EGG—2663
Distribution Category: UC-705

Software Reliability Models for Critical Applications

Hoang Pham
Michelle Pham

EGG--2663

DE92 004797

Published December 1991

Idaho National Engineering Laboratory
EG&G Idaho, Inc.
Idaho Falls, Idaho 83415

Prepared for the
U.S. Department of Energy
Under DOE Field Office, Idaho
Contract No. DE-AC07-76ID01570

ABSTRACT

This report presents the results of the first phase of the ongoing EG&G Idaho, Inc., Software Reliability Research Program. The program is studying the existing software reliability models and proposes a state-of-the-art software reliability model that is relevant to the nuclear reactor control environment.

This report consists of three parts: (1) summaries of the literature review of existing software reliability and fault tolerant software reliability models and their related issues, (2) proposed technique for software reliability enhancement, and (3) general discussion and future research.

The development of this proposed state-of-the-art software reliability model will be performed in the second phase.

CONTENTS

TABLES

FIGURES

ACKNOWLEDGMENTS

The authors would like to express their appreciation to Professor Shambhu Upadhyaya at State University of New York at Buffalo for his remarkable contribution to the development of this work. We also thank Philip E. MacDonald, Group Manager, Steve A. Eide, Unit Manager, and the SSRE Center of Excellence for supporting this program.

1. INTRODUCTION

Since the invention of computers, computer software has gradually become an important part of many systems. In the 1970s, the cost of software surpassed the cost of hardware as being the major cost of a system [333]. In addition to the cost of developing software, the penalty costs of software failures are even more significant. As missions accomplished by human beings become more and more complex, for example, the air traffic control system, nuclear power plant control systems, the space program, and military systems, the failure of software may involve very high costs, human lives, and a social impact. Therefore, how to measure and predict the reliability of software and techniques to enhance reliability have become problems of common interest in the reliability literature.

In the past 20 years, more than 300 research papers have been published in the areas of software reliability characteristics, software reliability modeling, software reliability model validation, and software fault tolerance. Since software is an interdisciplinary science, software reliability models are also developed from different perspectives toward software and with different applications of the model. In order to pave the way for the future development and evaluation of highly reliable software and systems involving software and hardware, a detailed taxonomy of the existing software reliability models, software fault tolerant models, and the assumptions behind those models is deemed important.

The purpose of this research is to collect the literature on existing software reliability models and fault tolerance techniques to propose new reliability enhancement methods and a fault tolerant technique relevant for nuclear reactor control systems. Continuing research will be performed at EG&G Idaho, Inc. to develop a state-of-the-art software reliability model in the second phase of this ongoing program.

Section 2 presents an overview of the software reliability models and fault tolerant software techniques. Section 3 presents reliability enhancement techniques and proposes a simple but effective fault tolerant technique applicable to the nuclear reactor environment. Section 4 is a general discussion and summary of proposed research to be conducted in the next phase.

2. LITERATURE REVIEW

This section presents a review of literature that deals with the issues of software reliability and fault tolerance. The general characteristics of existing software reliability models are presented first (section 2.1). Then these models are classified into various types and groups, and their characteristics discussed (section 2.2). Specific models in each group are presented in further detail in section 2.3. Section 2.4 reviews the existing fault tolerant software reliability models and discusses to some extent these models.

2.1 Characteristics of Software Reliability Models

In hardware reliability, the mechanism of failure occurrence is often treated as a black box. It is the failure process that is of interest to reliability engineers. The emphasis is on the analysis of failure data. In software reliability, one is interested in the failure mechanism. Most software reliability models are analytical models derived from assumptions of how failures occur. The emphasis is on the model's assumptions and the interpretation of parameters.

In order to develop a useful software reliability model and to make sound judgments when using the models, an in-depth understanding is needed of how software is produced; how errors are introduced, how software is tested, how errors occur, types of errors, and environmental factors can help us in justifying the reasonableness of the assumptions, the usefulness of the model, and the applicability of the model under a given user environment.

General descriptions of software and software reliability, software life cycle, the bug-counting concept, software reliability versus hardware reliability, time index, and error analysis are discussed below.

2.1.1 General Description of Software and Software Reliability

Similar to hardware reliability, time-domain software reliability is defined as the probability of failure-free operation of software for a specified period of time under specified conditions [333]. Software is a collection of instructions or statements of computer languages. It is also called a computer program or simply a program. Upon execution of a program, an input state is translated into an output state. Hence, a program can be regarded as a function, f, mapping the input space to the output space (f: input --> output), where the input space is the set of all input states, and the output space is the set of all output states. An input state can be defined as a combination of input variables or a typical transaction to the program.

A software program is designed to perform specified functions. When the actual output deviates from the expected output, a failure occurs. It is worth noting that the definition of failure differs from application to application and should be clearly defined in specifications. For instance, a response time of 30 seconds could be a serious failure for an air traffic control system, but acceptable for an airline reservation system. A fault is an incorrect logic, incorrect instruction, or inadequate instruction that, by execution, will cause a failure. In other words, faults are the sources of failures, and failures are the realization of faults. When a failure occurs, there must be a corresponding fault in the program, but the existence of faults may not cause the program to fail and a program will never fail as long as the faulty statements are not executed.

2.1.2 Software Life Cycle

A software life cycle is normally divided into a requirement and specification phase, design phase, coding phase, testing phase, and operational and maintenance phase. The design phase may include a preliminary design and a detailed design. The testing phase may include module testing, integration testing, and field testing. The maintenance phase may include one or more subcycles, each having all the phases in the development stage.

In the early phase of the software life cycle, a predictive model is needed because no failure data are available. This type of model predicts the number of errors in the program before testing. In the testing phase, the reliability of the software improves through debugging. A reliability growth model is needed to estimate the current reliability level and the time and resources required to achieve the objective reliability level. During this phase, reliability estimation is based on the analysis of failure data. After the release of a software program, the addition of new modules, removal of old modules, removal of detected errors, mixture of new code with previously written code, change of user environment, change of hardware, and management involvement have to be considered in the evaluation of software reliability. During this phase, an evolution model is needed.

2.1.3 The Bug-counting Concept

The bug-counting model assumes that, conceptually, there is a finite number of faults in the program. Given that faults can be counted as an integer number, bug-counting models estimate the number of initial faults at the beginning of the debugging phase and the number of remaining faults during, or at the end of, the debugging phase. Bug-counting models use a per-fault failure rate as the basic unit of failure occurrence. Depending upon the type of models, the failure rate of each fault is either assumed to be a constant, a function of debugging time, or a random variable from a distribution. Once the per-fault failure rate is determined, the program failure rate is computed by multiplying the number of faults remaining in the program by the failure rate of each fault.

During the debugging phase, the number of remaining faults changes. One way of modeling this failure process is to represent the number of remaining faults as a stochastic counting process. Similarly, the number of failures experienced can also be denoted as a stochastic counting process. By assuming perfect debugging, i.e., a fault is removed with certainty whenever a failure occurs, the number of remaining faults is a nonincreasing function of debugging time. With an imperfect debugging assumption, i.e., faults may not be removed, introduced, or changed at each debugging, the number of remaining faults may increase or decrease. This bug-counting process can be represented by a binomial model, Poisson model, compound Poisson process, or Markov process.

2.1.4 Software Reliability versus Hardware Reliability

Since the emergence of the study of software reliability, reliability theoreticians and practitioners have discussed the issues of software reliability versus hardware reliability in terms of similarity, differences, modeling techniques, etc. [131, 340]. Because the basic modeling techniques of software reliability are adapted from reliability theory developed for hardware systems in the past 30 years, a comparison of software reliability and hardware reliability can help in the use of these theories and in the study of hardware-software systems. Table 1 lists the differences and similarities between the two.

2.1.5 Time Index

In hardware, materials deteriorate over time. Hence, calendar time is a widely accepted index for reliability function. In software, failures will never happen if the program is not used. In the context of software reliability, *time* is more appropriately interpreted as the *stress* placed on, or *amount of work* performed by, the software. The following time units are generally accepted as indices of the software reliability function:

Execution time	-	CPU time; time during which the CPU is busy
Operation time	-	Time the software is in use
Calendar time	-	Index used for software running 24 hours a day
Run	-	A job submitted to the CPU
Instruction	-	Number of instructions executed
Path	-	The execution sequence of an input.

Models based on execution time, operational time, calendar time, and instructions executed belong to the time-domain model. Models based on run and path belong to the input-domain model.

Table 1. Software Reliability versus Hardware Reliability

Software Reliability	Hardware Reliability
Without considering program evolution, failure rate is statistically nonincreasing.	Failure rate has a bathtub curve. The burn-in state is similar to the software debugging state.
Failures never occur if the software is not used.	Material deterioration can cause failures even though the system is not used.
Failure mechanism is studied.	Failure mechanism may be treated as a black box.
CPU time and run are two popular indices for the reliability function.	Calendar time is a generally accepted index for the reliability function.
Most models are analytically derived from assumptions. Emphasis is on developing the model, the interpretation of the model assumptions, and the physical meaning of the parameters.	Failure data are fitted to some distributions. The selection of the underlying distribution is based on the analysis of failure data and experiences. Emphasis is placed on analyzing failure data.
Failures are caused by incorrect logic, incorrect statements, or incorrect input data. This is similar to the design errors of a complex hardware system.	Failures are caused by material deterioration, random failures, design errors, misuse, and environment.

2.1.6 Error Analysis

Error analysis, including the analysis of failures and the analysis of faults, plays an important role in the area of software reliability, for several reasons. First, failure data must be identified, collected, and analyzed before they can be plugged into any software reliability model. In doing so, an unambiguous definition of failures must be agreed upon. Although not critical to theoreticians, it is extremely important in practice. Second, the analysis of error sources and error removal techniques provides information in the selection of testing strategies and the development of new methodologies of software reliability modeling. Errors are classified by (a) severity, (b) special errors, (c) error type, (d) error introduced in the software life cycle, (e) error removed in the software life cycle, and (f) techniques of error removal.

Severity

In practice, it is often necessary to classify failures by their impact on the organization. As pointed out by Musa et al. [248], cost impact, human life impact, and service impact are common criteria. Each criterion can be further divided by the degree of severity. For example, minor error, incorrect result, partial operation, or system breakdown can be a criterion for service impact.

To estimate the failure rate of each severity level, Musa et al. [248] suggest the following approaches:

- Classify the failures and estimate failure rates separately for each class.
- Classify the failures, but lump the data together, weighing the time intervals between failures of different classes according to the severity of the failure class.
- Classify the failures, but ignore severity in estimating the overall failure rate. Develop failure rates for each failure class by multiplying the overall failure rate by the proportion of failures occurring in each class.

In addition to the estimation of failure rate of each severity class, the penalty costs of failure can be measured in dollar value [107].

Special Errors

Transient error, internal error, hardware caused software error, previously fixed error, and generated error are some special errors of interest to software reliability engineers. Transient errors are errors that exist within too short a time to be isolated [333]. This type of error may happen repeatedly. In failure data collection, transient errors of the same type should be counted only once. Internal errors are intermediate errors whose consequences are not observed in the final output [173], which happens when an internal error has not propagated to a point where the output is influenced. For instance, in fault-tolerant computing some errors may be guarded against by redundant codes and not observed in the final output. When setting up the reliability objective, decisions must be made to either count the internal errors or simply count the observable errors.

Hardware caused software errors are errors that, if not carefully investigated, may be regarded as common software errors [146]. For example, a program may terminate during execution and indicate it by an error message of operating system error. Without careful investigation, this error may be classified as software error when the operating system error was actually caused by the hardware.

Error Type

By analyzing the failure data or trouble reports, errors can be classified by their properties. One of the classification schemes given by Thayer et al. [363] includes the following error types:

- Computational errors
- Logical errors
- Input/output errors
- Data handling errors
- Operating system/system support errors
- Configuration errors
- User interface errors
- Data base interface errors
- Present data base errors
- Global variable definition errors
- Operator errors
- Unidentified errors.

As failure data are collected, the frequency of each type can be obtained. Other classification schemes can be seen in [98, 112].

Error Introduced in the Software Life Cycle

Within the software life cycle, errors can be introduced in the following phases [44, 173, 363]:

- Requirement and specification
- Design
  - Function design
  - Logical design
- Coding
- Documentation
- Maintenance.

For each phase, the frequency of occurrence can be obtained from failure data [44].

Error Removed in the Software Life Cycle

Errors are removed through testing, which can be divided into the following stages [363]:

- Validation
- Integration testing
- Acceptance testing
- Operation and demonstration.

The frequency of occurrence of each category is also of interest to software reliability engineers.

Techniques of error removal

Some techniques of error removal given in [153, 260, 363] are summarized below. This type of study gives us information on the selection and validation of software design and testing techniques.

- Automated requirement aids
- Simulation
- Design language
- Design standard
- Logic specification review
- Module logic inspection
- Module code inspection
- Unit test
- Subsystem test
- System test

2.2 Classification of Software Reliability Models

Software reliability models can be classified into two types of models: the deterministic and the probabilistic.

2.2.1 Deterministic Models

The deterministic type is used to study (1) the elements of a program by counting the number of operators, operands, and instructions, (2) the control flow of a program by counting the branches and tracing the execution paths, and (3) the data flow of a program by studying the data sharing and data passing.

Performance measures of the deterministic type are obtained by analyzing the program texture and do not involve any random event. There are two models in the deterministic type: Halstead's software science model and McCabe's cyclomatic complexity model. In general, these models represent a growing quantitative approach to the measurement of computer software. Halstead's software science model is used to estimate the number of errors in the program, whereas McCabe's cyclomatic complexity model [228] is used to determine an upper bound on the number of tests in a program.

2.2.2 Probabilistic Models

The probabilistic type represents the failure occurrences and the fault removals as probabilistic events. It can also be further divided into different groups of models: (a) error seeding, (b) failure rate, (c) curve fitting, (d) reliability growth, (e) program structure, (f) input domain, (g) execution path, (h) nonhomogeneous Poisson process, (i) Markov, and (j) Bayesian and unified.

Error Seeding

The error seeding group of models estimates the number of errors in a program by using the capture-recapture sampling technique. Errors are divided into indigenous errors and induced errors (seeded errors). The unknown number of indigenous errors is estimated from the number of induced errors and the ratio of the two types of errors obtained from the debugging data.

Failure Rate

The failure rate group of models is used to study the functional forms of the per-fault failure rate and program failure rate at the failure intervals. Since mean-time-between-failure is the reciprocal of failure rate in the exponential distribution random variable, models based on time-between-failure also belong to this category.

Models included in this group are the

- Jelinski and Moranda De-Eutrophication [150]
- Schick and Wolverton [306]
- Jelinski-Moranda geometric De-Eutrophication [242]
- Moranda geometric Poisson [213]
- Modified Schick and Wolverton [354]
- Goel and Okumoto imperfect debugging [117].

Curve Fitting

The curve fitting group of models uses regression analysis to study the relationship between software complexity and the number of errors in a program, the number of changes, failure rate, or time-between-failure. Both parametric and nonparametric methods have been attempted in this field.

Models included in this group are the

- Estimation of errors
- Estimation of change
- Estimation of time between failures
- Estimation of failure rate.

Reliability Growth

The reliability growth group of models measures and predicts the improvement of reliability through the debugging process. A growth function is used to represent the progress. The independent variables of the growth function can be time, number of test cases, or testing stages, and the dependent variables can be reliability, failure rate, or cumulative number of errors detected.

Models included in this group are the

- Duane growth [84]
- Weibull growth [381]
- Wagoner's Weibull [380]
- Logistic growth curve [398]
- Gompertz growth curve [261]
- Hyperbolic reliability growth.

Program Structure

The program structure group of models views a program as a reliability network. A node represents a module or a subroutine, and the directed arc represents the program execution sequence among modules. By estimating the reliability of each node, the reliability of transition between nodes, and the transition probability of the network, and assuming independence of failure at each node, the reliability of the program can be solved as a reliability network problem.

Models included in this group are the

- Littlewood Markov structure [213]
- Cheung's user-oriented Markov [69].

Input Domain

The input-domain group of models uses *run* (the execution of an input state) as the index of reliability function as opposed to *time* in the time domain model. The reliability is defined as the number of successful runs over the total number of runs. Emphasis is placed on the probability distribution of input states or the operational profile.

Models included in this group are the

- Basic input-domain
- Input-domain based stochastic.

Execution Path

The execution path group of models estimates software reliability based on the probability of executing a logic path of the program and the probability of an incorrect path. This model is similar to the input domain model because each input state corresponds to an execution path.

The model forming this group is the

- Shooman decomposition [328].

Nonhomogeneous Poisson Process

The nonhomogeneous Poisson process (NHPP) group of models provides an analytical framework for describing the software failure phenomenon during testing. The main issue in the NHPP model is to estimate the mean value function of the cummulative number of failures experienced up to a certain time point.

Models included in this group are the

- Musa exponential [259]
- Goel and Okumoto NHPP [119]
- S-shaped growth [268,391]
- Hyperexponential growth [139, 268].

Markov

The Markov group of models is a general way of representing the software failure process. The number of remaining faults is modeled as a stochastic counting process. When a continuous-time discrete-state Markov chain is adapted, the state of the process is the number of remaining faults, and time-between-failures is the sojourning time from one state to another. If we assume that the failure rate of the program is proportional to the number of remaining faults, linear death process and linear birth-and-death process are two models readily available. The former assumes that the remaining error is monotonically nonincreasing, while the latter allows faults to be introduced during debugging.

When a nonstationary Markov model is considered, the model becomes very rich and unifies many of the proposed models. The nonstationary failure rate property can also simulate the assumption of nonidentical failure rates of each fault.

Models included in this group are the

- Linear death with perfect debugging
- Linear death with imperfect debugging
- Nonstationary linear death with perfect debugging

- Nonstationary linear birth-and-death.

Bayesian and Unified

The Bayesian and Unified groups of models assume a prior distribution of the failure rate. These models are used when the software reliability engineer has a good feeling about the failure process, and the failure data are rare. Besides the continuous time discrete state Markov chain, the exponential order statistics [215, 216, 269] and the shock model [160, 176, 251] are two other general models.

2.3 Characteristics of Specific Software Reliability Models

2.3.1 Halstead's Software Science Model

Halstead's theory of software science is probably the best known technique to measure the complexity in a software program. Halstead [123] uses the number of distinct operators and the number of distinct operands in a program to develop expressions for the overall program length, volume, level (a measure of complexity), development time, development effort, and the number of errors in a program [106].

The following notation is used in the discussion:

n_1	=	the number of distinct operators that appear in a program
n_2	=	the number of distinct operands that appear in a program
N_1	=	the total number of operator occurrences
N_2	=	the total number of operand occurrences
N	=	length of the program
V	=	volume of the program
B	=	number of errors in the program
$\hat{B}$	=	estimate of B
I	=	number of machine instructions.

Halstead shows that the length of the program, N, can be estimated:

$$N = N_1 + N_2$$

where

$$N_1 = n_1 \log_2 n_1$$

$$N_2 = n_2 \log_2 n_2$$

and

$$V = N \log_2(n_1 + n_2) \ .$$

Halstead also derived a formula to estimate the number of errors in the program, B, from program volume. The fonnula is

$$\hat{B} = \frac{V}{3000}.$$

Fitzsimmons [101], provides the following example: For the SORT module program [101] shown in Figure 2.1, the volume for this program can be calculated as follows:

$$\begin{aligned} N &= N_1 + N_2 = 50 \\ V &= N \log_2(n_1 + n_2) \\ &= 50 \log_2(10 + 7) \\ &= 204. \end{aligned}$$

2.3.2 McCabe's Software Complexity Model

A cyclomatic complexity measure of software proposed by McCabe [228] is based on a control flow representation of a program. Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

The cyclomatic number V(G) of a graph G with n nodes, e edges, and p connected components is

In a strongly connected graph (there is a path joining any pair of nodes), the cyclomatic number is equal to the maximum number of linearly independent circuits. The linearly independent circuits form a basis for the set of all circuits in G, and any path through G can be expressed as a linear combination of them.

When a program is represented as a flowgraph with a unique entry node and a unique exit node, this flowgraph becomes a strongly connected graph if a dummy edge from the exit

Figure 2.1 Operators and operands for an Interchange Sort program

Interchange sort program

```
   SUBROUTINE SORT (X,N)
   DIMENSION X(N)
   IF (N.LT.2) RETURN
   DO 20 I = 2,N
      DO 10 J = 1,I
      IF (X(I).GE.X(J)) GO TO 10
         SAVE = X(I)
         X(I) = X(J)
         X(J) = SAVE
10 CONTINUE
20 CONTINUE
   RETURN
   END
```

Operators of the Interchange sort program

Operator	Count
1 End of statement	7
2 Array subscript	6
3 =	5
4 IF ( )	2
5 DO	2
6 ,	2
7 End of program	1
8 .LT.	1
9 .GE.	1
n_1 = 10 GO TO 10	1
	28 = N_1

Operands of the Interchange sort program

Operand	Count
1 X	6
2 I	5
3 J	4
4 N	2
5 2	2
6 SAVE	2
n_2 = 7 1	1
	22 = N_2

node to the entry node is added. When the number of connected components is greater than 1, i.e., a main program and some subroutines, the above formula is modified to

$$V(G) = e - n + 2p \ .$$

The cyclomatic number of a graph with multiple connected components is equal to the sum of the cyclomatic numbers of the connected components. Another simple way of computing the cyclomatic number is the following.

$$V(G) = \pi + 1$$

where π is the number of predicate nodes (decisions or branches) in the program. In other words, the cyclomatic number is a measure of the number of branches in a program. A branch occurs in IF, WHILE, REPEAT, and CASE statements (a GO TO statement is normally excluded from the structured program). The cyclomatic number has been widely used in predicting the number of errors and in measuring the software quality.

McCabe [228] notes that when used in the context of the basis path testing method, the cyclomatic complexity, V(G), provides an upper bound for the number of independent paths in the basis set of a program and an upper bound on the number of tests that must be conducted to ensure that all program statements have been executed at least once.

2.3.3 Models in The Error Seeding Group

Mills [236] proposed an error seeding method to estimate the number of errors in a program by introducing pseudo errors into the program. From the debugging data, which consist of indigenous errors and induced errors, the unknown number of indigenous errors can be estimated. This model can be represented by a hypergeometric distribution.

The probability of k induced errors in r removed errors follows a hypergeometric distribution.

$$P(k;N+n_1,n_1,r) = \frac{\binom{n_1}{k}\binom{N}{r-k}}{\binom{N+n_1}{r}}$$

where

N = number of indigenous errors

n_1 = number of induced errors

r = number of errors removed during debugging

k = number of induced errors in r removed errors

r-k = number of indigenous errors in r removed errors.

Since n_1, r, and k are known, the maximum likelihood estimate (MLE) of N can be shown to be [236]

$$\hat{N} = \frac{n_1(r-k)}{k}.$$

This method was criticized for its inability to determine the type, location, and difficulty level of the induced errors such that they would likely be detected equally likely as the indigenous errors. Basin [32] suggests a two-step procedure with which one programmer detects n_1 errors and a second programmer independently detects r errors from the same program. With this method, the n_1 errors detected by the first programmer resembles the induced errors in the Mill's model. Let k be the common errors found by two programmers. The hypergeometric model becomes [32]

$$P(k;N,N-n_1,r) = \frac{\binom{n_1}{k}\binom{N-n_1}{r-k}}{\binom{N}{r}}$$

and the MLE of N is

$$\hat{N} = \frac{n_1 r}{k}.$$

Since no errors are actually introduced into the program, the difficulties in Mill's method are overcome.

Lipow [200] modified Mill's model by introducing an imperfect debugging probability q. The probability of removing k induced errors and r-k indigenous errors in m tests is a combination of binomial and hypergeometric distributions.

$$P(k; N+n_1, n_1, r, m) = \binom{m}{r} (1-q)^r q^{m-r} \frac{\binom{n_1}{k}\binom{N}{r-k}}{\binom{N+n_1}{r}}$$

$$N \geq r-k \geq 0, \; n_1 \geq k \geq 0, \; and \; m \geq r.$$

The interval estimate of N can be found in Huang [139].

2.3.4 Models in The Failure Rate Group

Based on the concept of bug-counting, the number of faults in the program increases or decreases by an integer number (normally assumed to decrease by 1) at each debugging. As the number of remaining faults changes, the failure rate of the program changes accordingly. Since the number of faults in the program is a discrete function, the failure rate of the program is also a discrete function with discontinuities at the failure times. Failure rate models study (1) how failure rate changes at the failure time and (2) the functional form of the failure rate during the failure intervals.

Jelinski and Moranda De-Eutrophication Model

The Jelinski and Moranda De-Eutrophication model [150] is one of the earliest software reliability models. It is the most often cited model. Many probabilistic software reliability models are either variants or extensions of this basic model.

Assumptions include the following:

- The program contains N initial faults
- Each error in the program is independent and equally likely to cause a failure during test
- Time intervals between occurrences of failure are independent of each other
- Whenever a failure occurs, a corresponding fault is removed with certainty
- The error removal time is negligible and no new errors are introduced during the error removal process
- The software failure rate during a failure interval is constant and is proportional to the number of faults remaining in the program.

The program failure rate at the ith failure interval is

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

where

ϕ	=	a proportional constant
N	=	the initial number of errors in the program
t_i	=	the time between the (i-1)st and the ith failures.

The probability density function (pdf) and cumulative distribution function (cdf) of t_i are

$$f(t_i) = \phi(N-(i-1))\exp(-\phi(N-(i-1))t_i)$$

and

$$F(t_i) = 1-\exp(-\phi(N-(i-1))t_i)$$

respectively.

The software reliability function is

$$R(t_i) = \exp(-\phi(N-(i-1))t_i)$$

Schick and Wolverton Model

The Schick and Wolverton (S-W) model [306] is similar to the J-M model, except it further assumes that the failure rate at the ith time interval increases with time since the last debugging. In the model, the program failure rate function between the (i-1)st and the ith failure can be expressed as

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

where ϕ is a proportional constant, N is the initial number of errors in the program, and t_i is the test time since the (i-1)st failure.

Jelinski-Moranda Geometric De-Eutrophication Model

The J-M geometric De-Eutrophication model [242] assumes that the program failure rate function is initially a constant D and decreases geometrically at failure times. The program failure rate and reliability function of time-between-failures at the ith failure interval can be expressed as

$$\lambda(t_i) = Dk^{i-1}$$

and

$$R(t_i) = \exp(-Dk^{i-1}t_i)$$

where

D	=	initial program failure rate
k	=	parameter of geometric function ($0 < k < 1$).

A modified version of the J-M geometric model was suggested by Lipow [201] to allow multiple error removal in a time interval. The program failure rate becomes

$$\lambda(t_i) = D\,k^{n_{i-1}}$$

where n_{i-1} is the cumulative number of errors found up to the (i-1)st time interval.

Moranda Geometric Poisson Model

The Moranda geometric Poisson model [213] assumes that at fixed times T, 2T, ... of equal length interval, and that the number of failures occurring at interval i, n_i, follow a Poisson distribution with intensity rate $D\,k^{i-1}$. The probability of getting m failures at the ith interval is

$$Pr\{n_i = m\} = \frac{e^{-D\,k^{i-1}}(D\,k^{i-1})^m}{m!} \quad .$$

Modified Schick and Wolverton Model

Sukert [354] modifies the S-W model to allow more than one failure at each time interval. The program failure rate becomes

$$\lambda(t_i) = \phi[N - n_{i-1}]\,t_i$$

where n_{i-1} is the cumulative number of failures at the (i-1)th failure interval.

Goel and Okumoto Imperfect Debugging Model

Goel and Okumoto [117] extend the J-M model by assuming that a fault is removed with probability p whenever a failure occurs. The program failure rate at the ith failure interval is

$$\lambda(t_i) = \phi[N-p(i-1)].$$

2.3.5 Models in The Curve Fitting Group

The curve fitting group of models finds a functional relationship between dependent and independent variables. Linear regression, quadratic regression, exponential regression, isotonic regression, and time series analysis, have been applied to software failure data analysis. The dependent variables are the number of errors in a program, the number of modules change in the maintenance phase, time between failures, and program failure rate. Models of each type are discussed below.

Estimation Of Errors Model

The number of errors in a program can be estimated by a linear [27] or quadratic [144] regression model. A general formula is

$$N = \sum_i a_i X_i$$

or

$$N = \sum_i a_i X_i + \sum_i b_i X_i^2$$

where

N	=	number of errors in the program
X_i	=	ith error factors
a_i, b_i	=	coefficients.

Typical error factors are software complexity metrics and the environmental factors. Most curve fitting models involve only one error factor. A few of them study multiple error factors.

Estimation Of Change Model

Belady and Lehman [37] use time series analysis to study the program evolution process. Some of the models studied by [37] are

$$
\begin{aligned}
M_R &= K_0 + K_1R + S + \varepsilon \\
C_R &= K_0 + K_1R + K_2R^2 + S + \varepsilon \\
C_R &= K_0 + K_1R + K_2R^2 + K_3HR_R + S + \varepsilon \\
HR_R &= K_1 + S + \varepsilon \\
CMH_D &= K_0 + K_1D + S + \varepsilon
\end{aligned}
$$

where

R	=	release sequence number
M_R	=	number of modules at release R
I_R	=	inter-release interval R
MH_R	=	modules handled in I_R
HR_R	=	MH_R/I_R; handle rate
C_R	=	MH_R/M_R; complexity
D	=	number of days since first release
CMH_D	=	cumulative modules handled up to day D
ε	=	error.

This model is applicable for software having multiple versions and evolving for a long period of time, for instance, an operating system.

Estimation Of Time Between Failures Model

Crow and Singpurwalla [76] argue that software failure may occur in clusters. Also addressed by Ramamoorthy and Bastani [285], failure data may come in clusters at the beginning of each testing when different testing strategies are applied one after another. To investigate whether clustering happens systematically, a Fourier series was used to represent time between failures [76]. Data from two software projects were analyzed. However, no statistical test was done to assess the adequacy of this model.

Estimation Of Failure Rate Model

Isotonic regression has been proposed to estimate the failure rate of a software. Given failure times $t_1, \ldots, t_n$, a rough estimate of failure rate at the ith failure interval is

$$\hat{\lambda}_i = \frac{1}{t_{i+1} - t_i} \quad .$$

Assuming that the failure rate is monotonically nonincreasing, an estimate of this function λ^*_i, i=1, 2, ..., n can be found by the least squares fit to the $\hat{\lambda}_i$, i=1, 2, ..., n. This problem can be written as a quadratic programming problem:

$$Min \sum_{i=1}^{n} (\hat{\lambda}_i - \lambda_i^*)^2 \ (t_i - t_{i-1})$$

subject to

$$\lambda_{i-1}^* - \lambda_i^* \geq 0$$

$$\lambda_n^* \geq 0 .$$

The objective function is the least squares fit of λ^*_i, and the constraints ensure monotonically nonincreasing of λ^*_i.

This nonparametric estimation of program failure rate has been suggested by Gubitz and Ott [122] and Miller and Sofer [235]. By imposing different assumptions to the problem, for example, monotonicity and convexity of the failure rate function, or equal spaced time intervals [235], the isotonic regression problem can be formulated into different forms.

2.3.6 Models in The Reliability Growth Group

Widely used in hardware reliability to measure and predict the improvement of the reliability program, the reliability growth model represents the reliability or failure rate of a system as a function of time, testing stage, correction action, or cost. Dhillon [84] summarizes 10 reliability growth models developed for hardware systems. This empirical approach is also adapted for predicting the progress of a software debugging process. Reliability growth models reported for software are summarized below.

Duane Growth Model

Plotting cumulative failure rate versus cumulative hours on log-log paper, Duane observed a linear relationship between the two. This model can be expressed as

$$\lambda_c(t) = N(t) / t = at^{-\beta}$$

and

$$\log \lambda_c = \log a - \beta \log t$$

where

$N(t)$	=	cumulative number of failure
t	=	total time
λ_c	=	cumulative failure rate
a, β	=	parameters.

The above formula shows that log λ_c is inversely proportional to log t. This model was adapted by Coutinho [73] to represent the software testing process. He plotted the cumulative number of deficiencies discovered and the cumulative number of correction actions made versus the cumulative testing weeks on log-log paper. These two plots revealed a find-and-fix cycle, and are jointly used to predict the testing progress. The least squares fit can be used to estimate the parameters of this model [84].

Wall and Ferguson Model

Wall and Ferguson [381] proposed a model similar to the Weibull growth model for predicting the failure rate of a software during testing. They used the following notation:

$N(t)$	=	cumulative number of failures at time t
$M(t)$	=	maturity (man-months of testing, CPU time, calendar time, or number of tests)
M_o	=	scaling constant
N_o	=	parameters to be estimated
$\lambda(t)$	=	failure rate at time t
λ_0	=	initial constant failure rate
$G(t)$	=	$M(t)/M_o$.

The model is summarized as follows:

$$N(t) = N_o [G(t)]^{\beta}$$
$$\lambda(t) = N'(t) = N_oG'(t) [G(t)]^{\beta-1} .$$

Let $N_oG'(t) = \lambda_0$, then

$$\lambda(t) = \lambda_0 [G'(t)]^{\beta-1}$$
$$= (\lambda_0 / \beta) \beta[G'(t)]^{\beta-1} .$$

For $0 < \beta < 1$, $\lambda(t)$ is a decreasing function of t. By letting $a = \lambda_0/\beta$, this model is similar to the Weibull growth model with failure rate

$$\lambda(t) = a\beta t^{\beta-1} .$$

This is the failure rate when failures follow the Weibull distribution. Note that the failure rate of the Weibull growth model can be derived from the Duane model. The MLEs of Weibull parameters can be found in [84]. Wall and Ferguson [381] tested this model on six software projects and found that failure data correlate well with the model. In their study, β lies between 0.3 and 0.7.

Wagoner's Weibull Model

Adapted from hardware reliability, Wagoner [380] uses a Weibull distribution to represent time between program failures. Let

$f(t)$	=	density function of time between failure
$\lambda(t)$	=	failure rate function
$R(t)$	=	reliability function
a,β	=	scale and shape parameters
n	=	total number of failures

n_i = number of failures up to the ith time interval

$F(t)$ = n_i/n.

The Weibull distribution has the following properties

$$f(t) = a\beta(at)^{\beta-1} \exp\,[-(at)^{\beta}]$$

$$R(t) = 1 - F(t) = \exp\,[-(at)^{\beta}]$$

and

$$\lambda(t) = a\beta(at)^{\beta-1}\ .$$

The parameters estimation can be found in [380].

Logistic Growth Curve Model

Suggested by Yamada and Osaki [398], the logistic growth curve model has been used to represent the cumulative number of errors detected during debugging. The expected cumulative number of errors, m(t), detected up to time t is

$$m(t) = \frac{k}{1+ae^{-\beta t}}$$

where k, a, and β are parameters to be estimated by regression analysis.

Gompertz Growth Curve Model

Nathan [261] adapted the Gompertz model to represent the cumulative number of errors corrected up to time t. The model has an S-shaped curve with the following form

$$N(t) = aA^{\gamma^t}$$

where

a = number of inherent errors

N(O) = number of corrections made before the first test interval is completed

$N(t)$	=	cumulative number of errors corrected at time t
A	=	$N(O)/a$
$\ln \gamma$	=	correction rate.

The above formula can be written as

$$\ln [\ln (N(t)/a)] = \ln [\ln (N(O)/a)] + t \ln \gamma$$

where a is the upper limit of N(t) when t approaches infinity.

The Gompertz model has been used in hardware reliability to predict system reliability. The model is as follows

$$R(t) = aB^{\gamma t}$$

where R(t) is the system reliability, a is the reliability upper bound, and γ is the rate of improvement. One method of estimating the parameters is given in Dhillon [84].

Hyperbolic Reliability Growth Model

Sukert [354] adapted the hyperbolic reliability growth model to represent the debugging process of software. He assumed that testing is divided into N stages, each consisting of one or more tests until a change is made. Success counts and failure counts are recorded and fitted to the model.

The following notation is used:

j	=	testing state
R_j	=	reliability at the jth state
γ	=	growth rate

R_∞	=	upper bound of the software reliability.

The reliability of the software at state j is

$$R_j = R_\infty - \gamma/j$$

and the least squares estimates of R_∞ and γ are in [209].

2.3.7 Models in The Program Structure Group

By using structure design and structure programming, a program can be decomposed into a number of functional units. These functional units or modules are the basic building blocks of software. The program structure model studies the reliabilities and interrelationship of the modules. It is assumed that failures of the modules are independent of each other. This assumption is reasonable at the module level since they can be designed, coded, and tested independently, but may not be true at the statement level. Two models involving program structure are discussed below.

Littlewood Markov Structure Model

Littlewood's model [213] represents the transitions between program modules during execution as the Markov process. Two sources of failures are considered in the model. The first source of failure comes from a Poisson failure process at each module. It is recognized that as modules are integrated, new errors will be introduced. The second source of failure is the interface between modules. Assuming that failures at modules and interfaces are independent of each other, Littlewood has shown that the failure process of the entire program is asymptotically Poisson. Let

N	=	number of modules
$P=(p_{ij})$	=	transition probability matrix of the process
$A=(a_{ij})$	=	infinitesimal matrix of the process

λ_i	=	Poisson failure rate of module i
q_{ij}	=	probability that transition from module i to module j fails
$\Pi=(\pi_i)$	=	limiting distribution of the process
μ'_{ij}	=	first moment of the waiting time distribution.

It can be shown that [213] as λ_i and q_{ij} approach zero, the program failure process is asymptotically a Poisson process with rate

$$\sum_{i=1}^{N} \pi_i \ (\lambda_i + \sum_{j \neq i} a_{ij} \ q_{ij}) \ .$$

Littlewood extends the above model by relaxing the assumption of exponential waiting time at each module. He assumes that the waiting time distribution can be approximated by its first and second moments. As λ_i and q_{ij} approach zero, the program failure process is asymptotically a Poisson process with rate

$$\frac{\sum_{ij} \pi_i \ p_{ij} \ (\mu'_{ij} \ \lambda_i + q_{ij})}{\sum_{i,j} \pi_i \ p_{ij} \ \mu'_{ij}} = \sum_{i} a_i \ \lambda_i + \sum_{ij} b_{ij} \cdot q_{ij}$$

where a_i represents the proportion of time spent in module i and b_{ij} is the frequency of transition from i to j.

Cheung's User-oriented Markov Model

Cheung's user-oriented software reliability model [69] estimates program reliability by representing a program as a reliability network. He uses a Markov model to represent the transitions among program modules and assumes that program modules are independent of each other. The execution starts with entry module N and ends with an exit module N_n. As the reliability of each module and the transition probability matrix of the Markov process are determined, the reliability of the program is the probability of successful execution from entry module to exit module at or before n steps. Let

n	=	number of modules
N_i	=	module i
R_i	=	reliability of module i
P^n	=	the nth power of matrix P
I	=	identity matrix
R_s	=	reliability of the program
C	=	state of correct output
F	=	state of failure
$Q=(q_{ij})$	=	transition probability matrix of the module transition
$P=(p_{ij})$	=	transition probability matrix of the Markov process
R	=	diagonal matrix with R_i at R(i,i) and zero elsewhere
M_{n1}	=	Minor of W(n,1)

$$G^T = \begin{bmatrix} 0 \ldots R_n \\ 1-R_1 \ldots 1-R_n \end{bmatrix}_{2xn}$$

Then

$$P = \begin{bmatrix} I & 0 \\ G & RQ \end{bmatrix}$$

and

$$\begin{aligned} R_s &= P^n(N_1, C) \\ &= S(N_1, N_n) R_n \end{aligned}$$

where

$$S = \sum_{k=0}^{\infty} (RQ)^k = (I - RQ)^{-1} = W^{-1} \quad .$$

Besides the evaluation of program reliability, a sensitivity analysis can be conducted to determine the most important module with the network. The importance of module i is defined as

$$I_i = \partial R / \partial R_i$$

where

$$R = R_n(-1)^{n+1} \; |M_{n1}| / |W| \; .$$

2.3.8 Models in The Input-Domain Group

Basic Input-Domain Model

A program maps the input space to the output space. Input space is the set of all possible input states. Similarly, output space is the set of all possible output states for a given program and input space. During the operational phase, some input states are executed more frequently than the others. A probability can be assigned to each input state to form the operational profile of the program. This operational profile can be used to construct the input-domain software reliability model.

In the input-domain model [264], software reliability is defined as the probability of successful run(s) randomly selected from the input space. Therefore, the reliability of one run, R(1), can be defined as

$$R(1) = \sum_{i} p_i e_i$$

$$e_i = \begin{cases} 0 & \text{if } I_i \text{ fails} \\ 1 & \text{otherwise} \end{cases}$$

or

$$R(1) = 1 - \lim_{N \to \infty} \frac{F_I}{N}$$

where

I_i	=	input state i
p_i	=	probability of running the ith input state
F_I	=	number of failures in N runs
N	=	number of runs
R(k)	=	reliability over k runs.

In the operational phase, if errors are not removed when failures occur, the probability of experiencing k failures out of M randomly selected runs follows a binomial distribution:

$$P_k = \binom{M}{k} [1 - R(1)]^k [R(1)]^{M-k} \quad .$$

During the testing phase, a sequence of M tests are selected randomly from the input space without repeating the same test. Then, the probability of k failures out of M runs follows a hypergeometric distribution:

$$G(k;N,F_I,M) = \frac{\binom{F_I}{k}\binom{N-F_I}{M-k}}{\binom{N}{M}} \quad .$$

If a sequence of k runs is not selected randomly from the operational profile, R(1) may be different for each run. In general, the reliability of k runs can be expressed as [265]

$$R(k) = \prod_{j=1}^{k} R_j(1)$$

where

R(k)	=	reliability over k runs
$R_j(1)$	=	R(1) of the jth input.

The maximum likelihood estimate of R(1) can be obtained by running some test cases. It can be expressed as

$$\hat{R}(1) = 1 - \frac{F_t}{N_t}$$

where

F_t = number of test cases that cause failure

N_t = number of test cases.

Since the number of elements in the input space is a very large number, the number of test cases has to be large in order to have a high confidence in estimation. To simplify the estimation of R(1), Nelson [265] modifies the above basic model by assuming that the input space is partitioned into m sets. As test cases are selected from each partition and all the errors from the test cases are removed, the reliability of one run can be formulated as

$$R(1) = \sum_i p_i \, (1 - f_i)$$

where

p_i = probability that an input is from partition i

f_i = probability that an input from partition i will cause failure.

Input-Domain Based Stochastic Model

The input-domain based stochastic model was proposed by Ramamoorthy and Bastani [285]. Unlike the failure rate model, which keeps track of the failure rate at failure times, this model keeps track of the reliability of each run given a certain number of failures have occurred.

The following notation is used in the discussion:

j	=	number of failures occurred
k	=	number of runs since the jth failure
$T_j(k)$	=	testing process for the kth run after the jth failure
$f(T_j(k))$	=	severity of testing process; $0 < f(T_j(k)) < 1/\lambda_j$
λ_j	=	error size given j failures have occurred; a random variable
$V_j(k)$	=	probability of failure for the kth run after j failures; $f(T_j(k))\lambda_j$
$R_j(k \mid \lambda_j)$	=	probability that no failure occurs over k runs after j failures
$E_{\lambda j}(\cdot)$	=	expectation over λ_j
Δ_j	=	size of the jth error
X	=	random variable that follows distribution F.

Then

$$R_j(k \mid \lambda_j) = \prod_{i=1}^{k} [1 - V_j(i)]$$
$$= \prod_{i=1}^{k} [1 - f(T_j(i))\lambda_j]$$

and

$$R_j(k) = E_{\lambda_j} [\prod_{i=1}^{k} [1 - f(T_j(i))\lambda_j]] .$$

2.3.9 Models in The Execution Path Group

The basic idea of the execution path model is similar to that of the input-domain model. The model is based on (1) the probability that an arbitrary path is selected, (2) the probability that an arbitrary path will cause a failure, and (3) the time required to execute a path. By partitioning the input space into disjoint subsets, some authors [265,328] implicitly assume that each partition corresponds to a logic path. Since one logic path may include more than one physical path and two logical paths may share the same physical path, the question of whether the execution path model should be based on logical path or physical path remains unanswered.

If the logical path approach is used, testing should start with partitioning the input space and finding out the logic path for each partition. The test cases can then be selected from the disjoint subsets. If the physical path approach is used, testing should start with enumerating all the possible paths [228]. The test cases are then selected from those paths. Since the relationship between input state, partition of input state, and path, is not readily apparent, the execution path model is discussed separately from the input domain model.

Shooman Decomposition Model

The decomposition model proposed by Shooman [328] assumes that the program is designed using structured programming methodology. Hence, the program can be decomposed into a number of paths. He also assumes that the majority of the paths are independent of each other. Let

N	=	number of test cases
k	=	number of paths
t_i	=	time to run test i
$E(t_i)$	=	expected time to run test i
q_i	=	probability of error on each run of case i
q_0	=	probability of system failure on each run
f_i	=	probability that case i is selected
n_f	=	total number of failures in N test
H	=	total testing hours
λ_0	=	program failure rate.

Then

$$n_f = N \sum_{i=1}^{k} f_i q_i$$

and

$$q_0 = \lim_{N \to \infty} n_f / N.$$

Assume that on the average a failure in path i takes $t_i/2$ to uncover,

$$H = N \sum_{i=1}^{k} f_i t_i (1 - q_i/2)$$

and

$$\lambda_0 = \lim_{N \to 0} n_f / H.$$

This model is very similar to the basic input-domain model. If R(1) denotes the reliability of an arbitrary path, then

$$R(1) = 1 - \sum_{i=1}^{k} f_i q_i .$$

2.3.10 Models in The Nonhomogeneous Poisson Process Group

The nonhomogeneous Poisson process model (NHPP) represents the number of failures experienced up to time t as an NHPP, $\{N(t), t \geq 0\}$. The main issue in the NHPP model is to determine an appropriate mean value function to denote the expected number of failures experienced up to a certain time point. With different assumptions, the model will end up with different functional forms of the mean value function.

One simple class of NHPP model is the exponential mean value function model, which has an exponential growth of the cumulative number of failures experienced. Musa exponential model [259] and Goel and Okumoto NHPP model [119] belong to this class. Other types of mean value functions suggested by Ohba [268] are the delayed S-shaped growth model, inflection S-shaped growth model, and hyperexponential growth model.

The NHPP model has the following assumptions [332].

- The failure process has an independent increment; i.e., the number of failures occurred during the time interval (t, t+s] depends on current time t and the length of time intervals s, and does not depend on the past history of the process.

- The failure rate of the process is

  $$\begin{aligned}\Pr\{\text{exactly 1 failure in } (t,t+\Delta t) &= \Pr\{N(t+\Delta t) - N(t)=1\} \\ &= \lambda(t)\,\Delta t + o(\Delta t)\end{aligned}$$

  where $\lambda(t)$ is the hazard function.

- During a small interval, Δt, the probability of more than one failures is negligible, i.e.,

  $$\Pr\{2 \text{ or more failures in } (t, t+\Delta t)\} = o(\Delta t)\ .$$

- Initial condition is $N(0)=0$.

Based on the above assumptions, it can be shown that N(t) has a Poisson distribution with mean $\mu(t)$, i.e.,

$$Pr\{N(t)=k\} = \frac{[\mu(t)]^k}{k!}\, e^{-\mu(t)}\ .$$

By definition, the mean value function of the cumulative number of failures, $\mu(t)$, can be expressed in terms of the failure rate of the program, i.e.,

$$\mu(t) = \int_0^t \lambda(s)\,ds.$$

The reliability function of the program is

$$R(t) = e^{-\mu(t)} = \exp[-\int_0^t \lambda(s)\,ds].$$

The exponential growth curve is a special case of NHPP with

$$\mu(t) = NF(t) = N\left\{1 - \exp\left[-\int_0^t \phi(s)\,ds\right]\right\}$$

and

$$\lambda(t) = Nf(t) = N\phi(t)\ \exp\left[-\int_0^t \phi(s)\,ds\right]$$

where

$\phi(s)$	=	a proportional function of s
N	=	is the number of initial errors.

Based on the above general NHPP model, some special models are discussed below.

Musa Exponential Model

The Musa exponential model [259] can be summarized as

$$\phi(t) = \phi$$
$$\mu(t) = X_0\,[1 - e^{-\phi Bt}]$$

and

$$\lambda(t) = X_0\,\phi\,B\,e^{-\phi Bt}$$
$$= \phi\,B\,[X_0 - \mu(t)].$$

Goel and Okumoto NHPP Model

The Goel-Okumoto model [119] has mean value function of

$$\mu(t) = N(1 - e^{-\phi t})$$

and

$$\begin{aligned}\lambda(t) &= \frac{\partial\mu}{\partial t}\\ &= N\phi\, e^{-\phi t},\end{aligned}$$

where

N	=	the number of initial errors
ϕ	=	a proportional constant.

An extension of the exponential mean value function model has been suggested by Yamada and Osaki [399]. They assume that faults come from different sources with different failure rates. Let

n	=	number of types of fault
ϕ_i	=	failure rate of each type i fault
p_i	=	probability of type i fault.

Then

$$\mu(t) = N\sum_{i=1}^{n} p_i[1 - e^{-\phi_i t}].$$

Delayed S-shaped Growth Model

The delayed S-shaped model [268,391] divides the debugging process into a fault detection stage followed by a fault removal stage. A fault is said to be removed from the program if it goes through both stages. By assuming that the probability of fault detection is proportional to the number of faults not detected, and the probability of fault removal is proportional to the number of faults detected but not removed, this model can be expressed by the following differential equations

$$h'(t) = a[N - h(t)]$$
$$\mu'(t) = \lambda[h(t) - \mu(t)]$$

where

N	=	number of initial faults
h(t)	=	number of faults detected at time t
$\mu(t)$	=	number of faults removed at time t
a	=	detection rate of each undetected fault
λ	=	removal rate of each detected but not yet removed fault.

By further assuming that $\mu=\lambda=\phi$, $\mu(t)$ can be solved as

$$\mu(t) = N\,[1 - (1 + \phi t)\,e^{-\phi t}].$$

This function becomes the mean value function of the NHPP model.

Inflection S-shaped Growth Model

Ohba [268] models the dependency of faults by postulating the following assumptions:

- Some of the faults are not detectable before some other faults are removed.
- The detection rate is proportional to the number of detectable faults in the program.
- Failure rate of each detectable fault is constant and identical.
- All faults can be removed.

Then, the program failure rate during the ith failure interval is defined as

$$\lambda_i = \phi \, \mu_i \, [N - (i-1)]$$

where μ_i is the proportion of detectable faults when i faults are removed, N is the number of initial faults, and $\mu_i[N - (i-1)]$ is the number of detectable faults at the ith failure interval. As more faults are detected, more dependent faults become detectable. Therefore, the proportion of detectable faults is an increasing function of the detected faults. Let this function be

$$\mu_i = r + i(1-r)/N, \quad 0 \leq r \leq 1.$$

Based on the above formulation, it can be shown that the mean value function of this NHPP model is

$$\mu(t) = \frac{N(1 - e^{-\phi t})}{1 + (1-r)\, r^{-1}\, e^{-\phi t}} \; .$$

As r approaches 1, the above model approaches the exponential growth model. As r approaches 0, the above model approaches the logistic growth model.

Hyperexponential Growth Model

The hyperexponential growth model [139,268] is based on the assumption that a program has a number of clusters of modules, each having a different initial number of errors and a different failure rate. Examples are new modules versus reused modules, simple modules versus complex modules, and modules which interact with hardware versus modules which do not interact with hardware. Since the sum of exponential distributions becomes a hyperexponential distribution, the mean value function of the hyperexponential class NHPP model is

$$\mu(t) = \sum_{i=1}^{n} N_i [1 - e^{-\phi_i t}]$$

where

n	=	number of clusters of modules
N_i	=	number of initial faults in cluster i
ϕ_i	=	failure rate of each fault in cluster i.

2.3.11 Models in The Markov Group

The Markov model is a generalized bug-counting model which represents the number of remaining faults at time t, N(t), as a continuous time discrete state Markov chain. The state of the Markov process is the number of remaining faults. The continuous time is the exponential time-to-failure. Binomial type model and Poisson type model are special cases of the Markov process.

A Markov process has the property that the future behavior of the process depends only on the current state and is independent of its past history. This assumption seems reasonable for software failure processes. It can be argued that the future of a failure process depends only on the number of remaining faults at the present time and is not affected by the past error content [248].

A general Markov process allows transitions to occur from any state to any other state. In other words, multiple faults can be removed or introduced at each debugging. This model is suggested by Sumita and Shanthikumar [355]. In practice, there were not enough failure data to estimate all parameters of the transition probability matrix. Some models have been developed as special cases of the Markov chain. They are the linear death model with perfect debugging, stationary linear death model with imperfect debugging, and nonstationary linear death model with perfect debugging. These models are discussed below.

Linear Death Model With Perfect Debugging

The Jelinski and Moranda model [150] is essentially a linear death model with perfect debugging. Let

p_{ij}	=	probability of transition from state i to state j
$P_k(t)$	=	$Pr\{N(t)=k\}$; probability of k remaining fault at time t.
ϕ	=	failure rate of each fault.

The transition probabilities can be expressed as

$$p_{ij} = \begin{cases} 1 & j=i-1 \\ 1 & i=j=0 \\ 0 & \text{otherwise} \end{cases} \qquad i,j = 0,1, \dots ,N$$

and the transition rate diagram is shown in Figure 2.2.

The differential-difference equation of $P_k(t)$, say $P'_k(t)$, is

$$P'_k(t) = (k+1)\ \phi\ P_{k+1}(t) - k\ \phi\ P_k(t).$$

Solving the above equation with the initial condition N(0)=N, all the performance measures of the J-M model derived in the binomial model can also be derived from this Markov chain point of view.

Linear Death Model With Imperfect Debugging

Suggested by Goel and Okumoto [116,117], the transition probabilities of the linear death model with imperfect debugging can be expressed as

$$p_{ij} = \begin{cases} p & , j=i-1 \\ q=1-p & , j=i \\ 1 & , i=j=0 \\ 0 & , \text{otherwise} \quad i,j = 0,1, \ldots ,N \end{cases}$$

where p is the probability of successful debugging. The transition rate diagram is shown in Figure 2.3.

This model assumes a probability q of not removing the fault whenever a failure occurs. Some performance measures are summarized as follows. The expected number of remaining faults at time t is

$$M(t) = E[N(t)] = Ne^{-p\phi t}.$$

The expected number of failures up to time t is

$$\mu(t) = E[X(t)] = N/P\ [1 - e^{-p\phi t}].$$

The expected number of imperfect debugging errors by time t is

$$\mu_i(t) = q\ \mu(t).$$

The reliability function of the kth failure interval is

$$R_k(t) = \sum_{j=0}^{k-1} \binom{k-1}{j} p^{k-j-1} q^j \bar{F}_{N-(k-j-1)}(t)$$

where

$$\bar{F}_j(t) = e^{-j\phi t}.$$

It has been shown [116] that

$$R_k(t) \approx \exp\{-[N - p(k-1)]\ \phi t\}.$$

Figure 2.2 Linear death with perfect debugging

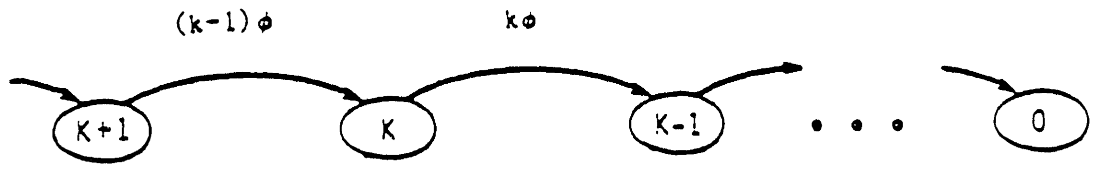
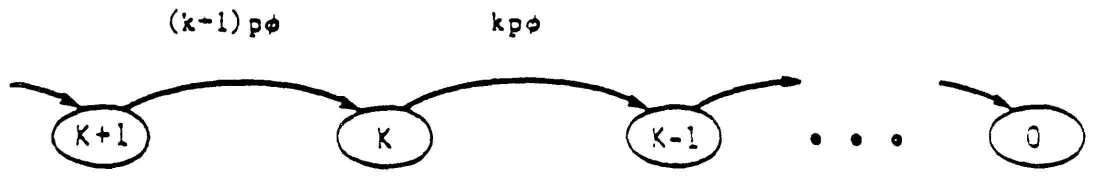


Figure 2.3 Linear death with imperfect debugging



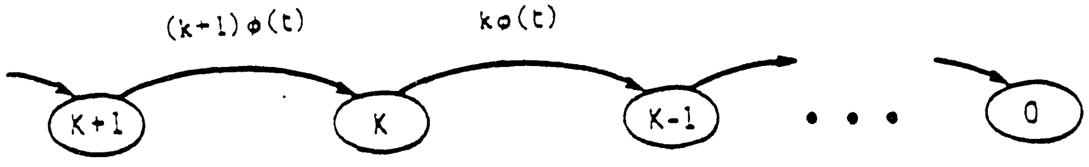
Nonstationary Linear Death Model With Perfect Debugging

Suggested by Shanthikumar [319,320], the transition probabilities of the nonstationary linear death model with perfect debugging can be expressed as

$$p_{ij} = \begin{cases} 1 & , \ j=i-1 \\ 1 & , \ i=j=0 \\ 0 & , \ \text{otherwise} \quad j = 0,1, \ldots ,N \end{cases}$$

and the transition rate diagram is shown in Figure 2.4.

Figure 2.4 Nonstationary linear death with perfect debugging



The differential-difference equation of $P_k(t)$ is

$$P'_k(t) = (k+1)\,\phi(t)\,P_{k+1}(t) - k\,\phi(t)\,P_k(t).$$

Solving the above equation with the initial condition $N(0) = N$,

$$P_k(t) = \binom{N}{k}[F(t)]^{N-k}\,[1 - F(t)]^k$$

where

$$F(t) = 1 - \exp\left[-\int_0^t \phi(s)\,ds\right].$$

This is the binomial type model derived in the failure rate model. Other performance measures can be found in Section 2.3.4.

2.3.12 Models in The Bayesian Group and Models in The Unified Group

The Bayesian model and unified model are two other types of probabilistic software reliability models. The Bayesian approach was discussed by Jewell [151], Serra and Barlow [317], Kuo [181], Littlewood [206,209], Littlewood and Verrall [217], and Langberg and Singpurwalla [183]. Besides the nonstationary linear death models [176,181], other unified models are the exponential order statistics model by Miller [234], and Scholz [311], and the shock model by Langberg and Singpurwalla [183].

2.4 Fault Tolerant Software Reliability Models

It is important to provide very high reliability to critical applications such as aircraft controller and nuclear reactor controller software. No matter how thorough the testing, debugging, modularization, and verification of software are, design bugs still plague the software. After reaching a certain level of refinement of the software, any effort to increase the reliability even by a small margin will take exponential cost. Consider, for example, a fairly reliable software subjected to continuous testing and debugging, and guaranteed to have no more than 10 faults throughout the life cycle. Now, in order to improve the reliability such that, for example, only 7 faults may be tolerated, the effort and cost to guarantee this may be enormous. An alternate approach to increase fault tolerance is to provide redundancy. Redundancy has been accepted as a viable approach to obtain reliability from unreliable components.

Fault tolerant software assures the reliability of the system by use of protective redundancy at the software level. There are two main techniques for obtaining fault tolerant software:

- Recovery Block scheme
- N-Version Programming

2.4.1 Recovery Block Scheme

The recovery block scheme [9, 288] consists of three elements: (1) a primary module, which executes critical software functions, (2) an acceptance test, which tests the output of the primary module after each execution, and (3) a set of alternate modules, which performs the same function as the primary module.

The simplest scheme of the recovery block is:

```
Ensure T
    By P
    Else by $Q_1$
     Else by $Q_2$
         .
         .
         .
      Else by $Q_{n-1}$
Else Error
```

where T is the acceptance test condition that is expected to be met by successful execution of either the primary module P or the alternate modules Q_1, Q_2, ... , Q_{n-1}.

When the failure of the primary module is detected by an acceptance test, an alternate module is executed. If all the alternate modules are exhausted, the system crashes. Pham and Upadhyaya [277] have developed a model to obtain the optimal number of modules in the recovery block scheme given the reliabilities of the individual modules. An optimization model for a modified recovery block scheme is also given in [38], and a numerical method is used to obtain the solution.

2.4.2 N-Version Programming

N-version programming (NVP) [65] is defined as the independent generation of $N \geq 2$ functionally equivalent programs, called *versions*, from the same initial specification [15]. *Independent generation of programs* means that the programming efforts are carried out by N individuals or groups that do not interact with respect to the programming process. Whenever possible, different algorithms, techniques, programming languages, environment, and tools are used in each effort [65].

N-version programming is the software equivalent of the N-Modular Redundancy technique. In this technique, N program versions are executed in parallel on identical input and the results are obtained by voting upon the outputs from the individual programs. The advantage of NVP is that when a version failure occurs, no additional time is required for reconfiguring the system and redoing the computation [274]. Pham [274] has given a cost model to obtain the optimal number of program versions that minimizes the expected cost of the NVP scheme. Pham further determined the optimum number of versions that minimizes the expected total system cost subject to a restricted type I design error. The problem of maximizing NVP system reliability subject to a constraint on expected system cost is also obtained in [274].

The main difference between the recovery block scheme and the N-version programming is that the modules are executed sequentially in the former. Therefore, recovery block generally is not applicable to critical systems where real-time response is of great concern, thus precluding it from our discussion.

N-version programming has been researched thoroughly during the past decade. Correlated errors form a main source of failure of the N-version programs. Correlated errors can be minimized by design diversity [14, 15, 302]. A design paradigm has been developed to assure design diversity in N-version software [221]. Several experiments have been conducted to validate the assumption of error independence in multiple versions [169], to analyze the types of faults [49, 327], to investigate the use of self checks and voting in error detection [192], and to establish the need for a complete and unambiguous specification [162]. The problem of consistent comparison has been investigated in [50] and the voting algorithms have been studied in [109], [220]. There has been some effort on modeling the reliability of fault tolerant software [1, 11, 202, 267, 312, 397]. An environment for developing fault tolerant software has been discussed in [281].

However, in critical systems with real-time deadlines, voting at the end of the program, as in the basic N-version programming, may not be acceptable. Therefore, voting at intermediate points is called for. Such a scheme, where the comparison of results is done at intermediate points, is called the Community Error Recovery (CER) scheme [375] and is shown to offer a higher degree of fault tolerance compared to the basic N-version programming [65]. This approach, however requires the synchronization of the various versions of the software at the comparison points.

Another scheme which adopts intermediate voting is the N self checking programs [185], where each version is subject to an acceptance test or a checking by comparison. Whenever a particular version raises an exception, the correct result is obtained from the remaining versions and execution continued. This method is not much different from the CER approach, the only difference being the on-line detection in the former by an acceptance test rather than a comparison.

It is apparent from this literature study that there has been no significant advance in the reliability enhancement literature for the past 15 years or so. Most of the papers appearing recently on software fault tolerance are variations of the well known N-version programming or recovery block schemes. In Section 4, we propose a new approach, called a self checking duplex system, for the enhancement of software reliability. This scheme incorporates redundancy at two levels and can increase the reliability of software in critical systems to significant levels.

This section has reviewed classification of the software reliability models (mainly by the modeling techniques) and fault tolerant software reliability models. Table 2 summarizes related references for each category. These models are the fundamental sources for the study of software-related problems. Besides reliability assessment, systems reliability optimization, system design, reliability cost model, and hardware software systems, these are other areas in which software reliability models can be applied.

Table 2 Summary of References

Group Models	References
General Software	2, 26, 64, 7981, 85, 91, 114, 120, 121, 130, 133, 142,
Reliability Models	143, 173, 193, 194, 212, 219, 248, 253, 293, 299, 321, 332, 340, 354, 359, 363, 374
Error Analysis	6, 98, 112, 146, 245, 246, 290, 300, 371, 373
Software Science	123, 197, 307, 324, 325
Complexity Metrics	27, 28, 29, 30, 31, 66, 92, 102, 110, 144, 153, 198, 228, 241, 279, 294, 295, 318, 324, 341, 361, 371, 382, 387, 405
Error Seeding Group	28, 32, 139, 289, 305, 306, 369
Curve Fitting Group	37, 40, 58, 76, 122, 127, 235, 338, 361
Failure Rate Group	10, 28, 63, 80, 86, 113, 126, 150, 152, 172, 205, 206, 207, 208, 210, 211, 214, 242, 243, 262, 263, 320, 330, 345
Reliability Growth Group	73, 75, 84, 204, 261, 381
Program Structure Group	19, 69, 213, 216, 308, 357, 358
Input-Domain Group	33, 264, 265, 283, 285, 340, 343, 349, 360, 385

Execution Path Group	89, 90, 93, 328
Nonhomogeneous Poisson Process Group (NHPP)	68, 119, 124, 181, 238, 250, 251, 257, 259, 268, 297, 347, 401
Markov Group	116, 117, 176, 187, 215, 319, 331, 355
Bayesian Group	3, 151, 206, 209, 217, 304, 364, 365
Other Unified Group	19, 171, 183, 234, 311
Cost Models	60, 87, 103, 104, 107, 115, 170, 181, 198, 256, 272, 273, 296, 324, 400
Hardware-Software Systems	61, 118, 128, 129, 131, 167, 182, 340, 356, 364, 365
General Fault Tolerant Systems	1, 8, 9, 11, 16, 34, 39, 41, 60, 62, 71, 72, 95, 132, 149, 164, 185, 187, 189, 220, 222, 233, 266, 281, 301, 302, 313, 327, 336, 362
N-Version Programming	14, 15, 49, 50, 65, 109, 132, 162, 169, 192, 221, 312, 379
Recovery Block	9, 78, 132, 166, 277, 288, 312
Other Fault Tolerant Approaches	11, 12, 38, 174, 184, 221, 267, 312, 377

3. PROPOSED TECHNIQUE FOR SOFTWARE RELIABILITY ENHANCEMENT

Before one could apply the N-version programming schemes to enhance the reliability of critical software, such as the nuclear reactor controller system, their feasibility should first be determined. Nuclear reactor controller software requires ultra-high reliability. The existing N-version schemes may not be able to offer the required reliability because of vulnerability to failures due to identical causes. If the majority of the versions fail because of common design error, then a wrong result may be gi''en by voting on incorrect outputs. The likelihood of common-cause failures in nuclear controller software cannot be ruled out because of the complexity o: the application.

Efforts to alleviate the common-cause failures include the development of diverse versions by independent teams so as to minimize the commonalities between the various versions of the software [14]. But according to recent research at the University of California, Irvine [49], the use of different languages and design philosophy has little effect on the reliability in N-version programming because people tend to make similar logical mistakes in a difficult-to-program part of the software. Thus, in the presence of a common-cause failure all the different variations of the N-version programming prove to be equally useless. It seems beneficial to have a single version in order to minimize cost.

According to the latest research on the topics related to software reliability, fault tolerance is a highly recommended application for the nuclear reactor control system to improve the reliability of the embedded software. However, a new approach that could alleviate the weakness of the existing fault tolerant software reliability models is even more desirable.

In this section, we propose a new technique called *Self-Checking Duplex System* applicable to real time software embedded systems. Section 3.1 presents more details about this new technique. Further research is proposed to design a model using this technique. Also, new design guidelines are proposed to reduce or eliminate the common cause failures in software programs. These guidelines are presented in Section 3.2.

3.1 Self-Checking Duplex System

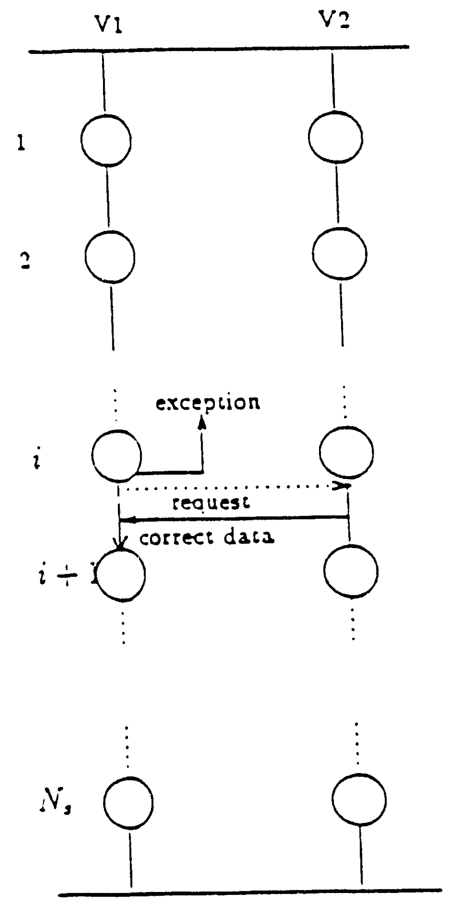
The University of California, Irvine [49] research suggests not to go for N-version if it is known that the probability of making common mistakes while programming cannot be totally avoided. Instead, we recommend (1) developing fewer versions, (2) minimizing the errors in the individual versions, and (3) minimizing/eliminating the incidence of common-cause failures in these versions.

If individual versions are made highly reliable, an ultra-high reliability can be achieved merely by having two versions. These two versions should be made self-checking and should work simultaneously as a duplex system as shown in Figure 3.1. If one of the versions (module i in the illustration of Figure 3.1) raises an exception, correct results can be obtained from the other version as shown in the Figure 3.1. We call this new approach a self-checking duplex system. Figure 3.1 shows a simple architecture of the proposed system where both versions are represented by a sequence of N_s modules.

Although N self-checking versions can be used [185], our preliminary studies show that two versions are sufficient to raise the reliability to acceptable levels using our new approach. It is also more practical to have as few self-checking versions as possible because of the high cost of developing N different self-checking versions.

Self-checking software can be developed in a variety of ways [138, 404]. Self-checking provides an online detection of errors and prevents the contamination of the

Figure 3.1 Duplex System



software by not letting the errors manifest. Software integrity can be assured by testing for illegal branching, infinite loops, wrong branching, etc., and testing for functionality and reasonableness of the results. It is easier to incorporate self-checking assertions into the software during the design stage, because the team that develops the software is expected to have the best understanding of the problem. A good understanding of the application and the algorithms is deemed important for creating and placing meaningful assertions in the code. Both local and global self-checking assertions need to be incorporated to guarantee a high reliability. Hua and Abraham provide systematic method for developing the self-checking assertions [138]. In the following paragraphs we show how self-checking assertions can provide ultra-high reliability.

Let the executable assertions be inserted in a module both locally and globally. By inserting local and global assertions, it is possible to check not only the internal states of the modules, but also the input/output specifications. Because the inputs to an intermediate module such as i + 1 (see Figure 3.1) are reset to the correct value by the corresponding module of the other version if and only if an error is detected, any undetected error at module i will propagate to the next module i+1. Let p represent the probability of detecting an error in module i of a self-checking version. Now, given that an error goes undetected at the ith, i+1th, ... , i+l-1th module, the probability of this error being detected at the i+lth module of the version is

$$p_l = p \cdot \sum_{j=1}^{i+l} (1 - p)^{j-1} .$$

This probability is very high as described in the following paragraph.

Suppose that the probability of detecting a design error at a particular module by self-checking is 0.9. If an error is not detected at this module, the probability of this error being detected at the following module is 0.99 and the probability of detecting it at the next module is 0.999 and so on. This establishes that self-checking assertions could be a very powerful tool in increasing the software reliability.

The self-checking duplex system incorporates fault tolerance in two layers. The first layer of protection is provided by self checking assertions. The second layer is duplication. In the self-checking system, if one of the versions detects an error at the end of the current module, results are obtained from the other version. After exchanging the correct results, both versions will continue execution in a lock-step fashion. Finally, the outputs of the duplicated versions are compared for consistency before accepting the result as correct. By keeping the size of the modules sufficiently small, a larger number of errors can be masked by this approach. However, too small a size for the module will increase the overhead of implanting the self-checking assertions. The analysis of the reliability and the optimal modules size selection requires further research. This is a significant part of the proposed research.

Common-cause failure is still a problem in the self-checking duplex system. There is no known technique to address this in N-version programming. Therefore, it can only be attempted to reduce the common-cause failures by design diversity. Next, we discuss and provide some guidelines to reduce the probability of common-cause failures.

3.2 Reduction of Common-Cause Failures

Clearly, a complex software is developed in a modular fashion and not all the modules are equally complex and difficult to design. Therefore, it will be an accurate statement if we say that the common-cause failures are confined to the "difficult to logically understand and design" part of the problem. The common-cause failures can be reduced if such critical parts are identified and certain design guidelines are followed.

We propose the following design guidelines that will reduce or eliminate the common-cause failures:

- Techniques to identify critical parts in a program. Generally, the control flow complexity of an algorithm indicates the level of difficulty. We can therefore use the McCabe measure to identify the critical parts of a program [228].

- The manager of the project should identify the critical sections of the problem, meet with the program development teams individually, and steer them to different techniques for solving the critical parts. As an example, suppose that the critical part involves sorting a file; then, one team should be asked to use, Quicksort [5]. The other team should be asked not to use Quicksort but to use some other naive scheme. In this way, the probability of committing identical logical mistakes can be reduced or eliminated.

Our proposal will address the development of additional design guidelines to minimize the common-cause failures.

4. GENERAL DISCUSSION AND FUTURE RESEARCH

Enhancement of reliability of a software module followed by a global enhancement by maintaining multiple versions of the software will bring the reliability to reasonable levels. Such an approach [67] has found success in Wafer Scale Integration and we would like to apply this concept to software reliability improvement. Our analysis shows that if a single version is made sufficiently self-checking, design errors can be caught with very high probability and corrected appropriately with a very high probability by duplication.

It is essential to estimate the reliability of the software before any redundancy schemes, such as the self-checking duplex scheme proposed here, can be applied. Techniques should be developed to estimate the number of errors that remain in the application software at the time of release; then, if the reliability is below acceptable levels, the test expenditure has to be spent on debugging and testing to obtain the desired level of reliability. Kubat and Koch [179] investigated several test protocols to estimate the number of remaining errors in a system at the time of software release. Singpurwalla [339] has presented a decision-theoretic approach to determine the optimal time interval for testing and debugging software. Based on these techniques, we propose to develop a model to predict the reliability of a software program. This model will greatly help in determining the amount of effort required before stopping testing and debugging of the software.

Computers used in harsh environments, such as the nuclear power industry, may be subject to radiation. Radiation, power supply glitches, humidity, and vibration, may give rise to what is called hardware-induced software faults. Such faults are generally transitory, but may be sufficient to disrupt computation. Interestingly, the majority of failures in computer systems has been recognized to be of temporary nature, including operator faults. Although the proposed self-checking duplex system provides fault tolerance to design errors, it can also handle a class of transient faults. However, transient faults that affect both versions cannot be tolerated by the self-checking duplex system. In such cases, a backward error recovery such as checkpointing and rollback [377] can be used.

In summary, we propose to investigate the following seven problems:

1. Develop a model to predict the reliability of a single software program.

2. Develop a cost model to determine the optimal time interval for testing the software.

3. Develop a technique for the development of self-checking software versions.

4. Evaluate the performance of the self-checking duplex system and the development of an appropriate reliability model to estimate the reliability of the fault tolerant software.

5. Determine the optimal size of modules in the self-checking software versions.

6. Develop general and more comprehensive design guidelines to minimize the common-cause failures among the various versions of the software.

7. Consider hardware-induced software errors in the nuclear reactor controller software.

REFERENCES

[1] Abbott, R. J. "Resourceful systems for fault tolerance, reliability, and safety," ACM Computing Survey, Vol. 22, No. 1, March 1990.

[2] Abdel-Ghaly, A. A., P. Y. Chan, and B. Littlewood. "Evaluation of competing software reliability predictions," IEEE Transactions on Software Engineering, Vol. SE-12, No. 9, 1986.

[3] Adams, E. N. "Optimizing preventive service of software products," IBM Journal Research and Development, Vol 28, No. 1, 1984.

[4] Adams, E. N. "Minimizing cost impact of software defects," IBM Research Division, Report, RC 8229 (35669), April 1980.

[5] Aho, A. V., J. E. Hopcroft, and J. E. Ullman. "Data structures and algorithms," Addison-Wesley, 1983.

[6] Akimaya & Fumo. "An example of software system debugging," IFIP Congress, 1971.

[7] Anderson, E. E. "A heuristic for software evaluation and selection," Software - Practice and Experience, Vol. 19, No. 8, August 1989.

[8] Anderson, T., P. A. Barrett, D. N. Halliwell, and M. R. Moulding. "Software fault tolerance: An evaluation," IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, 1985.

[9] Anderson, T. and P. A. Lee. Fault tolerance: Principles and practice. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[10] Angus, J. E. "The application of software reliability models to a major CCCI system," Proceedings Annual Reliability and Maintainability Symposium, 1984.

[11] Arlat, J., K. Kanoun, and J. C. Laprie. "Dependability modeling and evaluation of software fault tolerant systems," IEEE Transactions on Computers, Vol. 39, No. 4, April 1990.

[12] Arlat, J., K. Kanound, and J. C. Laprie. "Dependability evaluation of software fault-tolerance," in Proceedings 18th IEEE International Symposium Fault Tolerant Computing (FTCS-18), Tokyo, Japan, June 1988.

[13] Ashrafi, N., R. C. Baker, and J. P. Kuilboer. "Proposed structure for decomposition software reliability prediction model," Software Information Technology, Vol. 35, No. 7, 1990.

[14] Avizienis, A., M. Lyn and W. Schutz. "In search of effective diversity: A six-language study of fault tolerant flight control software". Digest of 18th International Symposium on Fault Tolerant Computing, Tokyo, Japan, 1988.

[15] Avizienis, A. and J. P. J. Kelly. "Fault tolerance by design diversity: Concepts and experiments," IEEE Computers, Vol. 17, August 1984.

[16] Avizienis, A. "Design diversity - The challenge of the eighties," in Proceedings 12th Annual International Symposium Fault-Tolerant Computing, Santa Monica, CA, June 1982.

[17] Bai, D. S. and W. Y. Yun. "Optimum number of errors corrected before releasing a software system," IEEE Transactions on Reliability, R-37, No. 1, 1988.

[18] Bailey, N. T. J. The Element of Stochastic Process, John Wiley & Sons, New York, 1964.

[19] Barlow, R. E. & N. D. Singpurwalla. "Assessing the reliability of computer software and computer networks: an opportunity for partnership with computer scientists," American Statistician, Vol. 39, No. 2, 1985.

[20] Barrow, H. G. "VERIFY: A program for proving correctness of digital hardware designs," Artificial Intelligence, Vol. 24, December 1984.

[21] Basili, V. R. "Quantitative evaluation of software engineering methodology," in Proceedings First Pan Pacific Computer Conference, Melbourne, Australia, September 10-13, 1985.

[22] Basili, V. R. and B. T. Perricone. "Software errors and complexity: An empirical investigation," Communications of the ACM, Vol. 27, No. 1, 1984.

[23] Basili, V. R. and R. W. Selby. "Data collection and analysis in software research and management," in Proceedings America Statistic Association and Biometric Society Joint Statistical Meetings, Philadelphia, PA, August 13-16, 1984.

[24] Basili, V. R. and R. W. Selby. "Comparing the effectiveness of software testing strategies," IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, 1987.

[25] Basili, V. R. and D. M. Weiss. "A methodology for collecting valid software engineering data," IEEE Transactions on Software Engineering, Vol. SE-10, No. 6,

1984.

[26] Basili, V. R. & R. W. Selby, Jr. "Four applications of a software data collection and analysis methodology," NATO Advanced Study Institute, The Challenge of Advanced Computing Technology to System Design Method, 1985.

[27] Basili, V. R., R. W. Selby, & T. Y. Phillips. "Metric analysis and data validation across Fortran projects," IEEE Transactions on Software Engineering, SE-9, No. 6, 1983.

[28] Basili, V. R. & D. H. Hutchens. "An empirical study of a syntactic complexity family," IEEE Transactions on Software Engineering, SE-9, No. 6, 1983.

[29] Basili, V. R. & R. W Reiter. "Evaluating automable measure of software development," Proceedings Workshop on Quantitative Software Models, 1979.

[30] Basili, V. R. & A. J. Turner. "Iterative enhancement: a practical technique for software development," IEEE Transactions on Software Engineering, SE-1, 1985.

[31] Basili, V. R. & B. T. Perricone. "Software errors and complexity: an empirical investigation," Communications of the ACM, 27, No. 1, 1984.

[32] Basin, S. L. Estimation of software error rates via capture-recapture sampling. Science Application, Inc., Palto Alto, CA., 1973.

[33] Bastani, F. B. An Input Domain Based Theory of Software Reliability and Its Application. Ph.D. dissertation, University of California, Berkeley, 1980.

[34] Bastos Martini, M.R., K. Kanoun, and J. M. De Souza. "Software reliability evaluation of the TROPICO-R switching system," IEEE Transactions on Reliability, Vol. 39, No. 3, 1990.

[35] Beaudry, M. D. "Performance related reliability measures for computing systems," Proceedings International Conference on Fault-Tolerant Computing, 1977.

[36] Belady, L. A. and C. J. Evangelisti. "System partitioning and its measure," Journal System Software, Vol. 2, No. 1, 1982.

[37] Belady, L. A. and M. M. Lehman. "A model of large program development," IBM System Journal, Vol. 3, 1976.

[38] Belli, F. and P. Jedrzejowicz. "An approach to the reliability optimization of software with redundancy," IEEE Transactions on Software Engineering, Vol. SE-17, No. 3, 1991.

[39] Belli, F. and P. Jedrzejowics. "Fault-tolerant programs and their reliability," IEEE Transactions on Reliability, Vol. R-39, No. 2, 1990.

[40] Bendell, T. "The use of exploratory data analysis techniques for software reliability assessment and prediction," NATO Advanced Study Institute, The Challenge of Advanced Technology to System Design Method, 1985.

[41] Bhargava, B. "Software reliability in real-time systems," Proceedings COMPCON, 1981.

[42] Bishop, P. G. and F. D. Pullen, "Error masking: A source of failure dependency in multiversion programs", in Proceedings 1st International working Conference Dependable Computers Critical Application, Santa Barbara, CA, August 1989.

[43] Bittani, S., Ed., Software reliability modelling and identification (Lecture Notes in Computer Science, No. 341). Berlin:Springer, 1987.

[44] Boehm, B. W. Software engineering economics. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[45] Boehm, B. W. and R. W. Wolverton. "Software cost modeling: Some lessons learned," J. System Software, Vol. 1, No. 3, 1980.

[46] Boehm, B. W., J. R. Brown, & M. Lipow. "Quantitative evaluation of software quality," Proceedings Second International Conference on Software Engineering, 1976.

[47] Boehm, B. W. "Software and its impact: a quantitative assessment," Datamation, 19, No. 5.

[48] Bridgewater, K., J. L. Gersting, and D. Robers. "New conditions for N-version programming," 1988 International Hawaii Conference 21st, Software Track.

[49] Brilliant, S. S., J. C. Knight and N. G. Leveson. "Analysis of faults in an N-version software experiment," IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990.

[50] Brilliant, S. S., J. C. Knight and N. G. Leveson. "The consistent comparison problem in N-version software," IEEE Transactions on Software Engineering, Vol. 15, Nov. 1989.

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

[52] Brooks, F. P. Jr. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975.

[53] Brooks, W. D. and R. W. Motley. "Analysis of discrete software reliability models," Technical Report RADC-TR-80-84, 1980, Rome Air Development Center, New York.

[54] Brown, D. E. "A method for obtaining software reliability measures during development," IEEE Transactions on Reliability, Vol. R-36, 1987.

[55] Brown, D. B., S. Maghsoodloo, and W. H. Deason. "A cost model for determining the optimal number of software test case," IEEE Transactions on Software Engineering, Vol. SE-15, No. 2, 1989.

[56] Brown, J. R. and M. Lipow. "Testing for software reliability," in Proceedings International Conference Reliable Software, Los Angeles, CA, April 1975.

[57] Buckley F. J. & R. Poston. "Software quality assurance," IEEE Transactions Software Engineering, SE-10, No. 1, 1984.

[58] Butner, S. E. & R. K. Iyer. "A statistical study of reliability and system load at SLAC," Proceedings International Conference on Fault-Tolerant computing, 1980.

[59] Caruso, J. M. and D. W. Desormeau. "Intergrating prior knowledge with a software reliability growth model," 13th International Conference on Software Engineering, Austin, Texas, May 13-17, 1991.

[60] Caspi, P. A. & E. F. Kouka. "Stopping rules for a debugging process based on different software reliability models," Proceedings International Conference on Fault-Tolerant Computing, 1984.

[61] Castillo, X. & Siewiorek. "A workload dependent software reliability prediction model," Proceedings International Conference on Fault-Tolerant Computing, 1982.

[62] Castillo, X. & Siewiorek. "A performance-reliability model for computing systems," Proceedings International Conference on Fault-Tolerant Computing, 1980.

[63] Catuneanu, V. and A. Mihalache. "Improving the accuracy of the Littlewood-Verrall model," IEEE Transactions Reliability, Vol. R-34, 1985.

[64] Cavano, J. P. "Toward high confidence software," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[65] Chen, L. and A. Avizienis. "N-version programming: A fault tolerance approach to the reliability of software," in Proceedings Eighth International Symposium Fault-Tolerant Computing, Toulouse, France, June 1978.

[66] Chen, E. T. "Program complexity and program productivity," IEEE Transactions on Software Engineering, SE-4, No. 2, 1978.

[67] Chen, Y. Y. and S. J. Upadhyaya, "An analysis of a reconfigurable binary tree architecture based onmultiple-level redundancy", 20th Annual IEEE International Symposium on Fault Tolerant Computing, June 1990.

[68] Chenoweth, H. B. "Modified Musa theoretic software reliability," Proceedings Annual Reliability and Maintainability Symposium, 1981.

[69] Cheung, R. C. "An user-oriented software reliability model," IEEE Transactions on Software Engineering, Vol. SE-6, 1980.

[70] Chiang, C. L. Introduction to Stochastic Processes in Biostatistics. Wiley, New York, 1968.

[71] Chisholm, G. H., J. Kljaich, B. T. Smith, and A. S. Wojcik. "An approach to the verification of a fault-tolerant, computer-based, reactor safety system - A case study utilizing automated reasoning," Electric Power Research Institute Technology Report, Electric Power Research Institute, Palo Alto, CA, 1986.

[72] Cifersky, J. "Generalized Markov model for reliability evaluation of functionally degradable systems," Proceedings International Conference on Fault-Tolerant Computing, 1982.

[73] Coutinho, J. S. "Software reliability growth," Proceedings International Conference on Reliable Software, 1973.

[74] Cristian, F. "Correct and robust programs," IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, 1984.

[75] Crow, L. H. "Methods for assessing reliability growth potential," Proceedings Annual Reliability and Maintainability Symposium, 1984.

[76] Crow, L. H. & N. D. Singpurwalla. "An empirically developed Fourier series model for describing software failure," IEEE Transactions on Reliability, R-33, No. 2, 1984.

[77] Csenki, A. "Bayes predictive analysis of a fundamental software reliability model,", IEEE Transactions on Reliability, Vol. R-39, No. 2, 1990.

[78] Csenki, A., "Recovery block reliability analysis with failure clustering", in Proceedings 1st International Working Conference Dependable Computers Critical Application, Santa Barbara, CA, Aug. 1989.

[79] Culpepper, L. M. "A system for reliable engineering software," IEEE Transactions on Reliability, R-33, No. 2, 1984.

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

[81] Daniels, B. K. "Software Reliability," Reliability Engineering, 4, 1983.

[82] Darringer, J. A. "The application of program verification techniques to hardware verification," in Proceedings 16th Design Automat. Conference, San Diego, CA, June 1979.

[83] Davis, I. J. "Local correction of Helix (k) lists," IEEE Transactions on Computers, Vol. 38, No. 5, May 1989.

[84] Dhillon, B. S. Reliability Engineering in Systems Design and Operation. Van Nostrand Reinhold Co., New York, 1983.

[85] Dhillon, B. S. "Software Reliability - bibliography," Microelectronics and Reliability, Vol. 22, No. 3, 1982.

[86] Dickson, J. C. "Quantitative Analysis of Software Reliability," Proceedings Annual Reliability and Maintainability, 1972.

[87] Donelson, J. III. "Cost model for testing program based on nonhomogeneous Poisson failure model," IEEE Transactions on Reliability, R-26, No. 3, 1977.

[88] Dong-Hae Chi and Way Kuo. "Optimal design for software reliability and development cost," IEEE Transactions on Reliability, R-8, No. 2, 1990.

[89] Downs, T. "Extension to an approach to the modeling of software testing with some performance comparisons," IEEE Transactions on Software Engineering, SE-12, No. 9, 1986.

[90] Downs, T. "An approach to the modeling of software testing with some applications," IEEE Transactions on Software Engineering, SE-11, No. 4. 1985.

[91] Dunham, J. R. "Experiments in software reliability: life-critical applications," IEEE Transactions on Software Engineering, SE-12, No. 1, 1986.

[92] Dunsmore, H. E. & J. D. Gannon. "Experimental investigation of programming complexity," Proceedings ACM/NBS 16th Annual Technical Symposium: Systems and Software, 1977.

[93] Duran, J. W. and J. Wiorkowski. "Quantifying software validation by sampling," IEEE Transactions on Reliability, R-29, No. 2, 1980.

[94] Duvall, L. "Data needs for software reliability modeling," Proceedings Annual Reliability and Maintainability Symposium, 1980.

[95] Eckhardt, D. E. Jr. & L. D. Lee. "A theoretical basis for the analysis of multiversion software subject to coincident errors," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[96] Elliott, R. W. "Measuring computer software reliability," Computer and Industrial Engineering, 2, 1978.

[97] Elshoff, J. L. "An analysis of some commercial PL/1 programs," IEEE Transactions on Software Engineering, SE-12, 1976.

[98] Endres, A. "An analysis of errors and their caused in system programs," IEEE Transactions on Software Engineering, Vol. SE-1, No. 6, 1975.

[99] Fagan, M. E. "Design and code inspections to reduce errors in program development," IBM System J., Vol. 15, No. 3, 1976.

[100] Fagan, M. E. "Advances in software inspections," IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, 1986.

[101] Fitzsimmons, A. and T. Love. "A review and evaluation of software science," ACM Computing Surveys, Vol. 10, No. 1, March 1978.

[102] Fischer, K. F. & M. G. Walker. "Improved software reliability through requirements verification," IEEE Transactions on Reliability, R-28, No. 3, 1979.

[103] Forman, E. H. and N. D. Singpurwalla. "An empirical stopping rule for debugging and testing computer software," Journal American Statistic Association, Vol. 72, 1977.

[104] Forman, E. H. and N. D. Singpurwalla. "Optimal time intervals for testing hypotheses on computer software errors," IEEE Transactions on Reliability, Vol. R-28, 1979.

[105] Fosdick, L. D. and L. J. Osterweil. "Data flow analysis in software reliability," ACM Computing Surveys, Vol. 8, No. 3, September 1976.

[106] Fragola, J. R. and J. F. Spahn. "The software error effects analysis: a quantitative design tool," Proceedings International Conference on Reliable Software, 1973.

[107] Friedman, M. "Modeling the penalty cost of software failure," Proceedings Annual Reliability and Maintainability Symposium, 1987.

[108] Geist, R. and K. Trivedi. "Reliability estimation of fault-tolerant systems: Tools and techniques". IEEE Computers, Vol. 23, No. 7, July 1990.

[109] Gersting, J. L., R. L. Nist, D. B. Roberts, and R. V. Valkenburg. "A comparison of voting Algorithms for N-version programming," Proceedings 24th Annual Hawaii International Conference on System Sciences, Vol. II, January 1991.

[110] Gilb, T. Software Metrics. Winthrop, Cambridge, Mass., 1977.

[111] Girard, E. & J. C. Rault. "A programming technique for software reliability," Proceedings International on Reliable Software, 1973.

[112] Glass, R. L. "Persistent software errors," IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, 1981.

[113] Goel, A. L. "A summary of the discussion on an analysis of computing software reliability models," IEEE Transactions on Software Engineering, SE-6, No. 5, 1980.

[114] Goel, A. L. "Software reliability models: Assumptions, limitations, and applicability," IEEE Transactions on Software Engineering, Vol. SE-2, No. 12, 1985.

[115] Goel, A. L. and K. Okumoto. "When to stop testing and start using software?" Performance Evaluation Review, 1, No. 10, 1981.

[116] Goel, A. L. and K. Okumoto. "A Markovian model for reliability and other performance measures of software systems," Proceedings COMPCON, 1979.

[117] Goel, A. L. and K. Okumoto. "An analysis of recurrent software errors in a real-time control system," Proceedings ACM Conference, 1978.

[118] Goel, A. L. and J. Soenjoto. "Models for hardware-software system operational performance evaluation," IEEE Transactions on Reliability, R-30, No. 3, 1981.

[119] Goel A. L. and K. Okumoto. "Time-dependent error-detection rate model for software and other performance measures," IEEE Transactions on Reliability, Vol. R-28, No. 3, 1979.

[120] Govil, K. K. "Philosophy of a new structure of software reliability modeling," Microelectronics and Reliability, 24, No. 3, 1984.

[121] Greenspan, S. J. & C. L. McGowan. "Structuring software development for reliability," Microelectronics and Reliability, 17, No. 1, 1978.

[122] Gubitz, M. & K. O. Ott. "Quantifying software reliability by a probabilistic model," Reliability Engineering, 5, 1983.

[123] Halstead, M. H. Elements of Software Science. Elsevier, New York, 1977.

[124] Hamilton, P. A. & J. D. Musa. "Measuring reliability of computation center software," Proceedings 3rd International Conference on Software Engineering, 1978.

[125] Haney, F. M. "Module connection analysis - a tool for scheduling software debugging activities," Proceedings AFIPS Conference, 41, part I, 1972.

[126] Hansen, G. A. "Measuring software reliability," Mini-Micro System, Aug. 1977.

[127] Hart, G. "The software integrity of a computer system installed in Royal Navy Frigate," Microelectronics and Reliability, 22, No. 6, 1982.

[128] Haynes, R. D. and W. E. Thompson. "Hardware and software reliability and confidence limits for computer controlled systems," Microelectronics and Reliability, 20, No. 1, 1980.

[129] Haynes, R. D. and W. E. Thompson. "Combined hardware and software availability," Proceedings Annual Reliability and Maintainability Symposium, 1981.

[130] Hecht, H. "Mini-tutorial on software reliability," Proceedings COMPSAC, 1980.

[131] Hecht, H. "Can software benefit from hardware experience?" Proceedings Annual Reliability and Maintainability Symposium, 1975.

[132] Hecht, H. "Fault-tolerant software," IEEE Transactions on Reliability, R-28, No. 3, 1979.

[133] Hecht, H. and M. Hecht. "Software reliability in system context," IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, 1986.

[134] Hellerman, L. "A measure of computational work," IEEE Transactions on Computer, C-21, No. 5, 1972.

[135] Heninger, K. L. "Specifying software requirements for complex systems: New techniques and their applications," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, 1980.

[136] Henry, S. and D. Kafura. "Software quality metrics based on interconnectivity," Journal of System and Software, Vol. 2, No. 2, 1981.

[137] Howden, W. E. "Empirical studies of software validation," Microelectronics and Reliability, 19, No. 1, 1979.

[138] Hua, K. A. and J. A. Abraham. "Design of systems with concurrent error detection using software redundancy," Joint Fault Tolerant Computer Conference, 1986.

[139] Huang, X. Z. "The hypergeometric distribution model for predicting the reliability of software," Microelectronics and Reliability, 24, No. 1, 1984.

[140] Humphrey, W. S. and N. D. Singpurwalla. "Predicting (Individual) software productivity," IEEE Transactions on Software Engineering, Vol. SE-17, No. 2, 1991.

[141] Humphrey, W.S., Managing the software process, Reading, MA: Addison-Wesley, 1989.

[142] Iannino, A. "Criteria for software reliability model comparison," IEEE Transactions on Software Engineering, SE-10, No. 6, 1984.

[143] IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 729, 1983.

[144] Islam, M. M. and F. Lombardi. "Estimation of total errors in software," Microelectronics and Reliability, 22, No. 2, 1982.

[145] Iyer, R. K. and D. J. Rossetti. "Effect of system workload on operating system reliability: a study on IBM 3081," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[146] Iyer, R. K. and P. Velardi. "Hardware-related software errors: measurement and analysis," IEEE Transactions on Software Engineering, SE-11, No. 2, 1985.

[147] Jacoby, R. and Y. Tohma. "The hyper-geometric distribution software reliability growth model (HGDM): Precise formulation and applicability," in Proceedings COMPSAC-90, Sept. 1990.

[148] Jacoby, R. and Y. Tohma. "Parameter value computation by least square method and evaluation of software availability and reliability at service-operation by the hyper-geometric distribution software reliability growth model," 13th International Conference on Software Engineering, Austin, Texas, May 13-17, 1991.

[149] Jaffe, M. S., N. G. Leveson, M. P. E. Heimdahl, and B. E. Melhart. "Software requirements analysis for real-time Process-control systems", IEEE Transactions on Software Engineering, Vol. SE-17, No. 3, March 1991.

[150] Jelinski, Z. and P. B. Moranda. "Software reliability research," in Statistical Computer Performance Evaluation, W. Freiberger, Ed. New York: Academic, 1972.

[151] Jewell, W. S. " Bayesian extensions to a basic model of software reliability," IEEE Transactions on Software Engineering, Vol. SE-11, 1985.

[152] Joe, H. and N. Reid. "On the software reliability models of Jelinski-Moranda and Littlewood," IEEE Transactions on Reliability, R-34, No. 3, 1985.

[153] Jones, T. C. "Measuring programming quality and productivity," IBM Systems Journal, 17, No. 1, 1978.

[154] Kaaniche, M., K. Kanoun, and S. Metge. "Role of the hyperexponential model in the software validation process of a telecommunication equipment," in Proceedings 7th International Conference Reliability and Maintainability, Brest, France, June 1990.

[155] Kanoun, K., M. R. Martini, and J. M. De Souza. "A method for software reliability analysis and prediction application to the TROPICO-R switching system," IEEE Transactions on Software Engineering, Vol. SE-17, No. 4, 1991.

[156] Kanoun, K., J. C. Laprie, and T. Sabourin. "A method for software reliability growth analysis and assessment," in Proceedings Le Genie Logiciel et Ses Applications, Toulouse, France, December 5-9, 1988.

[157] Kanoun, K. and T. Sabourin. "Software dependability of a telephone switching system," in Proceedings 17th IEEE International Symposium Fault Tolerant Computing, Pittsburgh, PA, June 1987.

[158] Kanoun, K., J. C. Laprie, and T. Sabourin. "A method for software reliability growth analysis and assessment," in Proceedings 1st International Workshop Software Engineering and Its Applications, Toulouse, France, December 1988.

[159] Kapur, P. K. and R. B. Garg. "Optimum release policy for an inflection s-shaped software reliability growth model," Microelectronics and Reliability, Vol. 31, No. 1, 1991.

[160] Kareer, N., P. K. Kapur and P. S. Grover. "An S-shaped software reliability growth model with two types of errors," Microelectronics and Reliability, Vol. 30, No. 6, 1990.

[161] Keller, R. K., M. Cameron, R. N. Taylor, and D. B. Troup. "Chiron-1: A user interface development system tailored to software environments," Proceedings 24th Annual Hawaii International Conference System Sciences, Hawaii, 1991.

[162] Kelly, J., D. Eckhardt, M. Vouk, D. McAllister, and A. Caglayan. "A large scale second generation experiment in multi-version software: Description and early results," in Digest Papers, FTC5-18, Tokyo, Japan, 1988.

[163] Kenneth, R. and M. Pollak. "A semi-parametric approach to testing for reliability growth, with application to software systems," IEEE Transactions on Reliability, R-35, No. 3, 1986.

[164] Kim, K. H. "An approach to experimental evaluation of real time fault tolerant distributed computing schemes," IEEE Transactions on Software Engineering, Vol. SE-15, No. 6, June 1989.

[165] Kim, K. H. and H. O. Welch. "Distributed execution of recovery blocks: An approach for Uniform Treatment of Hardware and software faults in real time applications," IEEE Transactions on Computers, Vol. C-38, No. 5, May 1989.

[166] Kim, K, and J. Yoon. "Approaches to implementation of a reparable distributed recovery block scheme," in Digest Papers, FTCS-18, 1988, Tokyo, Japan.

[167] Kline, M. B. "Software and hardware R&M: what are the differences?" Proceedings Annual Reliability and Maintainability Symposium, 1980.

[168] Kljaich, J., B. T. Smith, and A. S. Wojcik. "Formal verification of fault tolerance using theorem-proving techniques," IEEE Trans Computers, Vol. C-38, No. 3, March 1989.

[169] Knight, J. C. and N. G. Leveson. "An experimental evaluation of the assumption of independence in multiversion programming," IEEE Transactions on Software Engineering, SE-12, January 1986.

[170] Koch, H. S. and P. Kubat. "Optimal release time of computer software," IEEE Transactions on Software Engineering, Vol. SE-9, 1983.

[171] Kock, H. S. and P. J. C. Spreij. "Software reliability as an application of martingale and filtering theory," IEEE Transactions on Reliability, Vol. R-32, 1983.

[172] Koch, H. S. and P. Kubat. "Quick and simple procedures to assess software reliability and facilitate project management," Journal of Systems and Software, 1981.

[173] Kopetz, H. Software Reliability. MacMillan Press, Ltd., London, 1979.

[174] Korel, B. and J. Laski. "Algorithmic software fault localization," Proceedings 24th Hawaii International Conference System Sciences, Hawaii, Vol. II.

[175] Koren, I. and A. D. Singh. "Fault tolerance in VLSI circuits". IEEE Computers, Vol. 23, No. 7, Jully 1990.

[176] Kremer, W. "Birth-death and bug counting," IEEE Transactions on Reliability, Vol. R-32, 1983.

[177] Kubat, P. "Assessing reliability of modular software." Operation Research Letters, Vol. 8, 1989.

[178] Kubat, P. and H. S. Koch. "Managing test-procedures to achieve reliable software," IEEE Transactions on Reliability, Vol. R-32, 1983.

[179] Kubat, P. and H. S. Koch. "Pragmatic testing protocols to measure software reliability,", IEEE Transactions on Reliability, Vol. R-32.

[180] Kuo, W. "On optimal burn-in modeling and its application to an electronic product," Proceedings 3rd International Conference on Reliability and Maintainability, France, 1982.

[181] Kuo, W. "Software reliability estimation: a realization of competing risk," Microelectronics and Reliability, 23, No. 2, 1983.

[182] Landrault, C. and J. C. Laprie. "Reliability and Availability modeling of systems featuring hardware and software faults," Proceedings International Conference on Fault-tolerant Computing, 1977.

[183] Langberg, N. and N. D. Singpurwalla. "A unification of some software reliability models," SIAM Journal of Statistic Computing, Vol. 6, No. 3, 1985.

[184] Laprie, J. C., J. Arlat, C. Beounes, and K. Kanoun. "Definition and analysis of hardware- and software-fault tolerant architectures," IEEE Computers, Vol. 23, No. 7, July 1990.

[185] Laprie, J. C., J. Arlat, C. Beounes, K. Kanoun, and C. Hourtolle. "Hardware-and software-fault tolerance: Definition and analysis of architectural solutions," in Proceedings FTCS-17, Pittsburgh, PA, July 1987.

[186] Laprie, J. C. "Dependability modeling and evaluation of hardware and software systems," in Proceedings 2nd GI/NTG/GMR Conference Fault Tolerant Computing, Bonn, Germany, September 1984.

[187] Laprie, J. C. "Dependability evaluation of software systems in operation," IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, 1984.

[188] Lawless, J. F. Statistical Models and Methods for Lifetime Data, John Wiley & Sons, New York, 1982.

[189] Leu, S. W., E. B. Fernandez, and T. Khoshgoftaar. "Fault-tolerant software reliability modeling using petri nets," Microelectronics and Reliability, Vol. 31, No. 4, 1991.

[190] Levendel, Y. "Reliability analysis of large software systems: Defect data modeling," IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990.

[191] Levendel, Y. "Defects and reliability analysis of large software systems: Field experience," in Proceedings 19th IEEE International Symposium Fault Tolerant Computing, Chicago, IL, June 1989.

[192] Leveson, N. G., S. S. Cha, J. C. Knight, and T. J. Shimeall. "The use of self checks and voting in software error detection: An Empirical study," IEEE Transactions on Software Engineering, Vol. 16, No. 4, April 1990.

[193] Leveson, N. G. and P. R. Harvey. "Analyzing software safety," IEEE Transactions on Software Engineering, SE-9, No. 5, 1983.

[194] Levy, L. S. "A metaprogramming method and its economic justification," IEEE Transactions on Software Engineering, SE-12, No. 2, 1986.

[195] Lipow, M. "On software reliability," IEEE Transactions on Reliability, Vol. R-28, No. 3, 1979.

[196] Lipow, M. and E. Book. "Implications of R&M 2000 on software," IEEE Transactions on Reliability, Vol. R-36, No. 3, 1987.

[197] Lipow, M. "Number of faults per line of code," IEEE Transactions on Software Engineering, SE-8, No. 4, 1982.

[198] Lipow, M. "Prediction of software failures," Journal of Systems and Software, 1, No. 1, 1979.

[199] Lipow, M. and T. A. Thayer. "Prediction of software failures," Proceedings Annual Reliability and Maintainability Symposium, 1977.

[200] Lipow, M. "Estimation of software residual errors," TRW Software Series, Report TRW-SS-72-09, Redondo Beach, CA, 1972.

[201] Lipow, M. "Some variation of a model for software time-to-failure," TRW Systems Group, Correspondence, ML-74-2260, August 1974.

[202] Littlewood B. and D. R. Miller. "A conceptual model of multi-version software," in Proceedings FTCS-17, Pittsburgh, PA, July 1987.

[203] Littlewood, B. and A. Sofer. "A bayesian modification to the Jelinski-Moranda software reliability model," Software Engineering J., Vol. 2, 1987.

[204] Littlewood, B. and P. A. Keiller. Adaptive Software Reliability Modeling. IEEE 0731-3071/84.

[205] Littlewood, B. "A critique of the Jelinski-Moranda model for software reliability," Proceedings Annual Reliability and Maintainability Symposium, 1981.

[206] Littlewood, B. "Stochastic reliability growth: A model for fault-removal in computer programs and hardware designs," IEEE Transactions on Reliability, Vol. R-30, No. 4, 1981.

[207] Littlewood, B. and J. Verral. "Likelihood function of a debugging model for computer software reliability," IEEE Transactions on Reliability, Vol. R-30, No. 2, 1980.

[208] Littlewood, B. "Theories of software reliability: How good are they and how can they be improved?," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, 1980.

[209] Littlewood, B. "A Bayesian differential debugging model for software reliability," Proceedings COMPSAC, 1980.

[210] Littlewood, B. "What makes a reliable program - few bugs or a small failure rate?" Proceedings COMPCON, 1980.

[211] Littlewood, B. "The Littlewood-Verrall model for software reliability compared with some rivals," Journal of Systems and Software, 1980.

[212] Littlewood, B. "How to measure software reliability and how not to," IEEE Transactions on Reliability, Vol. R-28, No. 2, 1979.

[213] Littlewood, B. "Software reliability model for modular program structure," IEEE Transactions on Reliability, Vol. R-28, No. 3, 1979.

[214] Littlewood, B. "Validation of a software reliability model," Proceedings Software Life Cycle Management Workshop, 1978.

[215] Littlewood, B. "A semi-Markov model for software reliability with failure cost," Proceedings Symposium on Computer Software Engineering, 1976.

[216] Littlewood, B. "A reliable model for systems with Markov structure," Applied Statistics, 24, No. 2, 1975.

[217] Littlewood, B. and J. L. Verral. "A bayesian reliability model with a stochastically monotone failure rate," IEEE Transactions on Reliability, 1974.

[218] Littlewood, B. and J. L. Verral. "A bayesian reliability growth model for computer software," Journal Royal Statistical Society, Series C, Vol. 22, 1973.

[219] Lloyd, D. K. and M. Lipow. Reliability Management, Methods, and Mathematics. Lloyd, Redondo Beach, CA., 1977.

[220] Lorczak, P. R., A. K. Caglayan, and D. E. Eckharatt. "A theoretical investigation of generalized voters for redundant systems. Proceedings FTCS-19, Chicago, IL, June 1989.

[221] Lyu, M. R. and A. Avizienis. "Assuring design diversity in N-version software: a design paradigm for N-version programming," Proceedings second International Working Conference on Dependable Computing for Critical Applications, 1991.

[222] Makam, S. V. and A. Avizienis. "ARIES 81: a reliability and life cycle evaluation tool for fault-tolerant systems," Proceedings International Conference on Fault-Tolerant Computing, 1982.

[223] Malaiya, Y. K. and P. Verma. "Detectability profile approach to software reliability," Advances in Reliability and Quality Control, M.H. Hamza, Ed., 1989.

[224] Mallory, S. R. "A hybrid software development model for medical instruments," Software Development, 1990.

[225] Martini, M. R. B., K. Kanoun, and J. M. De Souza. "Software reliability evaluation of the TROPICO_R switching system," IEEE Transactions on Reliability, Vol. 39, No. 3, 1990.

[226] Masuda, Y., N. Miyawaki, U. Sumita, S, Yokoyama. "A statistical approach for determining release time of software system with modular structure," IEEE Transactions on Reliability, Vol. R-38, No. 3, 1989.

[227] Mazzuchi, T. A. and R. Soyer. "A bayes empirical-bayes model for software reliability," IEEE Transactions on Reliability, Vol. R-37, No. 2, 1988.

[228] McCabe, T. J. "A complexity measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, 1976.

[229] McCall, J. A., P. Pierce, R. Hartley, and R. Thonerfelt. "Measuring technology for software life cycle support," COMPCON, 1985.

[230] McDonald, W. C., and R. W. Smith. "A flexible distributed testbed for real time applications," IEEE Transactions on Computers, Vol. 15, 1982.

[231] Meirnold R. J. and N. D. Singpurwalla. "Bayesian analysis of a commonly used model for describing software failures," The Statistician, Vol. 32, 1983.

[232] Meyer, F. J. and D. K. Pradhan. "Dynamic testing strategy for distributed systems," IEEE Transactions on Computers, C-38, No. 3, 1989.

[233] Migneault, G. E. "Software reliability and advanced avionics," Proceedings COMPCON, 1980.

[234] Miller, D. R. "Exponential order statistic models of software reliability growth," IEEE Transactions on software Engineering, Vol. SE-12, No. 1, 1986.

[235] Miller, D. R. and A. Sofer. "Completely monotone regression estimation of software failure rate," Proceedings International Conference on Software Engineering, 1985.

[236] Mills, H. D. "On the statistical validation of computer programs," IBM FSD, unpublished paper, July 1970.

[237] Mills, H. D. "On the development of large reliable software," in Rec. 1973 IEEE Symposium Computer Software Reliability, New York, April 30-May 2, 1973.

[238] Misra, P. N. "Software reliability analysis," IBM Systems Journal, 22, No. 3, 1983.

[239] Miyamoto, I. "Reliability evaluation and management for an entire software life cycle," Software Life Cycle Management Workshop, 1978.

[240] Moawad, R. "Comparison of current software reliability models," Proceedings International Conference on Software Engineering, 1984.

[241] Mohanty, S. N. "Models and measurements for quality assessment of software," Computing Surveys, 11, No. 3, 1979.

[242] Moranda, P. B. "An error detection model for application during software development," IEEE Transactions on Reliability, R-28, No. 5, 1979.

[243] Moranda, P. B. "A comparison of software error-rate models," Proceedings Texas Conference on Computing Systems, 1975.

[244] Moranda, P.B. "Prediction of software reliability during debugging," Proceedings Annual Reliability & Maintainability Symposium, 1975.

[245] Morey, R. C. "Estimating and improving the quality of information in a MIS," Communications of the ACM, 25, No. 5, 1982.

[246] Mourad, S. and D. Andrews. "The reliability of the IBM MVS/XA operating system," Proceedings International Conference on Fault-Tolerant Computing, 1985.

[247] Musa, J. D. and A. F. Ackerman. "Quantifying software validation: When to stop testing?" IEEE Software, May 1989.

[248] Musa, J. D., et al., Software reliability measurement, prediction, application. New York: McGraw-Hill International, 1987.

[249] Musa, J. D. "Software quality and reliability basics," in Proceedings 1987 Fall Joint Conference IEEE Exploring Technology Today and Tomorrow, October 25-29, 1987, Dallas, TX, 1987.

[250] Musa, J. D. and K. Okumoto. "Application of basic and logarithmic Poisson execution model in software reliability measure," NATO Advanced Study Institute, The Challenge of Advanced Computing Technology to System Design Method, 1985.

[251] Musa, J. D. and K. Okumoto. "A logarithmic poisson execution time model for software reliability measurement", Proceedings 7th International Conference on Software Engineering, 1984.

[252] Musa, J. D. and K. Okumoto. "A comparison of time domains for software reliability models," Journal of Systems and Software, 4, No. 4, 1984.

[253] Musa, J. D. and K. Okumoto. "Software reliability models: concepts, classification, comparisons and practice", Electronic Systems Effectiveness and Life Cycle Costing, Ed. J.K. Skwirzynski, Springer-Verlag, 1982.

[254] Musa, J. D. "The measurement and management of software reliability," Proceedings IEEE, Vol. 68, 1980.

[255] Musa, J. D. "Software reliability measurement," Journal of Systems and Software, 1, No. 3, 1980.

[256] Musa, J. D. "Software reliability measures applied to system engineering," Proceedings COMPCON, 1979.

[257] Musa, J. D. "Validation of execution time theory of software reliability," IEEE Transactions on Reliability, R-28, No. 3, 1979.

[258] Musa, J. D. "The use of software reliability measures in project management," Proceedings COMPSAC, 1978.

[259] Musa, J. D. "A theory of software reliability and its applications," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, 1975.

[260] Myers, G. J. Software reliability: Principles and Practices, John Wiley & Sons, 1976.

[261] Nathan, I. "A deterministic model to predict error-free status of complex software development," Workshop on Quantitative Software Models, 1979.

[262] Nayak, T. K. "Software reliability statistical modeling and estimation," IEEE Transactions on Reliability, R-35, No. 5, 1986.

[263] Nayak, T. K. "Estimating population size by recapture sampling," Report Department of Statistics, George Washington University, 1986.

[264] Nelson, E. C. A statistical basis for software reliability assessment. TRW-SS-73-03, TRW, 1973.

[265] Nelson, E. C. "Estimating software reliability from test data," Microelectronics Reliability, Vol. 17, No. 1, 1974.

[266] Nelson, V. P. "Fault-tolerant computing: fundamental concepts," IEEE Computers, Vol. 23, No. 7, July 1990.

[267] Nicola, V. F and A. Goyal "Modeling of correlated failures and community error recovery in multiversion software", IEEE Transactions on Software Engineering, Vol. 16, No. 3, 1990.

[268] Ohba, M. "Software reliability analysis models," IBM Journal Research Development, Vol. 21, No. 4, 1984.

[269] Ohba, M. and S. Yamada. "S-shaped software reliability growth models," in Proceedings 4th International Conference Reliability and Maintainability, Perros Guirec, France, 1984.

[270] Ohtera, H. and S. Yamada. "Optimal allocation and control problems for software-testing resources," IEEE Transactions on Reliability, Vol. R-39, No. 2, 1990.

[271] Ohtera, H. and S. Yamada. "Optimum software-release time considering an error-detection phenomenon during operation," IEEE Transactions on Reliability, Vol. 39, No. 5, 1990.

[272] Okumoto, K. and A. L. Goel. "Optimum release time for software systems, based on reliability and cost criteria," Journal Systems and Software, Vol. 1, 1980.

[273] Okumoto, K. "A statistical method for software quality control," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[274] Pham, H. "On the optimal design of fault tolerant N-version programming software systems subject to constraints", Submitted to IEEE Transactions on Computers.

[275] Pham, H. Optimal designs of systems with competing failure modes. Ph.D. Dissertation, State University of New York at Buffalo, New York.

[276] Pham, H. "Optimal cost-effective design of hybrid hardware redundant systems", International Journal Systems Science, Vol. 22, No. 3, 1991.

[277] Pham, H. and S. J. Upadhyaya. "Reliability analysis of a class of fault tolerant systems " IEEE Transactions on Reliability, Vol. 38, No. 3, August 1989.

[278] Podgurski, A. and L. A. Clarke. "A formal model of program dependencies and its implications for software testing, debugging, and maintenance,", IEEE Transactions on Software Engineering, Vol. 16, No. 9, 1990.

[279] Potier, D. "Experiments with computer software complexity and reliability," Proceedings International Conference on Software Engineering, 1982.

[280] Pressman, R. S. Software engineering, A practitioner's approach. Addison-Wesley, 1983.

[281] Purtilo, J. M. and P. Jalote. "An Environment for developing fault tolerant Software," IEEE Transactions on Software Engineering, Vol. 17, No. 2, February 1991.

[282] Raftery, A. E. "Analysis of a simple debugging model," Journal Royal Statistical Society, Series C, Vol 37, 1988.

[283] Ramamoorthy, C. V. and F. B. Bastani. "Modeling of the software reliability growth process," Proceedings COMPSAC, 1980.

[284] Ramamoorthy, C. V. and S. F. Ho. "Testing large software with automated software evaluation systems," IEEE Transactions on Software Engineering, SE-1, No.1, 1975.

[285] Ramamoorthy, C. V and F. B. Bastani. "Software reliability status and perspective," IEEE Transactions on Software Engineering, Vol. SE-8, 1982.

[286] Ramamoorthy, C. V., Y. K. Mok, E. B. Bastani, G. H. Chin, and K. Suzuki. "Application of a methodology for the development and validation of reliable process control software," IEEE Transactions on Software Engineering, Vol. SE-7, No. 6, 1981.

[287] Ramamoorthy, C. V., A. Prakash, W. Tsai, and Y. Usuda. "Software engineering: problems and perspectives," Computer, Vol. 17, No. 10, 1984.

[288] Randell, B. "System structure for software fault tolerance," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, 1975.

[289] Ramzan, M. T. "Seeded bug volume for software validation," Microelectronics and Reliability, 23, No. 5, 1983.

[290] Reifer, R. J. "Software failure modes and effects analysis," IEEE Transactions on Reliability, R-28, No. 3, 1979.

[291] Reiss, R. M. "A prediction experience with three software reliability models," Workshop on Quantitative Software Models, 1979.

[292] Roca, J. L. "Normal approach on correctness software estimation," Microelectronics and Reliability, Vol. 27, No. 3, 1987.

[293] Romeu, J. L. and K. A. Dey. "Classifying combined hardware/software R models," Proceedings Annual Reliability and maintainability Symposium, 1984.

[294] Ronback, J. A. "Software reliability - how it affects system reliability," Microelectronics and Reliability, 14, No. 2.

[295] Rosene, A. F., J. E. Connolly, and K. M. Bracy. "Software maintainability - what it means and how to achieve it," IEEE Transactions on Reliability, 30, No. 3, 1981.

[296] Ross, S. M. "Software reliability: The stopping rule problem," IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, 1985.

[297] Ross, S. M. "Statistical estimation of software reliability," IEEE Transactions on Software Engineering, SE-11, No. 5, 1985.

[298] Rossetti, D. J. and R. K. Iyer. "Software related failures on the IBM 3081: a relationship with system utilization," CH1810-IEEE, 1982.

[299] Rubey, R. J. "Planning for software reliability," Proceedings Annual Reliability & Maintainability Symposium, 1977.

[300] Rubey, R. J., J. A. Dana, and P. W. Biche. "Quantitative aspect of software validation," IEEE Transactions on Software Engineering, SE-1, No. 2, 1975.

[301] Rutledge, R. A. "The reliability of memory subject to hard and soft failures," Proceedings International Conference on Fault-Tolerant Computing, 1980.

[302] Saglietti, F. and W. Ehrenberger. "Software diversity - some considerations about its benefits and its limitations," Proceedings SAFECOMP '86, Sarlat, France, Oct. 1986.

[303] Sandoh, H. "Reliability demonstration testing for software," IEEE Transactions on Reliability, Vol. R-40, No. 1, 1991.

[304] Schick, G. J. and C. Y. Lin. "Use of a subjective prior distribution for the reliability of computer software," Journal of Systems and Software, 1, No. 3, 1980.

[305] Schick, G. J. and R. W. Wolverton. "Achieving reliability in large software system," Proceedings Annual Reliability & Maintainability Symposium, 1974.

[306] Schick, G. J. and R. W. Wolverton. "An analysis of competing software reliability models," IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, 1978.

[307] Schneider, V. "Some experimental estimators for developmental and delivered errors in software development projects," Proceedings COMPSAC, 1980.

[308] Schneidewind, N. F. "The use of simulation in the evaluation of software," Computer, 10, No. 4, 1977.

[309] Schneidewind, N. "Application of program graphs and complexity analysis to software development and testing," IEEE Transactions on Reliability, Vol. R-28, No. 3, 1979.

[310] Schutt, D. "On a hypergraph oriented measure for applied computer science," Proceedings COMPCON, 1977.

[311] Scholz, F. W. "Software reliability modeling and analysis," IEEE Transactions on Software Engineering, SE-12, No. 1, 1986.

[312] Scott, K., J. W. Gault, D. F. McAllister. "Fault tolerant software reliability," IEEE Transactions on Software Engineering, Vol. SE-13, May 1987.

[313] Scott. R. K. "Experimental validation of six fault-tolerant software reliability models," Proceedings International Conference on Fault-Tolerant Computing, 1984.

[314] Selby, R. W. "Empirically based analysis of failures in software systems," IEEE Transactions on Reliability, Vol. 39, No. 4, 1990.

[315] Selby, R. W. and V. R. Basili. "Analyzing error-prone system structure," IEEE Transactions on Software Engineering, Vol. 17, No. 2, 1991.

[316] Selby, R. W. Evaluations of software technologies: Testing, clean-room, and metrics. Ph.D. dissertation, Department Computer Science, University Maryland, College Park, Tech. Rep. TR-1500, 1985.

[317] Serra, A. and R. E. Barlow. ed. Theory of Reliability. North-Holland, Amsterdam, 1986.

[318] Shannon, C. and W. Weaver. The Mathematical Theory of Communication. University of Illinois Press, Urbana, 1975.

[319] Shanthikumar, J. G. "A general software reliability model for performance prediction," Microelectronics and Reliability, 21, No. 5, 1981.

[320] Shanthikumar, J. G. "A state and time-dependent error occurrence-rate software reliability model with imperfect debugging," Proceedings COMPCON, 1981.

[321] Shanthikumar, J. G. "Software reliability models: A review," Microelectronics and Reliability, Vol. 23, 1983.

[322] Shanthikumar, J. G. and S. Tafekci. "Optimal software release time using generalized decision trees," Proceedings Fourteenth Annual Hawaii International Conference System Science, 1981.

[323] Shanthikumar, J. G. and S. Tafekci. "Application of a software reliability model to decide software release time," Microelectronics and Reliability, Vol. 23, 1983.

[324] Shen, V. Y. "Identifying error-prone software - an empirical study," IEEE Transactions on Software Engineering, SE-11, No. 4, 1985.

[325] Shen, V. Y., S. D. Conte, and H. E. Hunsmore. "Software science revisited: a critical analysis of the theory and its empirical support," IEEE Transactions on Software Engineering, SE-9, No. 2, 1983.

[326] Sherer, S. A. and E. K. Clemons. "Software risk assessment," in AFIPS Conference Proceedings, Vol. 56, Chicago, IL, June 1987.

[327] Shimeall, T. J. and N. G. Leveson. "An empirical comparison of software fault tolerance and fault elimination," IEEE Transactions on Software Engineering, Feb. 1991, Vol. 17, No. 2.

[328] Shooman, M. L. "Structure models for software reliability prediction," Proceedings International Conference on Software Engineering, 1984.

[329] Shooman, M. L. "Software reliability - analysis and prediction," Integrity in Electronic Flight Control Systems, France, AGARD AG224, 1977.

[330] Shooman, M. L. "Software reliability: measurement and models," Proceedings Annual Reliability & Maintainability Symposium, 1975.

[331] Shooman, M. L. and A. K. Trivedi. "A many-state Markov model for computer software performance parameters," IEEE Transactions on Reliability, R-25, No. 2, 1976.

[332] Shooman, M. L., Software Engineering, New York:McGraw-Hill, 1983.

[333] Shooman, M. L., Software Engineering: Design, Reliability, Management, New York: McGraw-Hill, 1985.

[334] Shooman, M. L. "Operational testing and software reliability estimation during program development," Proceedings International Symposium Computer Software Reliability, 1973.

[335] Shooman, M. L. "Probabilistic models for software reliability prediction," in Statistical Computer Performance Evaluation, W. Freidberger, Ed., New York:Academic, 1972.

[336] Siewiorek, D. P. "Fault tolerance in commercial computers," IEEE Computers, Vol. 23, No. 7, July 1990.

[337] Simkins, D. J. "Software performance modeling and management," IEEE Transactions on Reliability, R-32, No. 3, 1983.

[338] Singpurwalla, N. D. and R. Soyer. "Assessing software reliability growth using a random coefficient autoregressive process and its ramifications," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[339] Singpurwalla, N. D. "Determining an optimal time interval for testing and debugging software," IEEE Transactions on Software Engineering, Vol. SE-17, No. 4, 1991.

[340] Soi, I. M. and K. Gopal. "Hardware vs. software reliability - a comparative study," Microelectronics and Reliability, 20, 1980.

[341] Soi, I. M. "Software complexity: an aid to software maintainability," Microelectronics and Reliability, 25, No. 2, 1985.

[342] Soi, I. M. and K. Gopal. "Some aspects of reliable software packages," Microelectronics and Reliability, 19, 1979.

[343] Soi, I. M. and K. Gopal. "Error prediction in softwaie," Microelectronics and Reliability, 18, 1978.

[344] Soi, I. M. and K. Gopal. "Detection and diagnosis of software malfunctions," Microelectronics and Reliability, 18, 1978.

[345] Spreij, P. "Parameter estimation for a specific software reliability model," IEEE Transactions on Reliability, Vol. R-34, 1985.

[346] Stark, G. E. "Dependability evaluation of integrated hardware/software systems," IEEE Transactions on Reliability, R-36, No. 4, 1987.

[347] Strandberg, K. and H. Anderson. "On a model for software reliability performance," Microelectronics and Reliability, 22, No. 2, 1982.

[348] Strong, E. J. "Software reliability and maintainability in large scale systems," Proceedings COMPSAC, 1978.

[349] Sugiura, N. "On the software reliability," Microelectronics and Reliability, 13, 1974.

[350] Sukert, A. N. "Empirical validation of three software error prediction models," IEEE Transactions on Reliability, Vol. R-28, 1979.

[351] Sukert, A. N. "A guidebook for software reliability assessment," Proceedings Annual Reliability & Maintainability Symposium, 1980.

[352] Sukert, A. N. "A four-project empirical study of software error prediction models," Proceedings COMPSAC, 1978.

[353] Sukert, A. N. "Analysis of software error model predictions and questions of data availability," Software Life Cycle Management Workshop, 1978.

[354] Sukert, A. N. "An investigation of software reliability models," Proceedings Annual Reliability & Maintainability Symposium, 1977.

[355] Sumita, U. and J. G. Shanthikumar. "A software reliability model with multiple-error introduction and removal," IEEE Transactions on Reliability, Vol. R-35.

[356] Sumita, U. and Y. Masuda. "Analysis of software availability/reliability under the influence of hardware failures," IEEE Transactions on Software Engineering, SE-12, No. 1, 1986.

[357] Suri, P. K. and K. K. Aggarwal. "Software reliability of programs with network structure," Microelectronics and Reliability, 21, No. 2, 1981.

[358] Suri, P. K. and K. K. Aggarwal. "Reliability evaluation of computer programs,"Microelectronics and Reliability, 20, 1980.

[359] Swearingen, D. and J. Donahoo. "Quantitative software reliability models-data parameters: a tutorial," Workshop on Quantitative Software Models, 1979.

[360] Szabo, S. G. "A schema for producing reliable software," Proceedings International Conference on Fault-Tolerant Computing, 1980.

[361] Takahashi, M. and Y. Kamayachi. "An empirical study of a model for program error prediction," Proceedings International Conference on Software Engineering, 1985.

[362] Taylor, D. J. "Redundancy in data structure: improving software fault-tolerance," IEEE Transactions on Software Engineering, 6, No. 6, 1980.

[363] Thayer, T. A., M. Lipow, and E. C. Nelson. Software Reliability. North-Holland Publishing Co., Amsterdam, 1978.

[364] Thompson, W. E. and P. O. Chelson. "Software reliability testing for embedded computer systems," Workshop on Quantitative Software Models, 1979.

[365] Thompson, W. E. and P. O. Chelson. "On the specification and testing of software reliability," Proceedings Annual Reliability & Maintainability Symposium, 1980.

[366] Tohma, Y., K. Tokunaga, S. Nagase, and Y. Murata. "Structural approach to the estimation of the number of residual software faults based on the hypergeometric distribution," IEEE Transactions on Software Engineering, Vol. 15, 1989.

[367] Tohma, Y., H. Yamano, M. Ohba, and R. Jacoby. "The estimation of parameters of the hypergeometric distribution and its application to the software reliability growth model," IEEE Transactions on Software Engineering, Vol. SE-17, No. 5, 1991.

[368] Trachtenberg, M. "A general theory of software-reliability modeling," IEEE Transactions on Reliability, Vol. R-39, No. 1, 1990.

[369] Trachtenberg, M. "The linear software reliability model and uniform testing," IEEE Transactions on Reliability, Vol. R-34, 1985.

[370] Trachtenberg, M. "Discovering how to ensure software reliability," RCA Engineer, 1982 January/February.

[371] Trachtenberg, M. "Order and difficulty of debugging," IEEE Transactions on Software Engineering, SE-9, No. 6, 1983.

[372] Tran, T. L. "Software metrics: measuring the progress of software development," Microelectronics and Reliability, Vol. 28, No. 5, 1988.

[373] Troy, R. and Y. Romain. "A statistical methodology for the study of the software failure process and its application to the ARGOS center," IEEE Transactions on Software Engineering, SE-12, No. 9, 1986.

[374] Troy, R. and R. Moawad, "Assessment of software reliability models," IEEE CH1810-1, 1982.

[375] Tso, K. S. and A. Avizienis. "Community error recovery in N-version software: a design study with experimentation," Proceedings Fault Tolerant Computing Symposium, FTCS-17, 1987.

[376] Tso, K. S., A. Avizienis, and J. P. J. Kelly. "Error recovery in multi-version software development," in Proceedings SAFECOMP '86, Sarlat, France, October 1986.

[377] Upadhyaya, S. J. and K. K. Saluja. "A watchdog processor based general rollback technique with multiple retries," IEEE Transactions on Software Engineering, SE-12, January 1986.

[378] Vosbury, N. A. "The process design system," in Proceedings IEEE International Computers Software Application Conference (COMPSAC), 1979.

[379] Vouk, M. A., A. M. Paradkar, and D.F. McAllister. "Modeling Execution time of Multi-Stage N-version fault tolerant software," IEEE Computer Software and Applications Conference, November 1990.

[380] Wagoner, W. L. The final report on a software reliability measurement study. Report TOR-0074(41221)-1. The Aerospace Corp., El Segundo, CA, 1973.

[381] Wall, J. K. and P. A. Ferguson. "Pragmatic software reliability prediction," Proceedings Annual Reliability & Maintainability Symposium, 1977.

[382] Walsh, T. "A software reliability study using a complexity measure," Proceedings AFIPS Conference, 48, 1979.

[383] Walters, G. F. and J. A. McCall. "Software quality metrics for life-cycle cost reduction," IEEE Transactions on Reliability, R-28, No. 3, 1979.

[384] Weiss, S. N. and E. J. Weyuker. "An extended domain-based model for software reliability," IEEE Transactions on Software Engineering, Vol. SE-14, No. 10, 1988.

[385] Williams, M. W. H. "Reliability of large real-time control software systems," Proceedings International Conference on Reliable Software, 1973.

[386] Wojcik, A. S., J. Kljaich, and N. Srinivas. "A formal verification system based on an automated reasoning system," in Proceedings 21th Design Automate Conference, Albuquerque, NM, June 1984.

[387] Woodward, M., M. Hennell, and D. Hedley. "A measure of control flow complexity in program text," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[388] Xizi, H. "Non-linear regression for predicting software reliability," Microelectronics and Reliability, Vol. 28, No. 6, 1988.

[389] Xizi, H. "The limit condition of some time between failure models of software reliability," Microelectronics and Reliability, Vol. 30, No. 3, 1990.

[390] Yamda, S., H. Ohtera and H. Narihisa. "Software reliability growth models with testing-effort,"IEEE Transactions on Reliability, R-35, No. 1, 1986.

[391] Yamada, S., M. Ohba, and S. Osaki. "S-shaped software reliability growth models and their applications," IEEE Transactions on Reliability, Vol. R-33, 1984.

[392] Yamada, S. and S. Osaki. "Optimal software release policies for a nonhomogeneous software error detection rate model," Microelectronic and Reliability, Vol 26, 1986.

[393] Yamada, S. and S. Osaki. "Optimal software release policies with simultaneous cost and reliability requirements," European Journal Operational Research, Vol. 31, 1987.

[394] Yamada, S., H. Ohtera, and H. Narihisa. "Software reliability growth models with testing-effort," IEEE Transactions on Reliability, Vol. R-35, 1986.

[395] Yamada, S., H. Ohtera, H. Narihisa. "A testing-effort dependent software reliability model and its application," Microelectronics and Reliability, Vol. 27, 1987.

[396] Yamada, S., H. Narihisa, and S. Osaki. "Optimum release policies for a software system with a scheduled software delivery time," International Journal System Science, Vol. 15, No. 8, 1984.

[397] Yamada, S. and S. Osaki. "Reliability growth modeling for software error detection," IEEE Transactions on Reliability, Vol. R-32, No. 5, 1983.

[398] Yamada, S. and S. Osaki. "Software reliability growth modeling: Models and applications," IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, 1985.

[399] Yamada, S. and S. Osaki. "Software reliability growth modeling: models and applications," IEEE Transactions on Software Engineering, SE-11, No. 12, 1985.

[400] Yamada, S. and S. Osaki. "Cost-reliability optimal release policies for software systems," IEEE Transactions on Reliability, R-34, No. 5, 1985.

[401] Yamada, S. and S. Osaki. "Reliability growth model for hardware and software systems based on nonhomogeneous Poisson process: a survey," Microelectronics and Reliability, 23, No. 1, 1983.

[402] Yamada, S. "Software reliability analysis based on a nonhomogeneous error detection rate model," Microelectronics and Reliability, 24, No. 5, 1984.

[403] Yamada, S. "S-shaped reliability growth modeling for software error-detection," IEEE Transactions on Reliability, R-32, No. 5, 1983.

[404] Yau, S. S. and R. C. Cheung. "Design of self-checking software," 1975 Reliable Software, April 1975.

[405] Yee, J. G. and S. Y. H. Su. "A scheme for tolerating fault data in real-time systems," Proceedings COMPSAC, 1978.

[406] Yun, W. Y. and D. S. Bai. "Optimum software release policy with random life cycle," IEEE Transactions on Reliability, Vol. R-39, No. 2.

[407] Zahedi, F. and N. Ashrafi. "Software reliability allocation based on structure, utility, price, and cost," IEEE Transactions on Software Engineering, Vol. SE-17, No. 4, 1991.