures* at t_t, $r(t_t)$, as follows in Equation (29):

$$t_t = [\log[\alpha / (\beta[r(t_t)])]] / \beta \tag{29}$$

5.3.1.11 Development of confidence intervals for Schneidewind software reliability model

Confidence interval calculations and plots should be made, as in the following *example*:

a) Let $F(T)$ be any of the prediction equations, where T is test or operational time

b) From large sample size (N) theory, we have the following (see Farr [B12]):

c) $F(\hat{T}) - (Z_{1-\alpha/2} \times S) \le F(T) \le F(\hat{T}) + (Z_{1-\alpha/2} \times S)$

where Equation (30) (see Dixon and Massey [B8]) is expressed as follows:

$$S = \sqrt{\sum_{i=1}^{N} \frac{\left(F_i(\hat{T}) - F(\bar{\hat{T}})i\right)^2}{N-1}} \tag{30}$$

where N = sample size, $Z_{1-\alpha/2}$, for $\alpha = 10$, $Z_{1-\alpha/2} = 1.6449$.

$D(T)$ = cumulative failures is used for $F(T)$ on the plot in Figure 8, where $D(T)$ is the predicted value, *UD_T* and *LD_T* are the upper and lower confidence limits, respectively, and *Act* $D(T)$ is the actual value.

5.3.1.12 Parameter analysis

Insight into the possible error in prediction should be made after the parameters α and β have been estimated by a tool, such as SMERFS[3] and CASRE, and accompanied by the MSE analysis, but *before* predictions are made. It is not advisable to waste resources in making predictions for projected low reliability (e.g., high failure rate) software because it is highly likely that this software *will* have low reliability and that major rework would be necessary to improve its reliability. An example is provided in Figure 9 where, failure rate is plotted against the ratio β/α for NASA GSFC projects, NASA Space Shuttle OI projects, and Project 2. Failure rate decreases and reliability increases with increasing β/α. This plot was made to *illustrate* the above point. In practice, predictions would *not* be made for software with low values of. β/α. Rather, prediction priority would be given to software with high values of β/α. In Figure 9, OI25 is this type of software. Contrariwise, low values of β/α are useful for signaling the need for reliability improvement. In Figure 9, GSFC Project 1 is this type of software. A cautionary note is that the foregoing analysis is an *a priori* assessment of likely prediction error and does not mean, necessarily, that accurate predictions could not be obtained with *low* values of β/α in certain situations.

[3] Information about SMERFS is available at http://www.slingcode.com/smerfs.

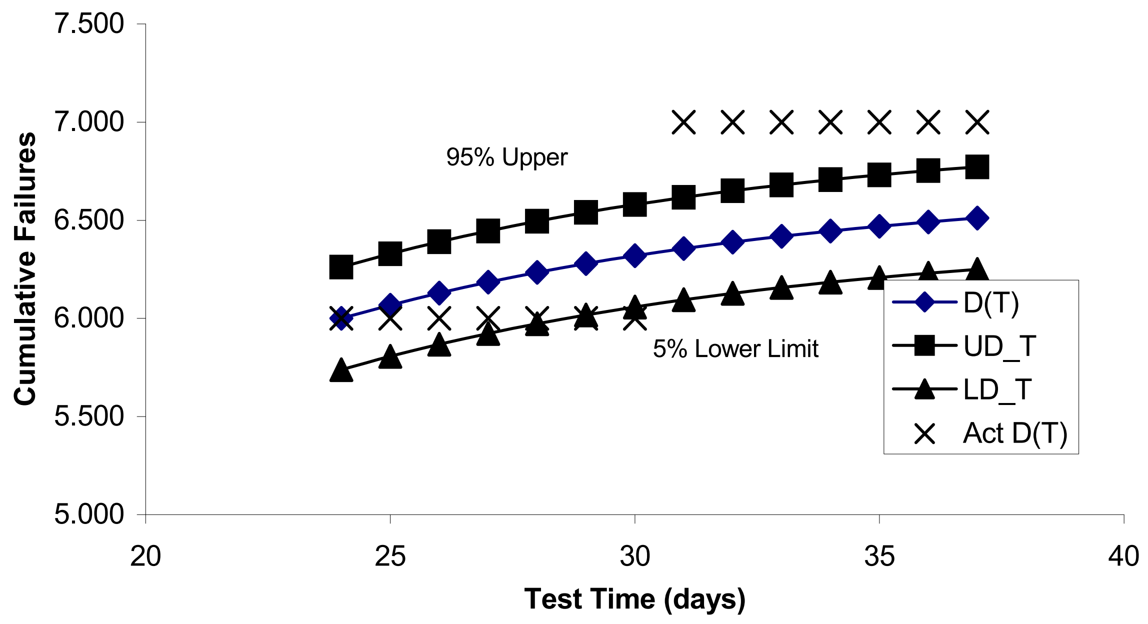


Figure 8—Confidence limits for cumulative failures NASA GSFC Facility 3 data

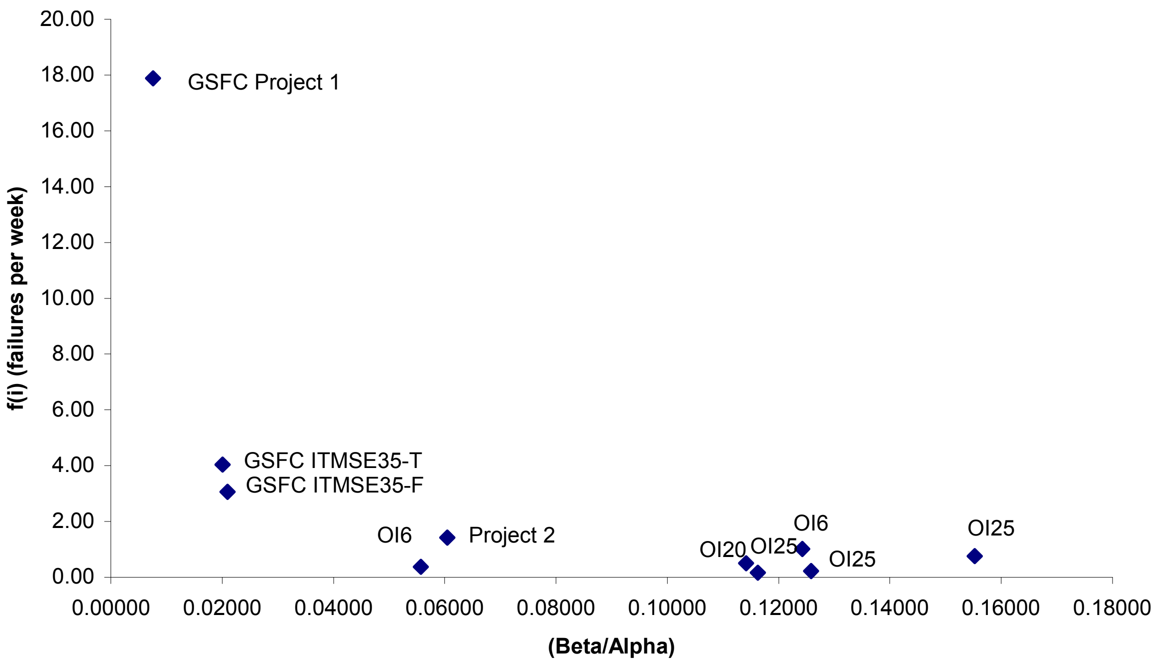


Figure 9—Failure rate *f*(*i*) versus (beta/alpha) ratio

5.3.1.12.1 Criteria for safety

The criteria for safety are as follows:

a) Compute predicted remaining failures $r(t_t) < r_c$, where r_c is a specified critical value

b) Compute predicted time to next failure $T_F(t_t) > t_m$, where t_m is mission duration

5.3.1.12.2 Risk assessment

The risk criterion metric for *remaining failures* at total test time t_t should be computed as follows in Equation (31):

$$\text{RCM}\, r(t_t) = \frac{r(t_t) - r_c}{r_c} = \frac{r(t_t)}{r_c} - 1 \tag{31}$$

The risk criterion metric for *time to next failure* at total test time t_t should be computed as follows in Equation (32):

$$\text{RCM } T_F(t_t) = \frac{t_m - T_F(t_t)}{t_m} = 1 - \frac{T_F(t_t)}{t_m} \tag{32}$$

5.3.1.13 Schneidewind model implementation status and reference applications

The model has been implemented in FORTRAN and C++ by the Naval Surface Warfare Center, Dahlgren, VA, USA, as part of the SMERFS. It can be run on PCs under all Windows® operating systems.[4, 5]

Known applications of this model are as follows:

— Reliability prediction and assessment of the on-board NASA Space Shuttle software (see *Handbook of Software Reliability Engineering* [B46], Schneidewind and Keller [B59], Schneidewind [B62])

— Naval Surface Warfare Center, Dahlgren, VA, USA: Research in reliability prediction and analysis of the TRIDENT I and II Fire Control Software (see Farr and Smith [B13])

— Marine Corps Tactical Systems Support Activity, Camp Pendleton, CA, USA: Development of distributed system reliability models (see Schneidewind [B61])

— NASA JPL, Pasadena, CA, USA: Experiments with multi-model SR approach (see Lyu and Nikora [B45])

— NASA GSFC, Greenbelt, MD, USA: Development of fault correction prediction models (see Schneidewind [B64])

— NASA GSFC (see Schneidewind [B63])

— Integrated, multi-model approach to reliability prediction (see Bowen [B3])

5.3.1.14 Informative references

There are several informative references to supplement the material in this section. The informative references, which are cited in the bibliography Annex G, are Dixon and Massey [B8], Farr [B11], *Handbook of Software Reliability Engineering* [B46], and Schneidewind [B58], [B60], [B61], [B62], [B64].

[4] This information is given for the convenience of users of this standard and does not constitute an endorsement by the IEEE of these products. Equivalent products may be used if they can be shown to lead to the same results.
[5] Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

5.3.2 Initial: Generalized exponential model

5.3.2.1 Generalized exponential model objectives

Many popular SR models yield similar results. The basic idea behind the generalized exponential model is to simplify the modeling process by using a single set of equations to represent models having exponential hazard rate functions.

The generalized exponential model contains the ideas of several well-known SR models. The main idea is that the failure occurrence rate is proportional to the number of faults remaining in the software. Furthermore, the failure rate remains constant between failure detections and the rate is reduced by the same amount after each fault is removed from the software. Thus, the correction of each fault has the same effect in reducing the hazard rate of the software. The objective of this model is to generalize the forms of several well-known models into a form that should be used to predict the following:

- Number of failures that will occur by a given time (execution time, labor time, or calendar time)
- Maximum number of failures that will occur over the life of the software
- Maximum number of failures that will occur after a given time
- Time required for a given number of failures to occur
- Number of faults corrected by a given time
- Time required to correct a given number of faults
- Reliability model for the software after release
- Mean time to failure (MTTF, MTBF) after release

5.3.2.2 Generalized exponential assumptions

The assumptions of the generalized exponential model are the following:

- The failure rate is proportional to the current fault content of a program.
- All failures are equally likely to occur and are independent of each other.
- Each failure is of the same severity as any other failure.
- The software is operated during test in a manner similar as the anticipated operational usage.
- The faults that caused the failure are corrected immediately without introducing new faults into the program.
- The model assumes that much of the software has been written and unit (module) tested and has or is being assembled. Thus, it is most applicable to the integration, system test, or deployment phases of development.

5.3.2.3 Generalized exponential structure

The generalized exponential structure should contain the hazard rate function as follows in Equation (33):

$$z(x) = K\left[E_0 - E_c(x)\right] \tag{33}$$

where

$z(x)$	is the hazard rate function in failures per time unit
x	is the time between failures

E_0	is the initial number of faults in the program that will lead to failures. It is also viewed as the number of failures that would be experienced if testing continued indefinitely
E_c	is the number of faults in the program, which have been found and corrected, once x units of time have been expended
K	is a constant of proportionality: failures per time unit per remaining fault

Equation (34) shows that the number of remaining faults per failure, E_r, is given as follows:

$$E_r = \frac{z(x)}{K} = [E_0 - E_c(x)] \tag{34}$$

NOTE 1—Equation (34) has no fault generation term; the assumption is that no new faults are generated during program testing that would lead to new failures. More advanced models that include fault generation are discussed in *Software Reliability: Measurement, Prediction, Application* [B54] and *Software Engineering: Design, Reliability, and Management* [B69].

NOTE 2—Many models in common use are represented by the specific cases of Equation (33). Some of these models are summarized in Table 2.

Table 2—Reliability models that fit the generalized exponential model form for the hazard rate function

Model name	Original hazard rate function	Parameter equivalences with generalized model form	Comments
Generalized form	$K[E_0 - E_c(x)]$	—	—
Exponential model [B67]	$K'\left[\frac{E_0}{I_T} - \varepsilon_c(x)\right]$	$\varepsilon_c = \frac{E_c}{I_T}$ $K = \frac{K'}{I_T}$	Normalized with respect to I_T, the number of instructions
Jelinski-Moranda [B31]	$\phi[N-(i-1)]$	$\phi = K$ $N = E_0$ $E_c = (i-1)$	Applied at the discovery of a fault and before it is corrected
Basic model (see pp. 449–455 in Musa's latest book [B55])	$\lambda_0[1-\mu/\nu_0]$	$\lambda_0 = KE_0$ $\nu_0 = E_0$ $\mu = E_c$	
Logarithmic [B53]	$\lambda_0 e^{-\phi\mu}$	$\lambda_0 = KE_0$ $E_0 - E_c(x) = E_0 e^{-\phi\mu}$	Basic assumption is that the remaining number of faults decreases exponentially.

5.3.2.3.1 Parameter estimation

The simplest method of parameter estimation is the moment method. Consider the generalized form model with its two unknown parameters K and E_0. The classical technique of moment estimation would match the first and second moments of the probability distribution to the corresponding moments of the data. A slight modification of this procedure is to match the first moment, the mean, at two different values of x. That is, letting the total number of runs be n, the number of successful runs be r, the sequence of clock times to failure t_1, t_2, …, t_{n-r} and the sequence of clock times for runs without failure T_1, T_2, …, T_r yields, for the hazard rate function as follows in Equation (35):

$$z(x) = \frac{\text{Failures}(x)}{\text{Hours}(x)} = \frac{n-r}{H} \tag{35}$$

where the total running hours, H, is given by

$$H = \sum_{i=1}^{n-r} t_i + \sum_{i=1}^{r} T_i$$

Equating the generalized form equation of Equation (33) and Equation (35) to failure data measured during actual or simulated operational test at two different values of time yields Equation (36) and Equation (37):

$$z(x_1) = \frac{n_1 - r_1}{H_1} = K[E_0 - E_c(x_1)] \tag{36}$$

$$z(x_2) = \frac{n_2 - r_2}{H_2} = K[E_0 - E_c(x_2)] \tag{37}$$

Simultaneous solution of Equation (36) and Equation (37) yields estimators denoted by ^, for the following parameters:

$$\begin{aligned}\hat{E}_0 &= \frac{E_c x_1 - \dfrac{z(x_1)}{z(x_2)} E_c(x_2)}{1 - \dfrac{z(x_1)}{z(x_2)}} \\ &= \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{z(x_2) - z(x_1)}\end{aligned} \tag{38}$$

$$\begin{aligned}\hat{K} &= \frac{z(x_1)}{E_0 - E_c(x_1)} \\ &= \frac{z(x_2) - z(x_1)}{E_c(x_1) - E_c(x_2)}\end{aligned} \tag{39}$$

Since all of the parameters of the five models in Table 2 are related to E_0 and K by simple transformations, Equation (38) and Equation (39) apply to all of the models. For example, if we use the Musa basic model of Table 2 and to determine $\hat{\lambda}_0$ and $\hat{v}_0$ by using Equation (38) and Equation (39) and the transformation $v_0 = E_0$ and $\mu_0 = KE_0$ to obtain

$$\hat{v} = \hat{E}_0 = \frac{z(x_2)E_c(x_1) - z(x_1)E_x(x_2)}{z(x_2) - z(x_1)} \tag{40}$$

$$\begin{aligned}\hat{\lambda}_0 &= \hat{K}\hat{E}_0 \\ &= \frac{z(x_2) - z(x_1)}{E_c(x_1) - E_c(x_2)} \times \frac{z(x_2)E_c(x_1) - z(x_1)E_x(x_2)}{z(x_2) - z(x_1)}\end{aligned} \tag{41}$$

$$\hat{\lambda}_0 = \frac{z(x_2)E_c(x_1) - z(x_1)E_c(x_2)}{E_c(x_1) - E_c(x_2)} \tag{42}$$

Similar results can be obtained for the other models in Table 2. More advanced estimates of the model parameters can be developed using least squares and maximum likelihood estimation (see Shooman [B71]).

5.3.2.4 Generalized exponential limitations

The generalized exponential model has the following limitations:

- It does not account for the possibility that each failure may be dependent on others.
- It assumes no new faults are introduced in the fault correction process.
- Each fault detection has a different effect on the software when the fault is corrected. The logarithmic model handles this by assuming that earlier fault corrections have a greater effect than later ones.
- It does not account for the possibility that failures can increase over time as the result of program changes, although techniques for handling this limitation have been developed.

5.3.2.5 Generalized exponential data requirements

During test, a record should be made of each of the total of n test runs. The test results include the r successes and the $n - r$ failures along with the time of occurrence measured in clock time and execution time, or test time if operational tests cannot be conducted. Additionally, there should be a record of the times for the r successful runs. Thus, the required data is the total number of runs n, the number of successful runs r, the sequence of clock times to failure t_1, t_2, …, t_{n-r} and the sequence of clock times for runs without failure T_1, T_2, …, T_r.

5.3.2.5.1 Generalized exponential applications

The generalized exponential model(s) are applicable when the fault correction process is well controlled (i.e., the fault correction process does not introduce additional faults).

The major model applications are described as follows. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

- Predicting future failures and fault corrections.
- Control: Comparing prediction results with predefined goals and flagging software that fails to meet those goals.
- Assessment: Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process). The formulation of test strategies is also part of the assessment. Test strategy formulation involves the determination of priority, duration, and completion date of testing, allocation of personnel, and allocation of computer resources to testing.

5.3.2.6 Reliability predictions

Prediction of total number of faults should be computed by $\hat{E}_0$. Other predictions should be made as follows in Equation (43):

$$\text{time required to remove the next } m \text{ faults} = \sum_{j=n+1}^{n+m} \frac{1}{\hat{K}_0(\hat{E}_0) - j + 1} \tag{43}$$

current failure rate at time τ $f(\tau) = \hat{K}_0 \left(\hat{E}_0 e^{-\hat{K}_0 \tau} \right)$ (44)

5.3.2.7 Generalized exponential implementation status and references

The generalized exponential model has not been implemented as a standalone model in the SMERFS software. For an illustration of how to apply these generalized exponential models, see Annex E. The many models it represents, however, have been implemented in several tools including SMERFS from the Naval Surface Warfare Center, Dahlgren, VA, USA. See Annex A for details.

While the generalized exponential model is generally not applied as a class, many of the models that it includes as special cases have been applied successfully. For example applications, see references by Jelinski and Moranda [B31], Kruger [B38], and Shooman [B69].

5.4 Initial model—Musa/Okumoto logarithmic Poisson execution time model

5.4.1.1 Musa/Okumoto model objectives

The logarithmic Poisson model is applicable when the testing is done according to an operational profile that has variations in frequency of application functions and when early fault corrections have a greater effect on the failure rate than later ones. Thus, the failure rate has a decreasing slope.

5.4.1.2 Musa/Okumoto model assumptions

The assumptions for this model are as follows:

- The software is operated in a similar manner as the anticipated operational usage.
- Failures are independent of each other.
- The failure rate decreases exponentially with execution time.

5.4.1.3 Musa/Okumoto model structure

From the model assumptions, we have the following:

$\lambda(t)$ = failure rate after t amount of execution time has been expended $\lambda_0 e^{-\theta\mu(t)}$

The parameter λ_0 is the initial failure rate parameter and θ is the failure rate decay parameter with $\theta > 0$.

Using a reparameterization of $\beta_0 = \theta^{-1}$ and $\beta_1 = \lambda_0\theta$, then the maximum likelihood estimates of β_0 and β_1 should be made, as shown in *Software Reliability: Measurement, Prediction, Application* [B54], according to Equation (45) and Equaiton (46) as follows:

$$\hat{\beta}_0 = \frac{n}{\ln(1+\hat{\beta}_1)t_n} \tag{45}$$

$$\frac{1}{\hat{\beta}_1} \sum_{i=1}^{n} \frac{1}{1+\hat{\beta}_1 t_i} = \frac{n t_n}{(1+\hat{\beta}_1 t_i)\ln(1+\hat{\beta}_1 t_i)} \tag{46}$$

Here, t_n is the cumulative CPU time from the start of the program to the current time. During this period, n failures have been observed. Once maximum likelihood estimates are made for β_0 and β_1, the maximum likelihood estimates for θ and λ_0 should be made, as follows in Equation (47) and Equation (48):

$$\hat{\theta} = \frac{1}{n} \ln\left(1 + \hat{\beta}_1 t_n\right) \tag{47}$$

$$\hat{\lambda}_0 = \hat{\beta}_0 \hat{\beta}_1 \, . \tag{48}$$

5.4.1.4 Musa/Okumoto model limitation

The failure rate may rise as modifications are made to the software.

5.4.1.5 Musa/Okumoto model data requirements

The required data is either:

— The time between failures, represented by X_i's.

The time of the failure occurrences, given as follows in Equation (49):

$$t_i = \sum_{j=1}^{i} X_j \tag{49}$$

5.4.1.6 Musa/Okumoto model applications

The major model applications are described as follows. These are separate but related applications that, in total, comprise an integrated reliability program.

— *Prediction:* Predicting future failure times and fault corrections, described in *Software Reliability: Measurement, Prediction, Application* [B54].

— *Control:* Comparing prediction results with predefined goals and flagging software that fails to meet goals.

— *Assessment:* Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, and revise process). The formulation of test strategies is also a part of assessment. It involves the determination of priority, duration, and completion date of testing, allocation of personnel, and computer resources to testing.

5.4.1.7 Musa/Okumoto model reliability predictions

In *Software Reliability: Measurement, Prediction, Application*, Musa, Iannino, and Okumoto [B54] show that from the assumptions of 5.4.1.2 and the fact that the derivative of the mean value function is the failure rate function, we have Equation (50) as follows:

$$\hat{\lambda}(\tau) = \frac{\hat{\lambda}_0}{\hat{\lambda}_0 \theta \tau + 1} \tag{50}$$

where

$\hat{\mu}(\tau) =$ mean number of failures experienced by time τ is expended $= \dfrac{1}{\hat{\theta}} \ln\left(\hat{\lambda}_0 \hat{\theta}\tau + 1\right)$ (51)

5.4.2 Musa/Okumoto model implementation status and reference applications

The model has been implemented by the Naval Surface Warfare Center, Dahlgren, VA, USA, as part of SMERFS. This model has also been implemented in CASRE.[6]

See the following references for example applications: Ehrlich et al. [B10], Musa and Okumoto [B53], *Software Reliability: Measurement, Prediction, Application* [B54], and *Handbook of Software Reliability Engineering* [B55].

5.5 Experimental approaches

Several improvements to the SR models described in 5.3 and 5.4 have been proposed. First, researchers at the City University of London have devised a method of recalibrating the models (see Brocklehurst and Littlewood [B4]) to reduce their biases. These findings suggest that the revised models yield consistently more accurate predictions than the original models. Second, work has been done in combining the results from two or more models in a linear fashion, with the objective of increasing predictive accuracy (see Lu et al. [B44], Lyu and Nikora [B45]). Combined models may yield more accurate results than individual models. Third, efforts to incorporate software complexity metrics into reliability models by Kafura and Yerneni [B33] and Munson and Khoshgoftaar [B51], and to gauge the effects of different types of testing (e.g., branch testing, data path testing) on reliability growth (see Mathur and Horgan [B47]) have been investigated. Finally, the use of neural networks for SR parameter estimation has been investigated (see Karunantithi et al. [B35]).

5.6 Software reliability data

This subclause provides a nine-step procedure for collecting SR data and the data categories.

5.6.1 Data collection procedure

The following nine-step procedure should be used for SR data collection.

Step 1: Establish the data collection plan objectives.

The first step in the procedure to collect data is to establish a data collection plan, which includes the objectives of data collection, the data requirements, and the data items to be collected. Data collection does involve cost, so each item should be examined to see if the need is worth the cost. This should be done in terms of the applications of SRE. If the item is questionable, alternatives such as collecting it at a lower frequency may be considered. Possibilities of collecting data that can serve multiple purposes should be examined.

Step 2: Plan the data collection process.

The second step in the procedure to collect data is to plan the data collection process. All parties (designers, coders, testers, users, and key management) should participate in the planning effort. The goals of the data collection effort should be presented. A first draft of the data collection plan should be presented as a starting point. The plan should include the following topics:

[6] Download available at the following URL: http://www.openchannelfoundation.org/projects/CASRE_3.0.

- What data items will be gathered?
- Who will gather the data?
- How often will the data be reported?
- Formats for data reporting (e.g., spreadsheet, web site)
- How is the data to be stored and processed?
- How will the collection process be monitored to ensure integrity of the data?

Recording procedures should be carefully considered to make them appropriate for the application. Solicitation of data from project members will reduce effort and make collection more reliable.

For the failure count method of SR modeling, the data collection interval should be selected to correspond to the normal reporting interval of the project from which data are being collected (e.g., weekly, monthly).

Step 3: Apply tools.

Tools identified in data collection should be used. The amount of automatic data collection should be considered. To minimize time, automated tools should be used whenever possible.

When decisions are made to automate data collection, the following questions should be considered:

- Availability of the tool. Can it be purchased or must it be developed?
- What is the cost involved in either the purchase of the tool or its development?
- When will the tool be available? If it must be developed, will its development schedule coincide with the planned use?
- What effect will the data collection process have on the software development schedule?

Records should be kept of the number of faults detected after the release of the software. This should be compared with reliability predictions of similar projects that did not employ the tools. Estimates of reduced fault correction time should be made based on the based on predictions, for example, of failure rate.

Step 4: Provide training.

Once the tools are operational, and the process is well defined, all concerned parties should be trained in the use of the tools and the performance of the process. The data collectors should understand the purpose of the measurements and know what data are to be gathered.

Step 5: Perform trial run.

A trial run of the data collection plan should be made to resolve any problems or misconceptions about the plan, the tools and the process.

Step 6: Implement the plan.

Data should be collected and reviewed promptly. If this is not done, quality will suffer. Reports should be generated to identify any unexpected results for further analysis.

Step 7: Monitor data collection.

The data collection activities should be monitored as it proceeds to insure that the objectives are met and that the program is meeting its reliability goals.

Step 8: Make predictions using the data.

SR should be predicted at regular, frequent intervals, using the collected data, to maximize visibility of the reliability of the software.

Step 9: Provide feedback.

Feedback concerning successes and failures in the implementation of the data collection plan, and the use of the data collection process and tools should be done as early as possible so that corrections can be completed in a timely manner.

NOTE—This step may be implemented within the Measurement Process in 6.3.7 of IEEE Std 12207-2008 [B29].

5.6.2 Data categories

5.6.2.1 Execution time data and failure count data

It is generally accepted that execution (CPU) time is superior to calendar time for SR measurement and modeling. Execution time gives a truer picture of the stress placed on the software. If execution time is not available, approximations such as clock time, weighted clock time, or units that are natural to the applications, such as transaction count, should be used (see pp. 156–158 in *Software Reliability: Measurement, Prediction, Application* [B54]).

Reliability models have the capability to estimate model parameters from either failure-count or time between failures (may also be called time-to-failures (TTFs) or time-to-next-failure) data, because the maximum likelihood estimation method can be applied to both. Both SMERFS and CASRE can accept either type of data.

5.6.2.2 Project data

Project data contains information to identify and characterize each system. Project data allow users to categorize projects based on application type, development methodology, and operational environment. The following project-related data are suggested:

- The name of each life-cycle activity (e.g., requirements definition, design, code, test, operations)
- The start and end dates for each life-cycle activity
- The effort spent (in staff months) during each life-cycle activity
- The average staff size and development team experience
- The number of different organizations developing software for the project

5.6.2.3 Component data

Component data can be used to see whether there are correlations between reliability and component characteristics (e.g., large components may be related to low reliability). For each system component (e.g., subsystem, module), the following data should be collected:

- The name of the software development project
- Computer hardware that is used to support software development
- Average and peak computer resource utilization (e.g., CPU, memory, input *I* output channel)
- Software size in terms of executable source lines of code, number of comments, and the total number of source lines of code
- Source language

5.6.2.4 Dynamic failure data

For each failure, the following data should be recorded:

- The type of failure (e.g., interface, code syntax)
- The method of fault and failure detection (e.g., inspection, system abort, invalid output)
- The code size where the fault was detected (e.g., instruction)
- The activity being performed when the problem was detected (e.g., testing, operations, and maintenance)
- The date and time of the failure
- The severity of the failure (e.g., critical, minor)
- And at least one of the following data items:
  - The number of CPU hours since the last failure
  - The number of runs or test cases executed since the last failure
  - The number of wall clock hours since the last failure
  - The number of test hours since the last failure
  - Test labor hours since the last failure

5.6.2.5 Fault correction data

For each fault corrected, the following information should be recorded:

- The type of correction (e.g., software change, documentation change, requirements change)
- The date and time the correction was made
- The labor hours required for correction

And at least one of the following data items:

- The CPU hours required for the correction
- The number of computer runs required to make the correction
- The wall clock time used to make the correction

Annex A

(informative)

Additional software reliability estimation models

This annex contains descriptions of additional models available to a researcher or SR analyst for use on projects that were not discussed in Clause 5. These models may be useful on projects where the assumptions of the initial models in Clause 5 do not apply or the models do not closely fit the data.

A.1 Littlewood/Verrall model

A.1.1 Littlewood/Verrall model objectives

There are two sources of uncertainty that need to be taken into account when software fails and corrections are attempted. First, there is uncertainty about the nature of the operational environment: we do not know when certain inputs will appear, and, in particular, we do not know which inputs will be selected next. Thus, even if we had complete knowledge of which inputs were failure-prone, we still could not tell with certainty when the next one would induce a failure.

The second source of uncertainty concerns what happens when an attempt is made to remove the fault that caused the failure. There is uncertainty for two reasons. First, it is clear that not all the faults contribute to the unreliability of a program to the same degree. If the software has failed because a fault has been detected that contributes significantly to unreliability, then there will be a correspondingly large increase in the reliability (reduction in the failure rate) when the fault is removed. Second, we can never be sure that we have removed a fault successfully; indeed, it is possible that some new fault has been introduced and the reliability of the program has been made worse. The result of these two effects is that the failure rate of a program changes in a random way as fault correction proceeds.

The Littlewood/Verrall model, unlike the other models discussed, takes account of both of these sources of uncertainty.

A.1.2 Littlewood/Verrall model assumptions

The following assumptions apply to the Littlewood/Verrall model:

- The software is operated during the collection of failure data in a manner that is similar to that for which predictions are to be made; the test environment is an accurate representation of the operational environment.
- The times between successive failures are conditionally independent exponentially distributed random variables.

A.1.3 Littlewood/Verrall model structure

This model treats the successive rates of occurrence of failures, as fault corrections take place, as random variables. It assumes the following in Equation (A.1):

$$P\left(t_i \middle| \Lambda_i = \lambda_i\right) = \lambda_i e^{-\lambda_i t_i} \tag{A.1}$$

The sequence of rates λ_i is treated as a sequence of independent stochastically decreasing random variables. This reflects the likelihood, but not certainty, that a fault correction will be effective. It is assumed that

$$g(\lambda_i) = \frac{\Psi(i)^n \lambda_i^{a-1} e^{-\Psi(i)\lambda_i}}{\Gamma(\alpha)} \text{ for } \lambda_I > 0$$

which is a gamma distribution with parameters α, $\psi(i)$.

The function $\psi(i)$ should determine reliability growth. If, as is usually the case, $\psi(i)$ is an increasing function of i, Λ_i, forms a decreasing sequence. By setting $\psi(i)$ to either $\beta_0 + \beta_1 i$ or $\beta_0 + \beta_1 i^2$ and eliminating α, Littlewood and Verrall present a method for estimating β_0 and β_1 based on maximum likelihood. By eliminating α from the likelihood equations, the estimate of α should be expressed as a function of the estimates of the other two parameters. See Farr [B11] and Littlewood and Verrall [B40] for details. The maximum likelihood calculation requires a numerical optimization routine, which is available in commercially available software, such as those found in Annex B.

Least squares estimates of the parameters (α, β_0, β_1) should be found by minimizing the following Equation (A.2):

$$S(\alpha, \beta_0, \beta_1) = \sum_{i=1}^{n} \left(Xi - \frac{[\Psi(i)]}{\alpha - 1} \right)^2 \qquad \text{(A.2)}$$

See Farr [B12] for further details.

A.1.4 Littlewood/Verrall limitations

The limitation is that the model cannot predict the number of faults remaining in the software.

A.1.5 Littlewood/Verrall data requirements

The only required data is either of the following:

— The time between failures, the X_i's

— The time of the failure occurrences, $t_i = \sum_{j=1}^{i} X_i$

A.1.6 Littlewood/Verrall applications

The major model applications are described as follows. These are separate but related uses of the model that, in total, comprise an integrated reliability program.

— *Prediction:* Predicting future failures, fault corrections, and related quantities described in 5.3.1.6.1

— *Control:* Comparing prediction results with predefined goals and flagging software that fails to meet those goals

— *Assessment:* Determining what action to take for software that fails to meet goals (e.g., intensify inspection, intensify testing, redesign software, revise process). The formulation of test strategies is also part of assessment

Reliability predictions

Prediction of MTTF should be made as follows in Equation (A.3):

$$\mathrm{MTT\hat{F}} = \hat{E}(X_i) = \frac{\hat{\Psi}(i)}{\hat{\alpha} - 1} \tag{A.3}$$

Prediction of failure rate should be made as follows in Equation (A.4):

$$\hat{\lambda}(t) = \frac{\hat{\alpha}}{t + \hat{\Psi}(i)} \tag{A.4}$$

Prediction of reliability should be made as follows in Equation (A.5):

$$R(t) = P(T_i > t) = \hat{\Psi}(i)^{\hat{\alpha}}[t + \hat{\Psi}(i)]^{-\hat{\alpha}} \tag{A.5}$$

A.1.7 Littlewood/Verrall implementation status and reference applications

The model has been implemented as part of the SMERFS. This model has also been implemented in the SRMP at the Center for Software Reliability in London, U.K., by Reliability and Statistical Consultants, Ltd. This package runs on a PC.

See the following references for examples of applications: Abdel-Ghaly et al. [B1], Kanoun and Sabourin [B34], and Mellor [B49].

A.2 Duane’s model

A.2.1 Duane’s model objectives

This model uses times of failure occurrences. The number of such occurrences considered per unit of time is assumed to follow an NHPP. This model was originally proposed by J. T. Duane who observed that the cumulative failure rate when plotted against the total testing time on log–log paper tended to follow a straight line. This model has had some success in its application (see Duane [B9]). It is best applied later in the testing phase and in the operational phase.

Duane’s model assumptions

The assumptions are as follows:

— The software is operated in a similar operational profile as the anticipated usage.

— The failure occurrences are independent.

— The cumulative number of failures at any time t, [$N(t)$], follows a Poisson distribution with mean $m(t)$. This mean is taken to be of the form $m(t) = \lambda t^b$.

A.2.2 Duane’s model structure

If $m(t) = \lambda t^b$ is plotted on log–log paper a straight line of the form $Y = a + bX$ with $a = \ln \lambda$, $b = b$, and $X = \ln t$ is obtained.

Maximum likelihood estimates are shown by Crow [B7] to be as follows:

$$\hat{b} = \frac{a}{\sum_{i=1}^{n-1} \ln\left(\frac{t_n}{t_i}\right)} \quad \hat{\lambda} = \frac{n}{t_n^{\hat{b}}}$$

where the t_i's are the observed failure times in either CPU time or wall clock time and n is the number of failures observed to date.

Least squares estimates for a and b of the straight line on log–log paper can be derived using recommended practice linear regression estimates.

A.2.3 Duane's model data requirements

The model requires the time of the failure occurrences, $t_i, i = 1, \cdots, n$.

A.3 Yamada, Ohba, Osaki s-shaped reliability model

A.3.1 S-shaped reliability growth model objectives

This model uses times of failure occurrences. The number of occurrences per unit of time is assumed to follow an NHPP. This model was proposed by Yamada, Ohba, and Osaki [B76]. It is based upon Goel and Okumoto's NHPP (see Goel and Okumoto [B17]). The difference is that the mean value function of the Poisson process is s-shaped. At the beginning of the testing phase, the fault detection rate is relatively flat but then increases exponentially as the testers become familiar with the program. Finally, it levels off near the end of testing as faults become more difficult to uncover.

A.3.2 S-shaped reliability growth model assumptions

The assumptions are as follows:

— The software is operated during test in a manner similar to anticipated usage.

— The failure occurrences are independent and random.

— The initial fault content is a random variable.

— The time between failure $i-1$ and failure i depends on the TTF of failure $i-1$.

— Each time a failure occurs, the fault which caused it is immediately removed, and no other faults are introduced.

A.3.3 S-shaped reliability growth model structure

The model structure is as follows in Equation (A.6):

$P(N_t = n)$ = probability that the cumulative number of faults up to time t, N_t, is equal to n

$$= \frac{M(t)^n e^{-M(t)}}{n!} \tag{A.6}$$

where n = 0, 1, …, with

$M(t)$ = the mean value function for the Poisson process

$$= a(1-(1+bt)e^{-bt}) \text{ with both } a,\, b > 0 \tag{A.7}$$

and with initial conditions $M(0) = 0$ and $M(\infty) = a$.

Therefore, the fault detection rate is given as follows in Equation (A.8):

$$\frac{dM(t)}{dt} = ab^2te^{-bt} \tag{A.8}$$

Letting n_i, $i = 1, \ldots, k$ be the cumulative number of faults found up to time t_i, $i = 1, \ldots, k$, the maximum likelihood estimates for a and b are shown to satisfy the following in Equation (A.9) and Equation (A.10):

$$\hat{a} = \frac{k_k}{\left(1-(1+\hat{b}t_k)e^{-\hat{b}t_k}\right)} \tag{A.9}$$

$$\hat{a}t_k^2e^{-\hat{b}t_k} = \sum\left[\frac{(n_i - n_{i-1})\left(t_ie^{-\hat{b}t_i} - t_{i-1}e^{-\hat{b}t_{i-1}}\right)}{(1+\hat{b}t_{i-1})e^{-\hat{b}t_{i-1}} - (1+\hat{b}t_i)e^{-\hat{b}t_i}}\right] \tag{A.10}$$

A.3.4 S-Shaped model data requirements

The model requires the failure times t_i, $i = 1, \ldots, k$ as input data.

A.4 Jelinski/Moranda reliability growth model

A.4.1 Jelinski/Moranda model objectives

The basic idea behind this model is that failure occurrence rate is proportional to the number of faults remaining, the rate remains constant between failure detections, and the rate is reduced by the same amount after each fault is removed. The last idea means that the correction of each fault has the same effect in reducing the hazard rate (instantaneous failure rate) of the program.

A.4.2 Jelinski/Moranda model assumptions

The assumptions of the Jelinski/Moranda MODEL are as follows:

- The rate of failure detection is proportional to the current fault content of a program.
- All failures are equally likely to occur and are independent of each other.
- Failures have equal severity.
- The failure rate remains constant over the interval between failure occurrences.
- The software, during test, is operated in a similar manner as the anticipated operational usage.
- The faults are corrected immediately without introducing of new faults.

A.4.3 Jelinski/Moranda model structure

Using these assumptions, the hazard rate is defined as follows in Equation (A.11):

$$z(t) = \phi[N-(i-1)] \tag{A.11}$$

where t is any time between the discovery of the $(i-1)$st failure and the ith failure. The quantity ϕ is proportionality constant in the first assumption. N is the total number of faults initially in the program. Hence if $(i-1)$ faults have been discovered by time t, there are $N-(i-1)$ remaining faults. The hazard rate is proportional to the remaining faults. As faults are discovered, the hazard rate is reduced by the same amount, ϕ, each time.

If $X_i = t_i - t_{i-1}$, the time between the discovery of the ith and the $(i-1)$st fault for $i = 1, \ldots, n$, where $t_0 = 0$, the X_i's are assumed to have an exponential distribution with hazard rate $z(t_i)$. That is,

$$f(X_i) = f[N-(i-1)]e^{-f[N-(i-1)]X_i}.$$

This leads to the maximum likelihood estimates of ϕ and N as the solutions to the following Equation (A.12) and Equation (A.13):

$$\hat{\phi} = \frac{n}{\hat{N}\left(\sum_{i=1}^{n} X_i\right) - \sum_{i=1}^{n}(i-1)X_i} \tag{A.12}$$

$$\sum_{i=1}^{n}\frac{1}{\hat{N}-(i-1)} = \frac{n}{\hat{N} - \dfrac{\sum_{i=1}^{n}(i-1)X_i}{\sum_{i=1}^{n} X_i}} \tag{A.13}$$

A.4.4 Jelinski/Moranda model data requirements

The model may use either of the following as input data for parameter estimation:

- The time between failure occurrences, i.e., the X_i's
- The total duration of the failure occurrences, $t_i = \sum_{j=1}^{i} X_j$

Annex B

(informative)

Determining system reliability

This annex describes methods for combining hardware and SR predictions into a system prediction.

B.1 Predict reliability for systems comprised of (hardware and software) subsystems

A simple way of dealing with the reliability of a system comprised of hardware and software is to make a structural model of the system. The most common types of structural models are reliability block diagrams (reliability graphs) and reliability fault trees.

If the hardware and software modes of failure are independent, then the system reliability, *RS*, can be treated as the product of the hardware and software reliability, and a separate model can be made for the hardware and software. Consider the following example.

A railroad boxcar will be automatically identified by scanning its serial number (written in bar code form) as the car rolls past a major station on a railroad system. Software compares the number read with a database for match, no match, or partial match. A simplified hardware graph for the system is given in Figure B.1, and the hardware reliability, *R*(*HW*), in the following Equation (B.1):

$$R(HW) = RS \times RC \times RD \times RP \tag{B.1}$$

The software graph is shown in Figure B.2, the software reliability, *R*(*SW*), in Equation (B.2), and combining these equations, the system reliability *R*(SYSTEM) is given in Equation (B.3) as follows:

$$R(SW) = RF \times RL \times RD \times RA \tag{B.2}$$

$$R(\mathrm{SYSTEM}) = R(HW) \times R(SW) \tag{B.3}$$

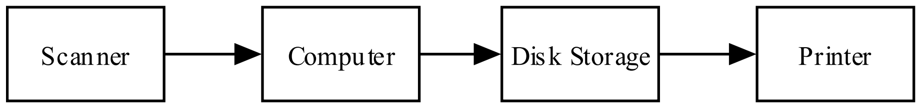


Figure B.1—Hardware model of a railroad boxcar identification system

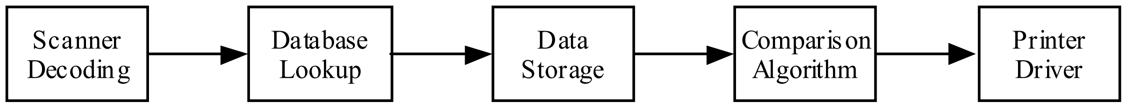


Figure B.2—Software model of a railroad boxcar identification system

In a more complex case, the hardware and software are not independent and a more complex model is needed. For example, consider a fault tolerant computer system with hardware failure probabilities *C*1, *C*2, *C*3, the same software on each computer, with failure probabilities *SW*1″, *SW*2″, *SW*3″, respectively, and a majority voter V (see Shooman [B71]). In the example, the voting algorithm V would compare the outputs of *SW*1″, *SW*2″, and *SW*3″ and declare a "correct" output based on two or three out of three outputs being equal. If none of the outputs is equal, the default would be used, i.e., *SW*1″. Furthermore, assume that some of the failures are dependent. That is, for example, a hardware failure in *C*1 causes a software failure in *SW*2″, or a software failure in *C*1 causes a failure in *SW*2″; these software components appear in parallel in the graph model in Figure B.3. Some software failures [*SW*″ in Equation (B.2)] are independent because this software is common to all computers. Therefore, failures in *SW*″ are not dependent on failures occurring in the non common parts of the software. This is shown in Figure B.3 and Equation (B.2) as *SW*″ in series with the parallel components.

$$R = [(C1 \times SW1'' + C2 \times SW2'' + C3 \times SW3'') \times SW'] \qquad \text{(B.4)}$$

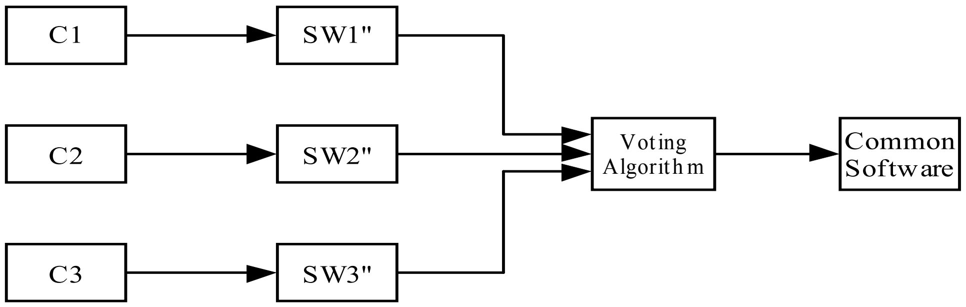


Figure B.3—Reliability graph for a fault tolerant computer system

Annex C

(informative)

Using reliability models for developing test strategies

C.1 Allocating test resources

It is important for software organizations to have a strategy for testing; otherwise, test costs are likely to get out of control. Without a strategy, each module you test may be treated equally with respect to allocation of resources. Modules need to be treated unequally. That is, more test time, effort and funds should be allocated to the modules which have the highest predicted number of failures, $F(t_1, t_2)$, during the interval $[t_1, t_2]$, where $[t_1, t_2]$ could be execution time or labor time (of testers) for a single module. In the remainder of this section, "time" means execution time. Use the convention that you make a prediction of failures at t_1 for a continuous interval with end-points $t_1 + 1$ and t_2.

The following sections describe how a reliability model can be used to predict $F(t_1, t_2)$. The test strategy is the following:

Allocate test execution time to your modules in proportion to $F(t_1, t_2)$

Model parameters and predictions are updated based on observing the actual number of failures, $X_{0,t1}$, during 0,t1. This is shown in Figure C.1, where you predict $F(t_1, t_2)$, at t_1 during $[t_1, t_2]$, based on the model and X_{0,t_1}. In this figure, t_m is total available test time for a single module. Note that it is possible to have $t_2 = t_m$ (i.e., the prediction is made to the end of the test period).

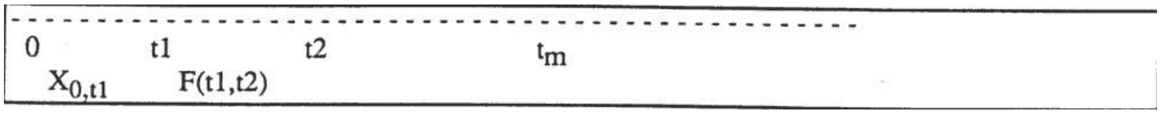


Figure C.1—Reliability prediction time scale

Based on the updated predictions, test resources may be reallocated. Of course, it could be disruptive to the organization to reallocate too frequently. So, prediction and reallocation could occur at major milestones (i.e., formal review of test results).

Using the Schneidewind SR model (see 5.3.1) and the Space Shuttle Primary Avionics Software Subsystem as an example, the process of using prediction for allocating test resources is developed. Two parameters, α and β, which will be used in the following equations, are estimated by applying the model to X_{0,t_1}. Once the parameters have been established, various quantities can be predicted to assist in allocating test resources, as shown in the following equations:

— The number of failures during $[0,t]$ follows in Equation (C.1):

$$F(t) = \frac{\alpha}{\beta}\left(1 - e^{-\beta t}\right) \tag{C.1}$$

— Using Equation (C.1) and Figure C.1, you can predict the number of failures during $[t_1, t_2]$ as follows in Equation (C.2):

$$F(t_1, t_2) = \frac{\alpha}{\beta}\left(1 - e^{-\beta t_2}\right) - X_{0,t_1} \tag{C.2}$$

— Also, the maximum number of failures during the life ($t = \infty$) of the software can be predicted as follows in Equation (C.3):

$$F(\infty) = \frac{\alpha}{\beta} + X_{s-1} \tag{C.3}$$

— Using Equation (C.3), the maximum remaining number of failures at time *t* can be predicted as follows in Equation (C.4):

$$RF(t) = \frac{\alpha}{\beta} + X_{s-1} - X(t) \tag{C.4}$$

Given *n* modules, allocate test execution time *periods* T_i for each module *i* according to the following Equation (C.5):

$$T_i = \frac{F_i(t_1 t_2)\, n(t_2 - t_1)}{\sum_{i=1}^{n} F_i(t_1, t_2)} \tag{C.5}$$

In Equation (C.5), note that although predictions are made using Equation (C.2) for a *single module,* the total *available* test execution time $(n)(t_2 - t_1)$ is allocated for each module *i* across *n modules.* Use the same interval $0, 20$ for each module to estimate α and β and the same interval $20, 30$ for each module to make predictions, but from then on a variable amount of test time T_i is used depending on the predictions.

Table C.1 and Table C.2 summarize the results of applying the model to the failure data for three Space Shuttle modules (OI). The modules are executed continuously, 24 h/day, day after day. For illustrative purposes, each period in the test interval is assumed to be equal to 30 days. After executing the modules during $0, 20$, the SMERFS program was applied to the observed failure data during $0, 20$ to obtain estimates of α and β. The total number of failures observed during $0, 20$ and the estimated parameters are shown in Table C.1.

Table C.1—Observed failures and model parameters

	$X(0, 20)$ failures	α	β
Module 1	12	1.69	0.13
Module 2	11	1.76	0.14
Module 3	10	0.68	0.03

Table C.2—Allocation of test resources

	$F(\infty)$ failures	$F(20,30)$ failures	$R(20)$ failures	T periods	$X(20,20+T)$ failures
Module 1					
Predicted	12.95	0.695	0.952	7.6	—
Actual	13.00	0.000	1.000	—	0
Module 2					
Predicted	12.5	1.32	1.5	14.4	—
Actual	13.0	1.32	2.0	—	1
Module 3					
Predicted	10.81	0.73	0.81	8.0	—
Actual	14.00	1.00	4.0	—	1

Equation (C.2) was used to obtain the predictions in Table C.2 during 20, 30. The prediction of $F(20,30)$ led to the prediction of T, the allocated number of test execution time periods. The number of additional failures that were subsequently observed, as testing continued during 20, 20÷T, is shown as $X(20,20+T)$. Comparing Table C.1 with Table C.2, it can be seen that there is the possibility of additional failures occurring in Module 1 (0.95 failures) and Module 2 (0.50 failures), based on predicted maximum number of failures $F(\infty)$. That is, for these modules, $[X(0,20)+X(20,30+T)]<F(\infty)$. Note that the actual $F(\infty)$ would only be known after all testing is complete and was not known at 20+T. Thus additional procedures are needed for deciding how long to test to reach a given number of remaining failures. A variant of this decision is the stopping rule (when to stop testing?). Making test decisions is discussed as follows in C.2.

C.2 Making test decisions

In addition to allocating test resources, reliability prediction can be used to estimate the minimum total test execution time t_2 (i.e., interval [0, t_2]) necessary to reduce the predicted maximum number of remaining failures to $R(t_2)$. To do this, subtract Equation (C.1) from Equation (C.3), set the result equal to $R(t_2)$, and solve for t_2 as follows in Equation (C.6):

$$t_2 = \frac{1}{\beta} \ln \frac{\alpha}{\beta R(t_2)} \tag{C.6}$$

where $R(t_2)$ can be established from Equation (C.7) as follows:

$$R(t_2) = (p)\frac{\alpha}{\beta} \tag{C.7}$$

where p is the desired fraction (percentage) of remaining failures at t_2. Substituting Equation (C.7) into Equation (C.6) gives Equation (C.8) as follows:

$$t_2 = \frac{1}{\beta} \ln \frac{1}{p} \tag{C.8}$$

Equation (C.8) is plotted for Module 1 and Module 2 in Figure C.2 for various values of p.

You can use Equation (C.8) as a rule to determine when to stop testing a given module.

Using Equation (C.8) and Figure C.2, you can produce Table C.3, which tells the following: the total minimum test execution time t_2 from time 0 to reach essentially 0 remaining failures [i.e., at $p = 0.001$ (0.1%), the predicted remaining failures of 0.01295 and 0.01250 for Module 1 and Module 2, respectively (see Equation (C.7) and Table C.3]; the additional test execution time beyond 20+T shown in Table C.3; and the actual amount of test time required, starting at 0, for the "last" failure to occur (this quantity comes from the data and not from prediction). You do not know that it is necessarily the last; you only know that it was the "last" after 64 periods (1910 days) and 44 periods (1314 days) for Module 1 and Module 2, respectively. So, $t_2 = 52.9$ and $t_2 = 49.0$ periods would constitute your stopping rule for Module 1 and Module 2, respectively. This procedure allows you to exercise control over software quality.

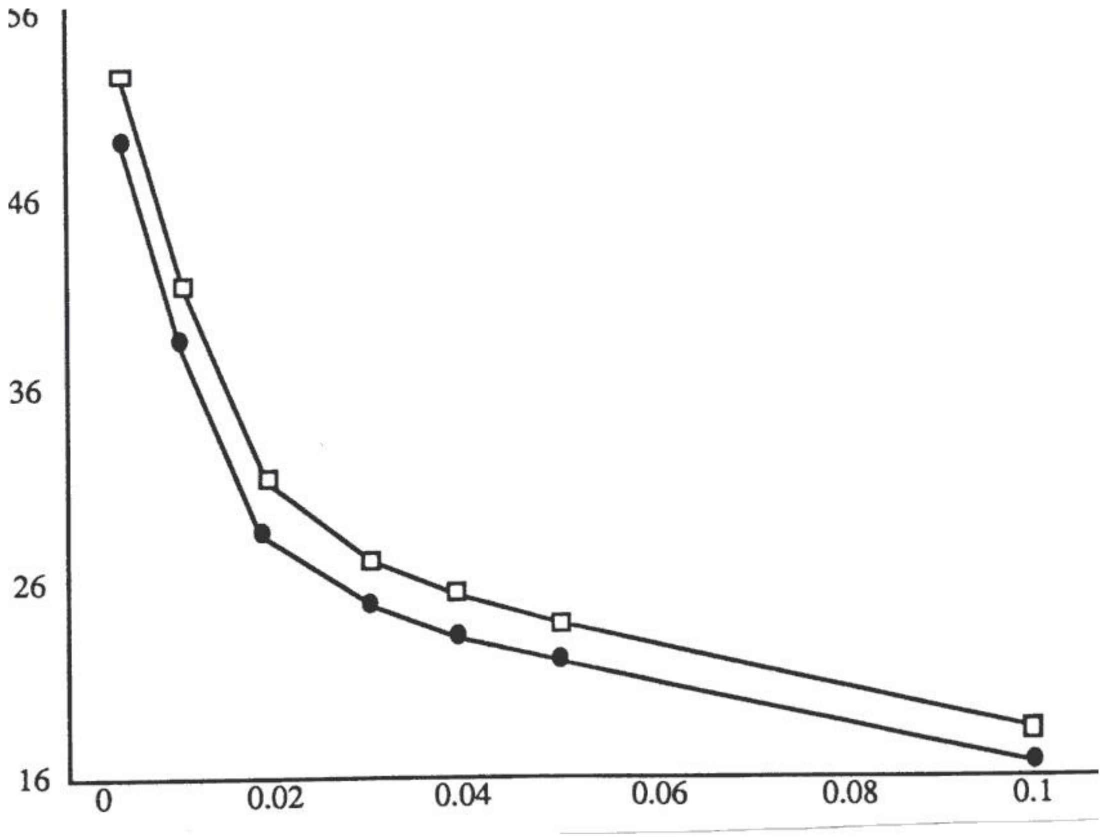


Figure C.2—Execution time in periods needed to reach the desired fraction of remaining failures (Module 1: upper curve, Module 2: lower curve)

Table C.3—Test time to reach "0" remaining failures (p = 0.001)

	T_2 periods	**Additional test time periods**	**Last failure found periods**
Module 1	52.9	45.3	64
Module 2	49.0	34.6	44

Annex D

(informative)

Automated software reliability measurement tools

This annex contains Table D.1,[7] which provides a list of the two most popular SR measurement tools.

Table D.1—Most popular SR measurement tools

Supplier	Naval Surface Warfare Center (NSWC/DD)	Open Channel Foundation
Tool name	Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS)	Computer Aided Software Reliability Estimation (CASRE)
Models	1. Littlewood/Verrall 2. Musa basic 3. Musa/Okumoto 4. Geometric 5. Execution time NHPP 6. Generalized Poisson NHPP 7. Brooks/Motley 8. Schneidewind 9. S-shaped	

[7] SMERFS is also available at http://www.slingcode.com/smerfs/.

Annex E

(informative)

A comparison of constant, linearly decreasing, and exponentially decreasing models

This annex discusses constant, linearly decreasing, and exponentially decreasing error removal models and their associated reliability models. Various techniques are discussed for evaluating the model constants from error removal and test data. This annex also compares the three models and discusses simplified techniques that are useful for rapid evaluation during the first few months of testing. These techniques are also useful as a check even if more elaborate estimation techniques are to be used. This annex can also be a help to those who are new to the field of SR since the methods are more transparent and less mathematics is required.

The simplest model is the constant error removal model. However, if the removal rate stays constant after sufficient error debugging, all errors will be removed. We know this is not possible so the model is only useful early in the integration process where there are still many errors and the error removal process is mainly limited by the number of testing personnel. Still, the mathematics are simple and the model will probably be satisfactory for a few months of testing.

The linearly decreasing model is an improvement over the constant error removal model since it attempts to follow the observed fact that it becomes harder to find errors as testing progresses. This is probably due to the fact that the initial errors found are in program execution paths that are exercised in testing regular program features. Once these initial errors are found, the test cases must become more creative so that they exercise less common features and combinations of features to uncover additional errors. The problem with the linearly decreasing model is that eventually the error removal rate becomes zero. If the reliability function or the mean time between failures satisfies the software specifications before the error removal rate decreases to zero, the model should be satisfactory. There is always the possibility of starting with a constant error removal model for a few months if the error removal rate appears to be personnel-limited and later switching to a linearly decreasing error removal model.

The exponentially decreasing model is perhaps the best of the group since it eliminates the defects of the other two models—error removal rates declines and only goes to zero as the debugging time approaches infinity. The following example describes a simplified means of evaluating the constants of the exponentially decreasing error removal model. Of course, one could also use a constant error removal model for the first few months followed by an exponentially decreasing model.

E.1 A simplified means of early evaluation of the model parameters

In addition to the parameter estimation techniques given previously, least squares and maximum likelihood, there are simplified methods that are easier to use in the first month or two of integration testing. These simplified techniques are also useful as a check on the more complex methods and may provide additional insight.

Two sets of data are generally available—error removal data and simulation test data. Generally, there is more error removal data, thus such data should be used as much as possible. It is easiest to explain these methods by referring to an illustrative, hypothetical example, the software for a computer-controlled telescope. The requirements for the example are given as follows.

E.1.1 Requirements—Computerized telescope pointing and tracking system

Professional and advanced amateur telescopes have had computer-controlled pointing and tracking systems for many years. Around 1998, technology advanced to the point where a computer-controlled telescope could be sold for $400 to $600. Such a telescope has good quality advanced optics, a rigid tripod with a compass and bubble level, a mounting system with drive motors about two axes, and a computerized hand controller about the size of a TV remote with an LCD screen and control buttons. Operation has three major modes: alignment, pointing, and tracking.

E.1.1.1 Alignment

To align the telescope, the user levels the tripod with the bubble level and points the telescope north using the compass. The user inputs the zip code and time. The computer determines a visible bright star, points the telescope toward it, and the user checks to see if the star is in the center of the view through the eyepiece. If not, small manual adjustments are made to center the image. This is repeated with two other stars and the telescope is then aligned.

E.1.1.2 Pointing

An advanced amateur can use a simple sky chart and their knowledge of the sky to aim the telescope at objects. However, most users will use the automatic system, especially in an urban location where the sky is not dark or clear. The user consults the controller database with 1500 objects and chooses a stored object such as a planet, star, or nebula using the controller. If the object is not visible, the information is displayed on the hand controller and a different selection is made. For a visible object, the controller provides drive signals to point the telescope at the object and the operator makes any final manual adjustments to center the image.

E.1.1.3 Tracking

Since the earth is rotating, the object drifts out of the visual field, within a fraction of a minute at high powers, and counter rotation about the two mounting axes is needed to stabilize the image. The computer in the hand controller generates these tracking inputs. One can use astrophotography to capture a picture of the stabilized image.

E.1.1.4 Other functions

Other functions store the objects viewed during a session, allow one to choose among objects in the database, and provide ways to turn off the drive signals for terrestrial viewing in manual mode. The controller can select a “tour of tonight’s best objects,” and if the user wants to know more about an object, the database can display details like distance, temperature, mass, and historical information.

E.1.2 Reliability goals

A good way to set an MTBF goal would be if we knew about the performance of previous computer-controlled telescope systems. Assuming we had field performance data for such systems along with subjective user ratings as to whether such systems performed with few errors or had significant bugs, we could set firm goals. However, let us assume that there are no such systems or we do not have access to field performance data. One could set the reliability goal by estimating how often the user could tolerate software error occurrence. Suppose that the skies are only clear enough for professional viewing 75% of the time and that a major telescope is scheduled for five days per week of viewing if the sky permits. If a viewing session is 6 h long, the telescope is in use $5 \times 4 \times 6 \times 0.75 = 90$ h/month. One would expect that one error per month might be acceptable, i.e., a 90 h MTBF for professional use. However, an amateur user would also consider the software annoying if one error per month occurred. Suppose that the amateur uses the telescope one night a week for 2 h. This accounts for poor viewing nights as well as availability of the viewer. Thus, 2 h per week times four weeks per month is 8 h, and one error per month is an 8 h MTBF for an amateur. Thus, somewhere in the range 8 h to 90 h MTBF would be an acceptable MTBF goal.

E.1.3 Reliability data

Let us assume that we have the following hypothetical error removal data for development of the telescope software: After two weeks, 50 errors are removed and after one month, 90 errors are removed. In addition, simulation test data was used after one month and in 10 h of testing, two errors occurred. Similarly, simulation test was performed after two months and in 8 h of testing one error was found. This data is used below to formulate a constant error removal model and an exponential decreasing error removal model.

E.1.3.1 Constant error removal model

Suppose we are to make a constant error removal model where the failure rate (hazard) is given by $z(\tau) = k[E_T - \rho_0\tau]$. The reliability function is given by $R(t) = \exp(-k[E_T - \rho_0\tau]t)$, and the mean time between failure function is given by $\text{MTBF} = 1/k[E_T - \rho_0\tau]$.

From the error removal data for the example, the error removal rate $\rho_0 = 90$ errors per month. Thus, one parameter has been evaluated. We can evaluate the other two parameters from the simulation test data by equating the MTBF function to the test data.

After one month $\text{MTBF} = 10/2 = 5 = 1/k[E_T - 90 \times 1]$.

After two months $\text{MTBF} = 8/1 = 8 = 1/k[E_T - 90 \times 2]$.

Dividing one equation by the other cancels k and allows one to solve for E_T, yielding $E_T = 330$. Substitution of this value into the first equation yields $k = 0.0008333$ and $1/k = 1200$.

The resulting functions are as follows in Equation (E.1), Equation (E.2), and Equation (E.3):

$$z(\tau) = k[E_T - \rho_0\tau] = 0.0008333[330 - 90\tau] \tag{E.1}$$

$$R(t) = \exp(-k[E_T - \rho_0\tau) = \exp(-0.0008333[330 - 90\tau]t) \tag{E.2}$$

$$\text{MTBF} = 1/k[E_T - \rho_0\tau] = 1200/[330 - 90\tau] \tag{E.3}$$

E.1.3.2 An exponentially decreasing error model

The equations for the exponentially decreasing error model are given as follows in Equation (E.4) through Equation (E.7):

Error removal model $$E_r(\tau) = E_T e^{-\alpha\tau} \tag{E.4}$$

Hazard (failure rate) $$z(\tau) = kE_T e^{-\alpha\tau} \tag{E.5}$$

Reliability function $$R(t) = e^{-(kE_T e^{-\alpha\tau})t} \tag{E.6}$$

MTBF $$\text{MTBF} = \frac{1}{kE_T e^{-\alpha\tau}} \tag{E.7}$$

The parameters to be estimated are E_T, α, and k. The variable t is the operating time and the variable τ is the number of months of testing. In the unusual and very fortunate case where the company has an extensive private reliability database, all the parameters can be initially estimated by choosing data from a few past similar models. Occasionally, enough data exists in the literature for initial evaluation. In the material to follow, we assume that no prior data exists, and we make early estimates of the three parameters from error removal and functional test data. Two approaches are possible—one based on matching error removal rates and the other on actual number of errors removed. (We will show shortly that the second procedure is preferable.)

E.1.3.3 Matching rates

Note that the data on errors removed per month is the error removal rate. Thus the error removal rate is the derivative of Equation (E.4):

$$\dot{E}_r = \frac{d}{d\tau}[E_r(\tau)] = \frac{d}{d\tau}[E_T e^{-\alpha\tau}] = -\frac{E_T}{\alpha} e^{-\alpha\tau} \tag{E.8}$$

Since errors are removed the error rate is negative, and substitution of a negative value for $\dot{E}_r$ will cancel the negative sign in Equation (E.8). Suppose that we have error removal data for the first month, we can take the average value and assume that this occurs at the middle at 1/2 month, and substitute it in Equation (E.8) as follows in Equation (E.9):

$$\dot{E}_r(1/2) = \frac{d}{d\tau}[E_r(1/2)] = -\frac{E_T}{\alpha} e^{-\alpha/2} \tag{E.9}$$

After the second month we have another error removal value and we can assume that this occurs at mid interval = 3/2, yielding the following in Equation (E.10):

$$\dot{E}_r(3/2) = \frac{d}{d\tau}[E_r(3/2)] = -\frac{E_T}{\alpha} e^{-3\alpha/2} \tag{E.10}$$

Given Equation (E.9) and Equation (E.10), we can solve for the two constants. In our example, we have data for error removal for two weeks and four weeks not one month and two months, thus we will fit these values at the mid points after (1/4) and (1/2) months and Equation (E.9) and Equation (E.10) are fitted at (1/4) and (1/2) months. If we divide the two equations, we obtain the following in Equation (E.11):

$$\frac{\dot{E}_r(1/4)}{\dot{E}_r(1/2)} = \frac{-\frac{E_T}{\alpha} e^{-\alpha/4}}{-\frac{E_T}{\alpha} e^{-\alpha/2}} = e^{-\alpha/4} \tag{E.11}$$

If we take the natural log of both sides of Equation (E.11), we obtain a value for α as follows in Equation (E.12) and Equation (E.13):

$$\alpha = -4(\ln[\dot{E}_r(1/4)] - \ln[\dot{E}_r(1/2)]) \tag{E.12}$$

$$\alpha = -4(\ln[50/0.5] - \ln[90/1]) = 0.4214 \tag{E.13}$$

Substituting this value of α into Equation (E.9), where the error removal rate for 1/2 month is the average number of errors removed over the first month, 90 yields the following in Equation (E.14):

$$\dot{E}_r(1/2) = -90 = -\frac{E_T}{0.4214} e^{-0.4214/2} \tag{E.14}$$

Solving for ET yields 46.83. Note that this seems to be too small a value for E_T since one has already removed 90 errors from the software.

We now have only one more parameter to evaluate, *k*. Using the simulation data and substituting into Equation (E.7) yields the following in Equation (E.15):

$$5 = \text{MTBF}(\tau = 1) = \frac{1}{kE_T e^{-\alpha \bullet 1}} = \frac{1}{k(46.83e^{-0.4214})} \tag{E.15}$$

Solving for *k* we obtain

$$k = 0.006509 \tag{E.16}$$

$$\text{MTBF} = \frac{1}{kE_T e^{-\alpha\tau}} = 3.2807\text{e}^{0.4214\tau} \tag{E.17}$$

We now evaluate the constants of the model using errors removed and not rates.

E.1.3.4 Matching errors removed

There is another simpler way to solve for the parameters of the exponential model. We start with Equation (E.4) and after two weeks the number of errors removed is as follows in Equation (E.18):

$$50 = E_T - E_T e^{-\alpha/2} \tag{E.18}$$

and after 1 month as follows in Equation (E.19):

$$90 = E_T - E_T e^{-\alpha/1} \tag{E.19}$$

Let $e^{-\alpha/2} = x$, then $e^{-\alpha} = x^2$ and Equation (E.18) and Equation (E.19) become Equation (E.20) and Equation (E.21), respectively, as follows:

$$50 = E_T - E_T x \tag{E.20}$$

$$90 = E_T - E_T x^2 \tag{E.21}$$

Solving Equation (E.20) for *x* and substituting into Equation (E.21) yields the following in Equation (E.22):

$$90 = 50\left[\frac{1-x^2}{1-x}\right] \tag{E.22}$$

Equation (E.22) is a quadratic equation in *x* as follows in Equation (E.23):

$$x^2 - (9/5)x - (4/5) = 0 \tag{E.23}$$

Solving for *x* yields the values *x* =1 and *x* = 0.8. Equating these values to $e^{-\alpha/2}$ and solving yields values for $\alpha = 0$ and 0.4463. Discarding the 0 value, and substituting $\alpha = 0.4463$ into Equation (E.18) yields a value of $E_T = 250$.

To evaluate *k*, we equate the MTBF at one month to Equation (E.17). Thus, Equation (E.24) follows:

$$\text{MTBF} = \frac{1}{kE_T e^{-\alpha\tau}} = 5 = \frac{1}{k250e^{-0.4463}} \tag{E.24}$$

Solving for *k* yields 0.00125 and the MTBF becomes as follows in Equation (E.25):

$$\text{MTBF} = 3.2e^{0.4463\tau} \tag{E.25}$$

E.1.4 Comparison of the three models

A simple means of comparing the three models is to plot the three MTBF functions and compare them. A comparison of the three functions is given in the table below:

Table E.1—Comparison of the three MTBF functions

τ	Constant [Equation (E.3)]	Exponential no. 1 [Equation (E.17)]	Exponential no. 2 [Equation (E.25)]
	MTBF = 1200/[330 − 90τ]	MTBF = $3.2807e^{0.4214\tau}$	MTBF = $3.2e^{0.4463\tau}$
0	3.6	3.2807	3.2
1	5	5.0	5.0
2	8	7.62	7.81
3	20	11.62	12.2
3.666	∞	15.38	16.4
4	—	17.70	19.1
5	—	26.98	29.8
6	—	41.12	46.6
7	—	62.67	72.8
8	—	95.51	113.7

The MTBF function is plotted versus the test time for the three models—constant, exponential 1, exponential 2.

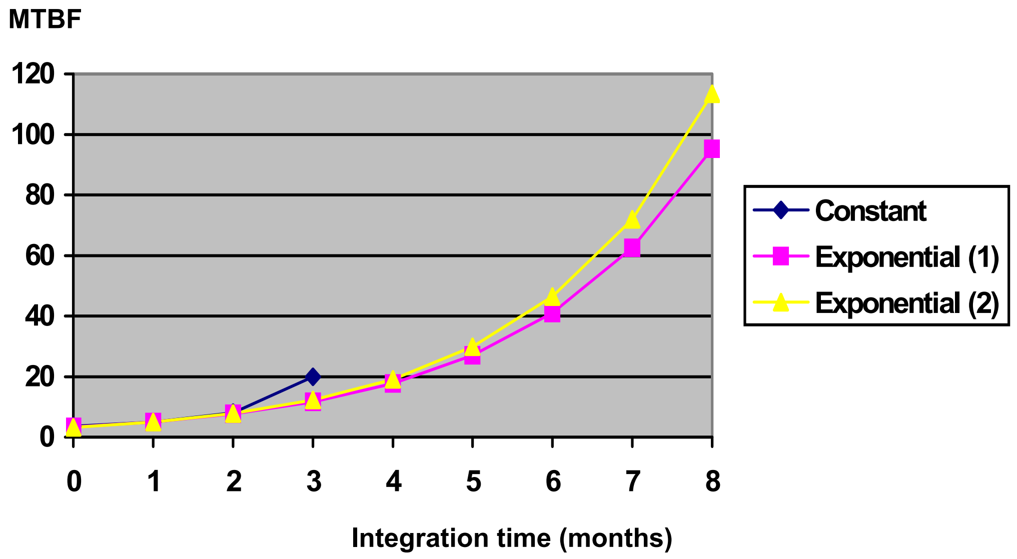


Figure E.1—MTBF function is plotted versus the test time

For the first few months, the three models give similar results. However, the constant error removal model becomes unrealistic after three months and the MTTF begins to approach infinity. The exponential models yield very similar graphs and are more realistic, and it takes about two months to reach a MTTF goal of 8 h, which was our goal for amateur usage. As was previously stated, the parameter determined for E_T in the first exponential model is too low; however, the model does yield a proper MTBF function. Because of this anomaly, it seems that the second method of evaluating the model parameters using errors removed rather then removal rates is superior. It not only yields an MTBF curve that agrees with the data, but it also gives believable results for the error value, E_T.

We now compare the values obtained for the parameter E_T. For the constant error removal model this constant is 330. For the exponential model where we use the rate matching approach, $E_T = 46.76$, which is too low. For the second method, which is based on error matching, $E_T = 250$. If we had an extensive database, we could check this with other projects. However, lacking such data we use the values given by Shooman on pp. 323–329 of *Reliability of Computer Systems and Networks, Fault Tolerance, Analysis, and Design* [3][8] and by Musa on p. 116 of *Software Reliability Measurement, Prediction, Application* [B54]. These data show that E_T is related to the SLOC, and is 0.006 to 0.02 of the SLOC value. From a student project for CS 607, Fall 2005 conducted by Maria Riviera and Vic Emile on a telescope hand controller design, the estimated value of SLOC was 13 038. Thus, $0.006 \times 13\,038 < E_T < 0.02 \times 13\,038$, yielding $78 < E_T < 260$, which agrees roughly with the value of 330 for the constant model and 250 for the second exponential model. Again, this shows that the value of 46.76 for the first exponential model is a poor value.

Our conclusion from this preliminary analysis is that it will take about two to three months to reach the MTBF goal of 8 h for amateur use and about eight months to reach a MTBF of 90 h, which is our professional goal. Remember the error removal data and the test data given above is HYPOTHETICAL data made up for the purposes of illustration. After several months of testing there will be more data and the least squares and maximum likelihood techniques can be used to give better estimates of the model parameters and to obtain more accurate predictions. For more details, see *Reliability of Computer Systems and Networks, Fault Tolerance, Analysis, and Design* [3] and *Software Failure Risk: Measurement and Management* [4].

E.2 Supporting published material

[1] Musa, J. D., et al. *Software Reliability Measurement, Prediction, Application,* New York: McGraw-Hill, 1987.

[2] Shooman, M. L., *Software Engineering Design, Reliability, Management,* New York: McGraw-Hill, 1983.

[3] Shooman, M. L., *Reliability of Computer Systems and Networks, Fault Tolerance, Analysis, and Design,* New York: McGraw-Hill, 2002.

[4] Sherer, S. A., *Software Failure Risk: Measurement and Management,* New York: Plenum Press, 1992.

[8] In this annex, the numbers in brackets correspond to those of supporting published material in E.2.

Annex F

(informative)

Software reliability prediction tools prior to testing

Software failure rates are a function of the development process used. The more comprehensive and better the process is, the lower the fault density of the resulting code. There is an intuitive correlation between the development process used and the quality and reliability of the resulting code as shown in Figure F.1. The software development process is largely an assurance and bug removal process, and 80% of the development effort is spent removing bugs. The greater the process and assurance emphasis, the better the quality of the resulting code, which is shown in Figure F.1. Several operational, field data points have been found to support this relationship. The operational process capability is measured by several techniques (see Capability Maturity Model® (CMM®) [B6][9, 10, 11]). Process measures of operational process capability can be used to project the latent fault content of the developed code.

F.1 Keene's development process prediction model (DPPM)

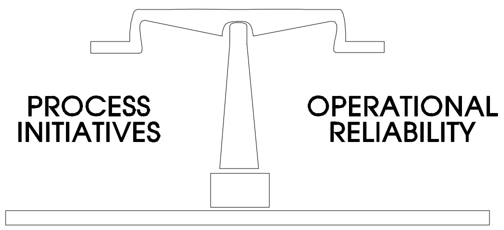


Figure F.1—Illustrating the relationship between process initiatives (capability) and operational reliability

Figure F.2 illustrates the defect density rate improves (decreases) as the development team's capability improves. Also, the higher the maturity level, development organizations will have a more consistent process, resulting in a tighter distribution of the observed fault density and failure rate of the fielded code. They will have less outliers and have greater predictability of the latent fault rate.

The shipped defects are removed as they are discovered and resolved in the field. It has been shown fielded code can be expected to improve exponentially over time (Cole and Keene [3], Keene [5], Keene [6], "New System Reliability Assessment Method" [9]) until it reaches a plateau level when it stabilizes.[12] Chillarege has reported failure data on a large operating system revealed the code stabilized after four years of deployment on the initial release and two years on subsequent releases (Chillarege [2]).

The projection of fault density according to the corresponding Software Engineering Institute (SEI) level is now shown in Table F.1. These fault density settings are based upon the author's proprietary experience with a dozen programs. This spans all of the SEI categories, except for lacking a data point for SEI level-IV programs. The SEI level V is set based upon the Space Shuttle's published performance.

[9] Process improvement models that meet these criteria include the SEI CMM® model [B6].
[10] Capability Maturity Model and CMM are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
[11] This information is given for the convenience of users and does not constitute an endorsement by the IEEE of these models. Equivalent models may be used if they can be shown to lead to the same results.
[12] In this annex, the numbers in brackets correspond to those of supporting published material in F.5.

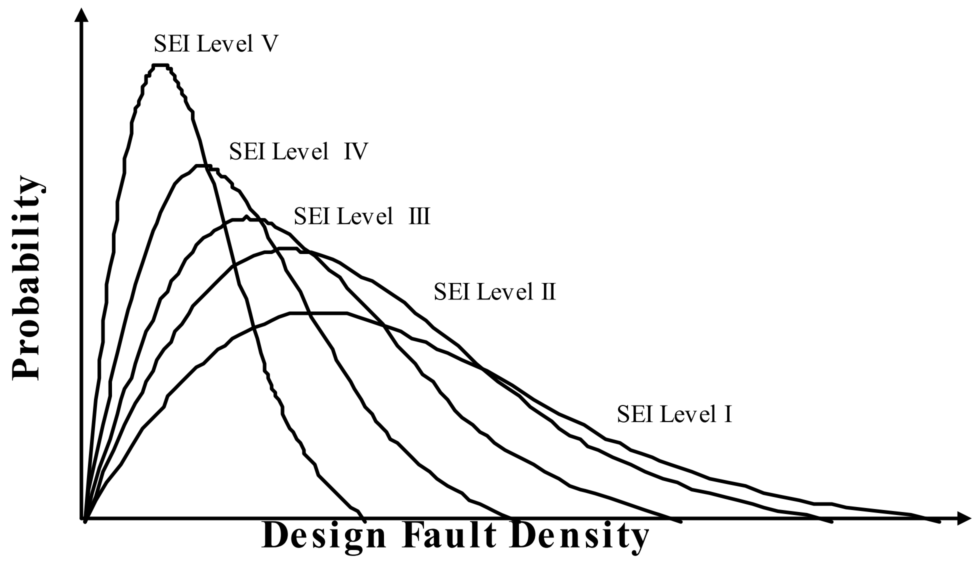


Figure F.2—Illustrating projected design defect density as a function of the development organization's design capability, as measured in terms of CMM capability

The latent fault densities at shipment are shown as a function of the SEI development maturity level in Table F.1.

Table F.1—Industry data prediction technique

SEI'S CMM level	Maturity profile Nov. 1996 542 Organizations	Design fault density faults/KSLOC (all severities)	Defect plateau level for 48 months after initial delivery or 24 months following subsequent deliveries
5	Optimizing: 0.4%	0.5	1.5%
4	Managed: 1.3%	1.0	3.0%
3	Defined: 11.8%	2.0	5.0%
2	Repeatable: 19.6%	3.0	7.0%
1	Initial: 66.9%	5.0	10.0%
unrated	The remainder of companies	>5.0	not estimated

Keene's development process prediction model (DPPM) correlates the delivered latent fault content with the development process capability. This model can be used in the program planning stages to predict the operational SR. The model requires user inputs of the following parameters:

— Estimated KSLOCs of deliverable code

— SEI capability level of the development organization

— SEI capability level of the maintenance organization

— Estimated number of months to reach maturity after release (historical)

— Use hours per week of the code

— *Percent fault activation (estimated parameter)* represents the average percentage of seats of system users that are likely to experience a particular fault. This is especially important (much less than

100%) for widely deployed systems such as the operating system AIX that has over a million seats. This ratio appears to be a decreasing function over time that the software is in the field. The early-discovered faults tend to infect a larger ratio of the total systems. The later removed faults are more elusive and specialized to smaller domains, i.e., have a smaller, and stabilizing, fault activation ratio. A fault activation level of 100% applies when there is only one instance of the system.

- *Fault latency* is the expected number of times a failure is expected to reoccur before being removed from the system. It is a function of the time it takes to isolate a failure, design and test a fix, and field the fix that precludes its reoccurrence. The default on this parameter is as follows:
  - SEI level V: Two reoccurrences
  - SEI level III and level IV: Three reoccurrences
  - SEI level I and level II: Five reoccurrences
- Percent severity 1 and severity 2 failures (historical)
- Estimated recovery time (MTTR) (historical)

F.2 Rayleigh model

The Rayleigh model uses defect discovery rates from each development stage, i.e., requirements review, high level design inspection, etc., to refine the estimate the latent defect rate at code delivery. This model projects and refines the defect discovery profile improving the projection of the estimated number of defects to be found at each succeeding development stage up to product release. One popular implementation of the Rayleigh model is the software error estimation procedure (SWEEP) released by Systems and Software Consortium, Inc. (SSCI).

NOTE—For example, the executable code for the Rayleigh model is provided in *Metrics and Models in Software Quality Engineering* [4].

The input data are the defect discovery rates found during the following development stages: high level design, low level design, code and unit test, software integration, unit test and system test. The defect discovery profile is illustrated in Figure F.3.

The SWEEP model refines the initial Keene model process-based estimate. The Figure F.4 shows the reliability growth curve from the Keene model can be beneficially applied to the latent error estimate of the SWEEP model.

NOTE 1—The reliability estimate provided by the Keene model gives an early (in development) reliability estimate to SR. This initial estimate can next be subsequently refined by the Rayleigh model incorporating actual development data defect rates collected at each process stage of development, i.e., requirements, high level design, low level design, code, software integration and test, system test.

NOTE 2—The Rayleigh model's projected latent defect density can then be extrapolated forward in time using the Keene model fault discovery and plateau profile. This is shown in Figure F.4 and explained in the following paragraph.

Figure F.4 illustrates the data fusion of the prediction models. Prior to the three steps in this process, there is a step involving an *a priori* estimate of the latent fault rate and its associated field failure rate of the code. This is accomplished by the Keene process based prediction model. Once the code is under development and subjected to inspections, reviews, and tests, the Rayleigh model can better map the actual projected defects. This prediction process is further tuned by applying the exponential fault discovery and removal profile of the Keene model.

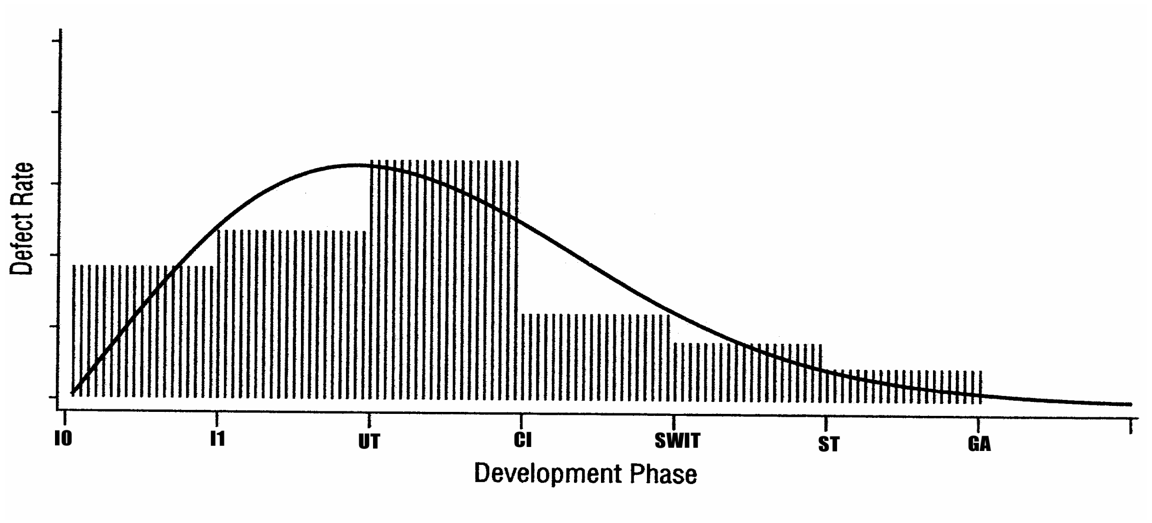


Figure F.3—Illustrative Rayleigh defect discovery profile over the development stages

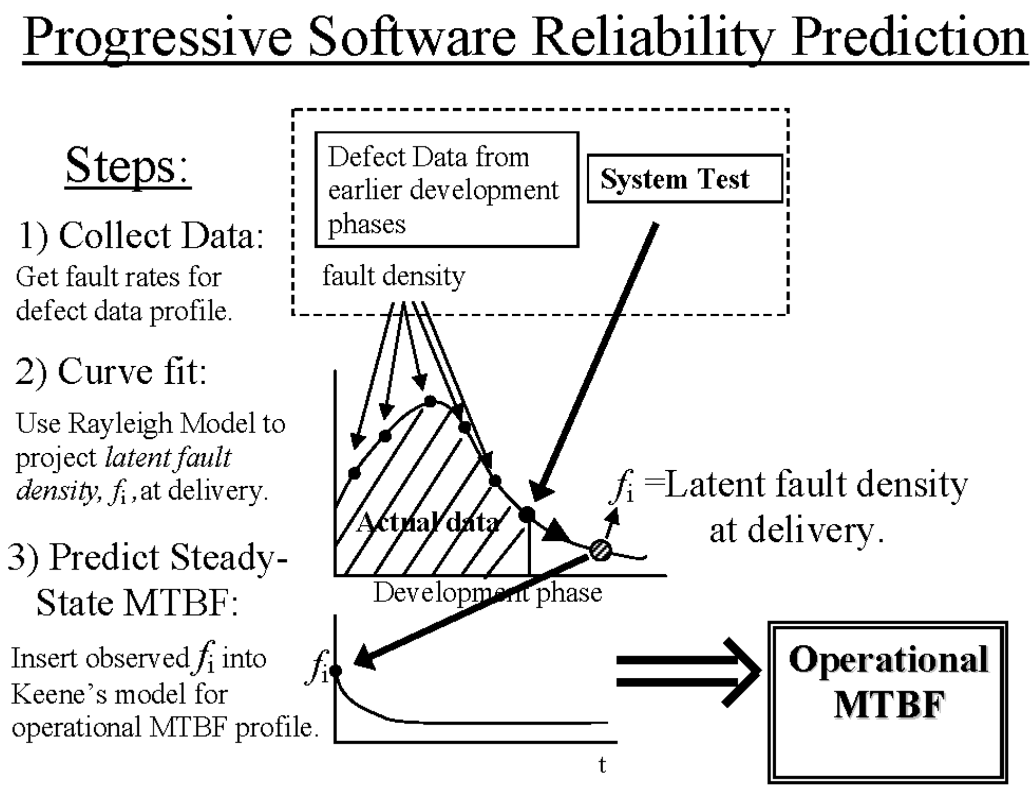

Progressive Software Reliability Prediction
Steps:
1) Collect Data:
Get fault rates for defect data profile.
2) Curve fit:
Use Rayleigh Model to project latent fault density, f_i, at delivery.
3) Predict Steady-State MTBF:
Insert observed f_i into Keene's model for operational MTBF profile.
Defect Data from earlier development phases
System Test
fault density
Actual data
Development phase
f_i =Latent fault density at delivery.
f_i
t
Operational MTBF

Figure F.4—Progressive SR prediction

F.3 Application of Keene and Rayleigh models

F.3.1 Software reliability estimates

The concerted application of the reliability models, DPPM, SWEEP, and CASRE, makes the best use of whatever data is available to predict the code reliability at the various stages of development. The reliability models in the CASRE tool suite are preferred for SR prediction, but these models require operational code. Their predictions are based upon intra fail times. The Keene DPPM provides an *a priori* reliability estimate that is largely based upon the capability level of the developing organization. It also provides a fault discovery and removal profile that converts the latent fault content at delivery to an observed user-experienced failure rate. The Rayleigh model correlates the rate of defects found throughout the development process to the expected latent error content. So the Keene model, the Rayleigh model, and the CASRE tool suite are progressively applied improving the failure rate estimate of the code.

The Keene DPPM gives the developing organization an early estimation of the risk that code will not meet reliability expectations or to assure the delivery of more reliable code. It applies more attention to the development history to better estimate the model parameters such as the expected failure latency of the code. The development organization can quantify its opportunity for delivering more reliable code by improving its development capability. Aerospace- and military-based companies usually know their SEI capability level so applying the Keene DPPM is a straightforward process. Commercial companies can look at the SEI CMM rating criteria and reasonably estimate their current capability level and also see what they can do to improve their process. There is the old axiom: “what gets measured, gets improved.” The Keene DPPM just quantifies the expected return of investment in terms of reliability improvement for investing in process improvement.

The SWEEP model allows the developer to critically examine the quality of his process prior to test. It makes use of the discovery rate of defects found in reviews and inspections throughout the development process. The projected latent defect content will be best when there is a lower profile of defects found and when the rate of defect discovery is peaked earlier in the development process.

The CASRE reliability tool suite gives the best estimate of operational reliability of the code since it is based upon observing actually operating code. There is a caveat here. Software improves during test as defects are exposed and removed. It is often said that “software learns to pass its tests.” This is true and also beneficial to the end user so long as the testing adequately represents the end user’s actual application. So the quality of the testing experience depends on how well the customer’s operational profile is known and replicated in the test suite. There is another limitation to basing the operational failure rate on actual test data. That is, the major problem of field failures lies in requirements deficiencies. The testing’s purpose is to verify that the code meets the product specifications. Requirements problems that escape the specification will not be caught in test. So each reliability prediction method has its limitations as well as its application strengths.

There is benefit in using all of these reliability models and combining the results as depicted in Figure F.4. They provide a reliability focus at each point in the development process.

F.3.2 Development process model

Government contractors have made use of the Keene DPPM for over a decade and have reported favorable results (Bentz and Smith [1], Peterson [7], Peterson et al. [8], Smith [10]) This model is best used in conjunction with SWEEP and CASRE as previously stated. The use of the model is straightforward and intuitive. It calls out experience factors for the developer to pay attention in the development process, such as the failure rate latency. The failure rate latency is the expected number of times a failure is likely to occur before it is isolated to a fault and that fault removed.

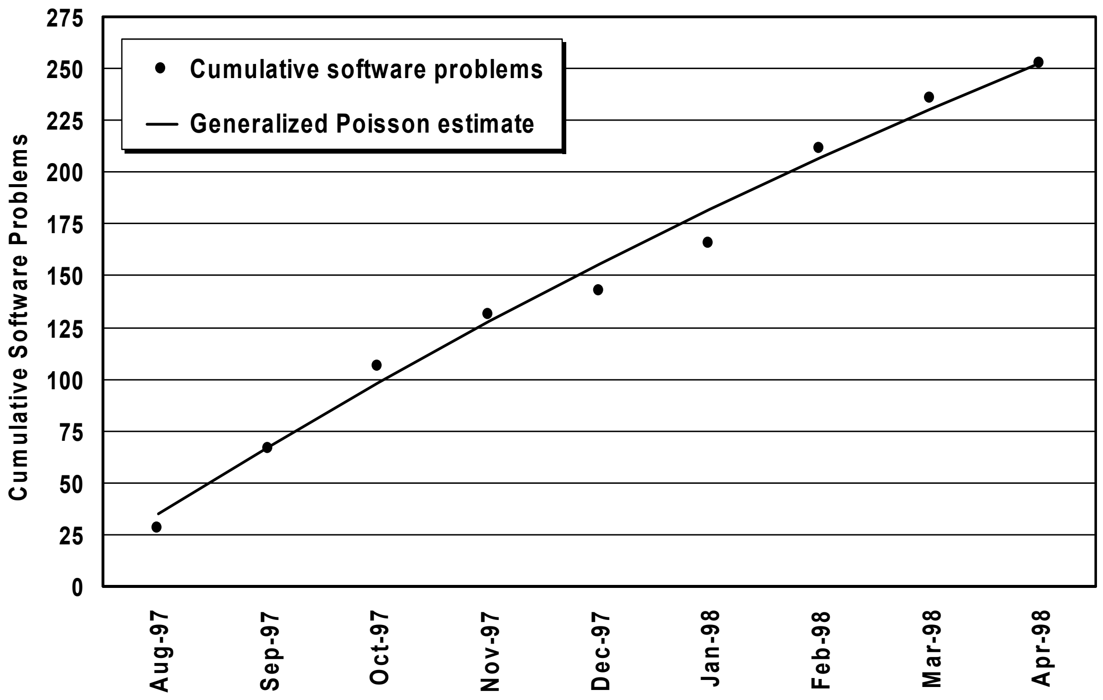


Figure F.5—CASRE results for DDS tactical software problems by month

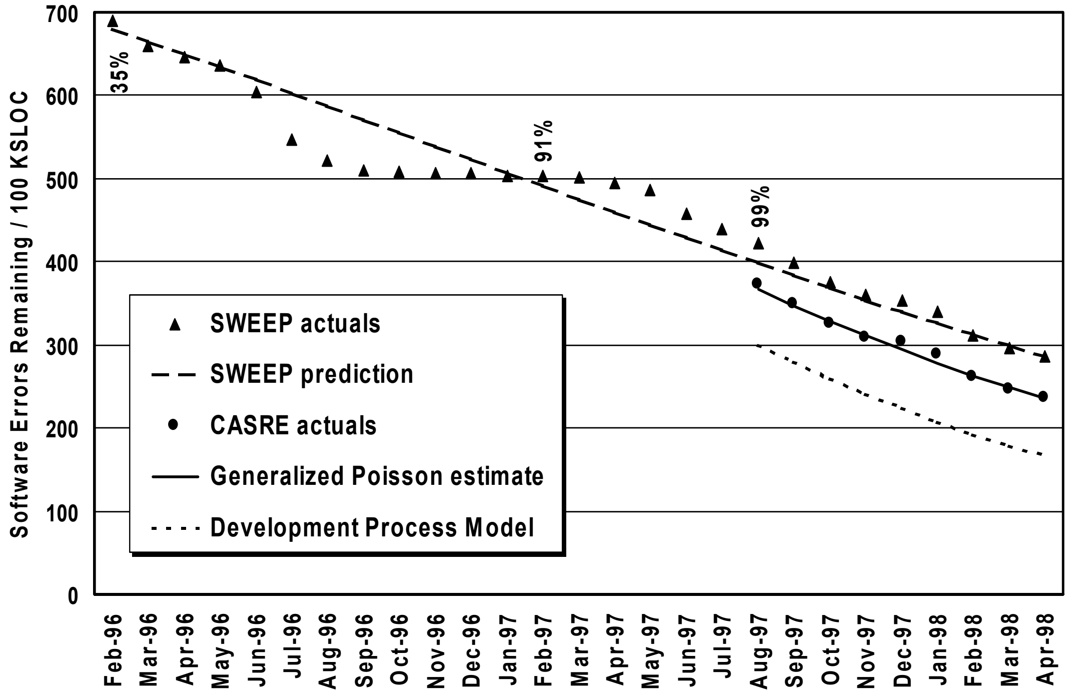
F.3.3 SWEEP predictions

The SWEEP model program, written by the Software Productivity Consortium, implements mathematical models to predict software fault rates. SWEEP uses software error data obtained from reviews, inspections, and testing throughout the development cycle to predict the succeeding number of errors that will be found later. It also estimates the latent fault content at delivery. SWEEP has the following three modes of operation: a time-based model, a phase-based model, and a planning aid. All three models use a two-parameter Rayleigh distribution to make predictions of the rate of discovery of the remaining defects in the software. SWEEP's time-based model was used for this analysis because software was being coded, tested, integrated, and retested simultaneously. After testing began, software error data was obtained from software problem reports in real time and grouped into months. As suggested in the SWEEP User Manual, the number of problems per month was normalized to errors per 100 KSLOC to account for the fact that the amount of software being tested was increasing.

F.3.4 Combined results

To easily compare and integrate all the model results, they are plotted on one graph. Since all three models could provide estimates of software errors remaining, it was decided to plot this on the vertical axis. For the CASRE curve, the errors remaining were calculated by subtracting the software problems from CASRE's estimate of the total number of errors. To match the SWEEP results, the CASRE results were also normalized to errors per 100 KSLOC. The development process model curve was easily plotted on the graph since it is based on errors per KSLOC. For the SWEEP curve, the errors remaining were calculated by subtracting the software problems from SWEEP's prediction of the total number of errors present. Figure F.6 shows the results of all three models. The percentages shown indicate the fraction of the total lines of code that were present during periods before September 1997. Note that the CASRE and SWEEP actuals differ only because the CASRE and SWEEP estimates of the total number of errors are different. The development process model curve would be even closer to the others if a few months of testing had been assumed before starting it. At least in this case, it seems clear that SR can be predicted well before system integration testing.

The graph in Figure F.6 shows that the reliability models are trending in a similar pattern. The DPPM model (illustrated as the "Development Process Model" in that graph, underestimates the remaining software error content by approximately 30%. This error content may vary between developing organizations and between projects. There is an opportunity for developing organizations to pay attention to their modeling results and compare these to actual field reliability results. Several models can then be normalized or refined for better prediction capability.



© 1999 IEEE. Reprinted, with permission, from the IEEE and Samuel Keene (author), for his paper presented at the *Tenth International Symposium on Software Reliability Engineering*.

Figure F.6—Combined results for DDS tactical software problems by month

F.4 Summary

The Keene development process model provides a ready model to estimate fault content and the resulting failure rate distribution at requirements planning time. It rewards better process capability of the developing organization with lower fault content and projected better field failure rates. This model requires the developer to know some things about his released code experience to fill in all the model parameters. It is now being popularly applied by several defense and aerospace contractors.

The Rayleigh SWEEP model is useful in projecting the number of defects to be found at each development stage. This helps in resource planning and in setting the expectation for the number of faults to be uncovered at each phase. It uses the defect data discovery profile to refine the initial Keene model prediction, for projected latent defects at delivery.

Both the Keene DPPM and the SWEEP Rayleigh model are useful additions with the CASRE test suite. They round out the reliability prediction tool kit and provide a continuing reliability focus throughout the development cycle. This continuing reliability focus throughout the development process will promote delivering and assuring a more reliable software product.

F.5 Supporting published material

[1] Bentz, R., and Smith, C., “Experience report for the Software Reliability Program on a military system acquisition and development,” *ISSRE ’96 Industrial Track Proceedings,* pp. 59–65.

[2] Chillarege, R., Biyani, S., and Rosenthal, J., “Measurement of failure rate in widely distributed software,” *25th Annual Symposium on Fault Tolerant Computing,* IEEE Computer Society, June 1995.

[3] Cole, G. F., and Keene, S., “Reliability growth of fielded software,” *ASQC Reliability Review,* vol. 14, pp. 5–23, Mar. 1994.

[4] Kan, S. H., *Metrics and Models in Software Quality Engineering,* Reading, MA: Addsion-Wesley Publishing, 1995, p. 192 (Rayleigh model discussion).

[5] Keene, S. J., “Modeling software R&M characteristics,” Parts I and II, *Reliability Review,* June and September 1997.

[6] Keene, S. J., “Modeling software R&M characteristics,” *ASQC Reliability Review,* Part I and II, vol. 17, no. 2 and no. 3, June 1997, pp.13–22.

[7] Peterson, J., “Modeling software reliability by applying the CASRE tool suite to a widely distributed, safety-critical system,” *11th Annual ISSRE 2000,* practical papers, San Jose, CA, Oct. 8–11, 2000.

[8] Peterson, J., Yin, M.-L., Keene, S., “Managing reliability development & growth in a widely distributed, safety-critical system,” *12th Annual ISSRE 2001,* practical papers, Hong Kong, China, Nov. 27–30, 2001.

[9] Reliability Analysis Center and Performance Technology, “New System Reliability Assessment Method,” IITRI Project Number A06830, pp. 53–68.

[10] Smith, C., *NCOSE Symposium Proceedings,* St. Petersburg, FL, June 23, 1998.

[11] Software Productivity Consortium, “Software Error Estimation Program User Manual,” Version 02.00.10, AD-A274697, Dec. 1993.

[12] Uber, C., and Smith, C., “Experience report on early software reliability prediction and estimation,” *Proceedings of the 10th International Symposium on Software Reliability Engineering,* practical papers, Boca Raton, FL, Nov. 1–4, pp 282–284.

Annex G

(informative)

Bibliography

[B1] Abdel-Ghaly, A. A., Chan, P. Y., and Littlewood, B., “Evaluation of competing software reliability predictions,” *IEEE Transactions on Software Engineering,* vol. SE-12, no. 9, pp. 950–967, 1986.

[B2] Boehm, B. W., *Software Engineering Economics,* New York: Prentice-Hall, 1981.

[B3] Bowen, J. B., “Application of a multi-model approach to estimating residual software faults and time between failures,” *Quality and Reliability Engineering International,* vol. 3, pp. 41–51, 1987.

[B4] Brocklehurst, S., and Littlewood, B., “New ways to get reliability measures,” *IEEE Software,* pp. 34–42, July 1992.

[B5] Brooks, W. D., and Motley, R.W., Analysis of Discrete Software Reliability Models*,* Technical Report #RADC-TR-80-84, Rome Air Development Center, 1980.

[B6] Capability Maturity Model® (CMM®) and Software Engineering Institute (SEI).[13]

[B7] Crow, L., Confidence Interval Procedures for Reliability Growth Analysis, Technical Report #197, U.S. Army Material Systems Analysis Activity, Aberdeen Proving Grounds, Maryland.

[B8] Dixon, W. J., and Massey, F. J., Jr., *Introduction to Statistical Analysis*, Third Edition. New York: McGraw-Hill, 1969, p.28.

[B9] Duane, J. T., “Learning curve approach to reliability monitoring,” *IEEE Transactions on Aerospace*, vol. 2, no. 2, pp. 563–566, Apr. 1964.

[B10] Ehrlich, W. K., Stampfel, J. P., and Wu, J. R., “Application of software reliability modeling to product quality and test process,” *Proceedings of the IEEE/TCSE Subcommittee on Software Reliability Engineering Kickoff Meeting,* NASA Headquarters, Washington, DC, Apr. 1990, paper 13.

[B11] Farr, W. H., A Survey of Software Reliability Modeling and Estimation, Technical Report #82-171, Naval Surface Warfare Center, Dahlgren, VA.

[B12] Farr, W. H., A Survey of Software Reliability Modeling and Estimating, Technical Report NSWC TR 82-171, Naval Surface Weapons Center, Sept. 1983, pp. 4–88.

[B13] Farr, W. H., and Smith, O. D., Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide, Technical Report NAVSWC TR-84-373, Revision 2, Naval Surface Warfare Center, Dahlgren, VA.

[B14] Fenton, N., and Littlewood, B., “Limits to Evaluation of Software Dependability,” *Software Reliability and Metrics,* London, U.K.: Elsevier Applied Science, pp. 81–110.

[B15] Freedman, R. S., and Shooman, M. L., An Expert System for Software Component Testing, Final Report, New York State Research and Development Grant Program, Contract No. SSF(87)-18, Polytechnic University, Oct. 1988.

[B16] Gifford, D., and Spector, A., “The TRW reservation system,” *Communications of the ACM*, vol. 2, no. 27, pp. 650–665, July 1984.

[B17] Goel, A., and Okumoto, K., “Time-dependent error-detection rate for software reliability and other performance measures,” *IEEE Transactions on Reliability*, vol. R-28, no. 3, pp. 206–211, 1979.

[13] SEI publications are available from the Software Engineering Institute of Carnegie Mellon University (http://www.sei.cmu.edu/).

[B18] Hecht, H., and Hecht. M., “Software reliability in the system context,” *IEEE Transactions on Software Engineering*, Jan. 1986.

[B19] Hecht, H., and Hecht, M., “Fault Tolerant Software,” in *Fault Tolerant Computing*, Pradham, D. K., ed. Prentice-Hall, 1986.

[B20] Hecht, H., “Talk on software reliability” AIAA Software Reliability Committee Meeting, Colorado Springs, CO, Aug. 22–25, 1989.

[B21] Hoel, P. G., *Introduction to Mathematical Statistics*, Fourth edition. New York: John Wiley and Sons, 1971.

[B22] IEEE 100™, *The Authoritative Dictionary of IEEE Standards Terms,* Seventh Edition, Institute of Electrical and Electronics Engineers, Inc.[14, 15]

[B23] IEEE Std 610.12™-1990, IEEE Standard Glossary of Software Engineering Terminology.

[B24] IEEE Std 982.1™-2005, IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.

[B25] IEEE Std 1044™-1993, IEEE Standard Classification for Software Anomalies.

[B26] IEEE Std 1061™-1998 (Reaff 2004), IEEE Standard for a Software Quality Metrics Methodology.

[B27] IEEE Std 1074™-2006, IEEE Standard for Developing a Software Project Life-Cycle Process.

[B28] IEEE Std 1413™-1998, IEEE Standard Methodology for Reliability Prediction and Assessment for Electronic Systems and Equipment.

[B29] IEEE Std 12207™-2008, Systems and Software Engineering—Software Life Cycle Processes.

[B30] Iyer, R. K., and Velardi, P., *A Statistical Study of Hardware Related Software Errors in MVS*, Stanford University Center for Reliable Computing, Oct. 1983.

[B31] Jelinski, Z., and Moranda, P., “Software Reliability Research,” in *Statistical Computer Performance Evaluation*, Freiberger, W., ed. New York: Academic Press, 1972, pp. 465–484.

[B32] Joe, H., and Reid, N., “Estimating the number of faults in a system,” *Journal of the American Statistical Association*, 80(389), pp. 222–226.

[B33] Kafura, D., and Yerneni, A., Reliability Modeling Using Complexity Metrics, Virginia Tech University Technical Report, Blacksburg, VA, 1987.

[B34] Kanoun, K., and Sabourin, T., “Software dependability of a telephone switching system,” *Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing (FTCS-17),* Pittsburgh, PA, 1987.

[B35] Karunantithi, N., Whitely, D., and Malaiya, Y. K., “Using neural networks in reliability prediction,” *IEEE Software,* pp. 53–60, July 1992.

[B36] Kline, M. B., “Software and hardware R&M: What are the differences?” *Proceedings of the Annual Reliability and Maintainability Symposium,* pp. 179–185, 1980.

[B37] Laprie, J. C., “Dependability evaluation of software systems in operation,” *IEEE Transactions on Software Engineering,* vol. SE-10, pp. 701–714, Nov. 84.

[B38] Kruger, G. A., “Validation and further application of software reliability growth models,” *Hewlett-Packard Journal*, vol. 8, no. 2, pp 13–25, Mar. 1991.

[14] IEEE publications are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).
[15] The IEEE standards or products referred to in Annex G are trademarks owned by the Institute of Electrical and Electronics Engineers, Incorporated.

[B39] Lipow, M., and Shooman, M. L., "Software reliability," in Consolidated Lecture Notes Tutorial Sessions Topics in Reliability & Maintainability & Statistics, *Annual Reliability and Maintainability Symposium,* 1986.

[B40] Littlewood, B., and Verrall, J. L., "A Bayesian reliability model with a stochastically monotone failure rate," *IEEE Transactions on Reliability,* vol. 23, pp. 108–114, June 1974.

[B41] Littlewood, B. "Software reliability model for modular program structure," *IEEE Transactions on Reliability,* vol. R-28, pp. 241–246, Aug. 1979.

[B42] Littlewood, B., Ghaly, A., and Chan, P. Y., "Tools for the Analysis of the Accuracy of Software Reliability Predictions," Skwirzynski, J. K., ed. *Software System Design Methods,* NATO ASI Series, F22, Heidelberg, Germany: Springer-Verlag, 1986, pp. 299–335.

[B43] Lloyd, D. K., and Lipow, M., *Reliability: Management, Methods, and Mathematics,* Second edition. ASQC, 1977.

[B44] Lu, M., Brocklehurst, S., and Littlewood, B., "Combination of predictions obtained from different software reliability growth models," *Proceedings of the Tenth Annual Software Reliability Symposium,* Denver, CO, pp. 24–33, June 1992.

[B45] Lyu, M. R., and Nikora, A., "Applying reliability models more effectively," *IEEE Software,* pp. 43–52, July 1992.

[B46] Lyu, M. R. (Editor-in-Chief), *Handbook of Software Reliability Engineering,* Los Alamitos, CA: Computer Society Press, and New York: McGraw-Hill, 1995.

[B47] Mathur, A. P., and Horgan J. R., "Experience in using three testing tools for research and education in software engineering," *Proceedings of the Symposium on Assessment of Quality Software Development Tools,* New Orleans, LA, pp. 128–143, May 27–29, 1992.

[B48] McCall, J. A., et al., 'Methodology for Software and System Reliability Prediction," Final Technical Report RADC-TR-87-171, Prepared for RADC, Science Applications International Corporation, 1987.

[B49] Mellor, P., "State of the Art Report on Software Reliability,." *Infotech,* London, 1986.

[B50] MIL-HDBK-217E, Military Handbook Reliability Prediction of Electronic Equipment, Rome Air Development Center, Griffis AFB, NY, Oct. 27, 1986.[16]

[B51] Munson, J. C., and Khoshgoftaar, T. M., "The use of software complexity metrics in software reliability modeling," *Proceedings of the International Symposium on Software Reliability Engineering,* Austin, TX, pp. 2–11, May 1991.

[B52] Musa, J., "A theory of software reliability and its application," *IEEE Transactions on Software Engineering,* vol. SE-1, no. 3, pp. 312–327, Sept. 1975.

[B53] Musa, J. D., and Okumoto, K., "A logarithmic Poisson execution time model for software reliability measurement," *Proceedings of the Seventh International Conference on Software Engineering*, Orlando, FL, pp. 230–238, Mar. 1984.

[B54] Musa, J. D., Iannino, A., and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill, 1987.

[B55] Musa, J. D., *Software Reliability Engineering: More Reliable Software Faster and Cheaper,* Second edition. AuthorHouse, 2004.

[B56] Nonelectronic Parts Reliability Data, NPRD-3, Reliability Analysis Center, Rome Air Development Center, Griffis AFB, NY, 1985, NTIS ADA 163514.

[B57] Rook, P., *Software Reliability Handbook*, London, U.K.: Elsevier Applied Science, 1990.

[16] MIL publications are available from Customer Service, Defense Printing Service, 700 Robbins Ave., Bldg. 4D, Philadelphia, PA 19111-5094.

[B58] Schneidewind, N. F., “Analysis of error processes in computer science,” *Proceedings of the International Conference on Reliable Software*, IEEE Computer Society, pp. 337–346, Apr. 21–23, 1975.

[B59] Schneidewind, N. F., and Keller, T. M., “Applying reliability models to the Space Shuttle,” *IEEE Software*, vol. 9, no. 4, pp. 28–33, July 1992.

[B60] Schneidewind, N. F., “Software reliability model with optimal selection of failure data,” *IEEE Transactions on Software Engineering,* vol. 19, no. 11, pp. 1095–1104, Nov. 1993.

[B61] Schneidewind, N. F., “Software reliability engineering for client-server systems,” *Proceedings of the 7th International Symposium on Software Reliability Engineering,* White Plains, NY, pp. 226–235, 1996.

[B62] Schneidewind, N. F., “Reliability modeling for safety critical software,” *IEEE Transactions on Reliability,* vol. 46, no. 1, pp. 88–98, Mar. 1997.

[B63] Schneidewind, N. F., “Reliability and maintainability of requirements changes,” *Proceedings of the International Conference on Software Maintenance,* Florence, Italy, pp. 127–136, Nov. 7–9, 2001.

[B64] Schneidewind, N. F., “Modeling the fault correction process,” *Proceedings of the Twelfth International Symposium on Software Reliability Engineering,* Hong Kong, pp. 185–190, 2001.

[B65] Schneidewind, N. F., “Report on results of discriminant analysis experiment,” *27th Annual NASA/IEEE Software Engineering Workshop,* Greenbelt, MD, Dec. 5, 2002.

[B66] Schneidewind, N. F., “Predicting risk with risk factors,” *29th Annual IEEE/NASA Software Engineering Workshop (SEW 2005),* Greenbelt, MD, Apr. 6–7, 2005.

[B67] Shooman, M. L., “Probabilistic Models for Software Reliability Prediction,” in *Statistical Computer Performance Evaluation,* Freiberger, W., ed. New York: Academic Press, 1972, pp. 485–502.

[B68] Shooman, M. L., “Structural models for software reliability prediction,” *Second National Conference on Software Reliability*, San Francisco, CA, Oct. 1976.

[B69] Shooman, M. L., *Software Engineering: Design, Reliability, and Management*, New York: McGraw-Hill Book Co., 1983.

[B70] Shooman, M. L., and Richeson, G., “Reliability of Shuttle Control Center software,” *Proceedings Annual Reliability and Maintainability Symposium,* pp. 125–135, 1983.

[B71] Shooman, M. L., *Probabilistic Reliability: An Engineering Approach*, New York: McGraw-Hill Book Co., 1968 (Second edition, Melbourne, FL: Krieger, 1990).

[B72] Shooman, M. L., “Early software reliability predictions,” Software Reliability Newsletter, Technical Issues Contributions, IEEE Computer Society Committee on Software Engineering, Software Reliability Subcommittee, 1990.

[B73] Siefert, D. M., “Implementing software reliability measures,” *The NCR Journal*, vol. 3, no. 1, pp. 24–34, 1989.

[B74] Stark, G. E., “Software reliability measurement for flight crew training simulators,” *AIAA Journal of Aircraft*, vol. 29, no. 3, pp. 355–359, 1992.

[B75] Takahashi, N., and Kamayachi, Y., “An empirical study of a model for program error prediction,” *Proceedings of the 8th International Conference on Software Engineering*, London, U.K., pp. 330–336.

[B76] Yamada, S., Ohba, M., and Osaki, S., “S-shaped reliability growth modeling for software error detection,” *IEEE Transactions on Reliability*, vol. R-32, no. 5, pp. 475–478, Dec. 1983.