

1987

Integration of software reliability into systems reliability optimization

Hsin-Hui Lin
Iowa State University

Follow this and additional works at: <https://lib.dr.iastate.edu/rtd>

 Part of the [Industrial Engineering Commons](#)

Recommended Citation

Lin, Hsin-Hui, "Integration of software reliability into systems reliability optimization " (1987). *Retrospective Theses and Dissertations*. 9274.

<https://lib.dr.iastate.edu/rtd/9274>

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.



300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

Order Number 8805109

**Integration of software reliability into systems reliability
optimization**

Lin, Hsin-Hui, Ph.D.

Iowa State University, 1987

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark .

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages _____
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received _____
16. Other _____

U·M·I

Integration of software reliability into systems reliability
optimization

by

Hsin-Hui Lin

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Industrial Engineering

Approved:

Signature was redacted for privacy.

~~In Charge of Major Work~~

Signature was redacted for privacy.

~~For the Major Department~~

Signature was redacted for privacy.

~~For the Graduate College~~

Iowa State University
Ames, Iowa

1987

TABLE OF CONTENTS

	PAGE
SECTION I. GENERAL INTRODUCTION	1
INTRODUCTION AND EXPLANATION OF DISSERTATION FORMAT	2
Abstract of Section II	5
Abstract of Section III	7
Abstract of Section IV	8
Abstract of Section V	9
Abstract of Section VI	9
SECTION II. A COMPARISON OF HEURISTIC RELIABILITY OPTIMIZATION METHODS	11
INTRODUCTION	12
REVIEW OF THE HEURISTIC REDUNDANCY ALLOCATION METHODS	18
Sharma and Venkateswaran Method (S-V)	21
The Gopal, Aggarwall, and Gupta Method (G-A-G)	21
The Extended Nakagawa and Nakashima Method (N-N)	23
The Kohda and Inoue Method (K-I)	25
THE SEQUENTIAL SEARCH METHODS	26
The Hooke and Jeeves Pattern Search Method (H-J)	27
The Gopal, Aggarwal, and Gupta Search Method (G-A-G)	28
FORMULATION OF THE RELIABILITY-REDUNDANCY ALLOCATION PROBLEM	31
COMPUTATION AND RESULTS	34
REFERENCES	39
SECTION III. RELIABILITY OPTIMIZATION WITH THE LAGRANGE MULTIPLIER AND BRANCH-AND-BOUND TECHNIQUES	41
INTRODUCTION	42
THE LAGRANGE MULTIPLIER AND KUHN-TUCKER CONDITIONS	44
THE BRANCH-AND-BOUND TECHNIQUE IN INTEGER PROGRAMMING	48
NUMERICAL EXAMPLES	50
Example 1	50
Example 2	51
CONCLUSION	59
REFERENCES	60

SECTION IV. A REVIEW AND CLASSIFICATION OF SOFTWARE RELIABILITY MODELS	62
INTRODUCTION	63
CHARACTERISTICS OF SOFTWARE RELIABILITY MODELS	64
General Description of Software and Software Reliability	65
Bug-Counting Concept	66
Error Size	67
User Environment	68
Time Index	69
Software Life Cycle	70
Graph Representation of a Program	71
Software Reliability versus Hardware Reliability	72
Error Analysis	74
Classification by severity	74
Some special errors	75
Classification by the type of error	76
Classification by error introduced in the software life cycle phase	77
Classification by error removed in the software life cycle phase	78
Classification by the techniques of error removal	78
CLASSIFICATION OF SOFTWARE RELIABILITY MODELS	80
The Deterministic Models	83
Software science	84
Entropy function (information content)	85
Software quality attributes	88
Complexity metrics	91
Lines of code	92
Program change	92
Job step	93
Data binding	93
Data span	93
Cyclomatic number	93
Maximum intersection number	95
Knot count	95
Calls and jumps	95
Maintainability	95
Accessibility	97
Testability	97
Testedness	98
Program evolution	99
Schneider model	100
Hybrid model	101
Environmental factors and error estimation	101
The Probabilistic Models	101
Error seeding model	102
Reliability growth model	104

Duane growth model	104
Weibull growth model	105
Wagoner's Weibull model	107
Logistic growth curve model	108
Gompertz growth curve model	108
Hyperbolic reliability growth model	109
Curve fitting model	110
Estimation of errors	110
Estimation of change	111
Estimation of time between failures	112
Estimation of failure rate	112
Isotonic regression	112
Exponential regression	113
Input-domain model	114
Basic input-domain model	114
Input-domain based stochastic model	118
Execution path model	120
Shooman decomposition model	120
Program structure model	122
Littlewood Markov structure model	122
Cheung's user-oriented Markov model	124
Failure rate models	126
Jelinski and Moranda De-Eutrophication Model	130
Extension of J-M model for varying program size	131
Jelinski-Moranda geometric De-Eutrophication model	132
Moranda geometric Poisson model	133
Schick and Wolverton model	133
Modified Schick and Wolverton model	134
Goel and Okumoto imperfect debugging model	134
Nonhomogeneous Poisson process model	135
Musa exponential model	138
Goel and Okumoto NHPP model	139
S-shaped growth model	140
Delayed S-shaped growth model	140
Inflection S-shaped growth model	141
Hyperexponential growth model	142
Markov chain	143
Linear death model with perfect debugging	144
Linear death model with imperfect debugging	145
Nonstationary linear death model with perfect debugging	147
Nonstationary linear birth-and-death model	148
Other types of probabilistic models	151
REFERENCES	155
SECTION V. RELIABILITY COSTS IN SOFTWARE LIFE-CYCLE MODELS	175
INTRODUCTION	176

REVIEW OF THE RELIABILITY-RELATED SOFTWARE COST MODELS	179
Cost Estimation	179
Resource Allocation	180
Program Evolution	182
SOFTWARE RELIABILITY AND COST	184
CONCLUSION	190
REFERENCES	191
SECTION VI. RELIABILITY OPTIMIZATION WITH SOFTWARE COMPONENT	193
SOFTWARE RELIABILITY-COST FUNCTION	194
SOFTWARE REDUNDANCY	199
Two-Component Model	199
Two-Component Markov Model With Common-Cause	202
Three-Component Model	204
Three-Component Markov Model With Common-Cause	206
N-Component Model	208
N-Component Markov Model With Common-Cause	211
FORMULATION OF THE HARDWARE-SOFTWARE RELIABILITY OPTIMIZATION . . .	216
A NUMERICAL EXAMPLE	219
REFERENCES	222
SECTION VII. CONCLUSIONS	223
CONCLUSIONS AND SUGGESTIONS FOR FUTURE STUDY	224
ACKNOWLEDGEMENTS	228

LIST OF TABLES

	PAGE
TABLE 2.1. Ranges for coefficients of constraint functions	34
TABLE 2.2. Simulation results I	37 ^a
TABLE 2.3. Simulation results II	38 ^a
TABLE 3.1. Data for example 1	51
TABLE 3.2. Data for example 2	54
TABLE 3.3. Comparison of two methods	58
TABLE 4.1. Software reliability versus hardware reliability	73
TABLE 4.2. Summary of References	153
TABLE 5.1. Reliability cost and software life-cycle phases	178
TABLE 6.1. Data for numerical example	220
TABLE 6.2. Result of the numerical example	221

LIST OF FIGURES

	PAGE
FIGURE 2.1. Structure diagrams	14
FIGURE 2.2. Combination of heuristic and search methods	16
FIGURE 2.3. Combination of H-J search and heuristic methods	29
FIGURE 2.4. Combination of G-A-G search and heuristic methods	30
FIGURE 3.1. Branch-and-bound of example 1	52
FIGURE 3.2. Branch-and-bound of example 2	55
FIGURE 4.1. Failure process	126
FIGURE 4.2. Linear death with perfect debugging	145
FIGURE 4.3. Linear death with imperfect debugging	146
FIGURE 4.4. Nonstationary linear death with perfect debugging	148
FIGURE 4.5. Nonstationary birth-and-death	149
FIGURE 4.6. Bivariate process of fault-count and failure count	150
FIGURE 6.1. Two-component software redundancy	200
FIGURE 6.2. Transformed two-component software redundancy	200
FIGURE 6.3. Two-component Markov model with common-cause failure	202
FIGURE 6.4. Three-component software redundancy	204
FIGURE 6.5. Transformed three-component software redundancy	205
FIGURE 6.6. Three-component Markov model with common-cause failures	206
FIGURE 6.7. N-component software redundancy	209
FIGURE 6.8. Transformed N-component software redundancy	209
FIGURE 6.9. N-component Markov model with common-cause failure	212

SECTION I. GENERAL INTRODUCTION

INTRODUCTION AND EXPLANATION OF DISSERTATION FORMAT

Reliability is extremely important for systems involving issues of high cost (e.g., space program), safety (e.g., nuclear power plant), or security (e.g., military equipment). By definition, reliability is the probability of failure-free operation of a system under specified conditions for a specified period of time. The system could be a hardware system, a software system, a human body, or a combination of these. As missions to be accomplished are becoming more and more complicated, the need for highly reliable systems is inevitable. In achieving high reliability, three problems are faced by reliability engineers. First, the reliability-cost function increases exponentially. Second, the reliability of a component is usually limited by technology. Third, resources for achieving high reliability are limited. These problems lead to the subject of reliability optimization.

Fifteen years ago, reliability studies concentrated on hardware systems. Both reliability theory and reliability optimization are well-known in terms of problem formulation and problem solving techniques. Since the 1960s, software has become increasingly an important part of larger systems. Since 1970, the cost of software has surpassed the cost of hardware as the major cost factor of a system. In response to this dramatic change, researchers began developing models for software reliability in the 1970s. Compared to the exponential growth in demand and size of today's software projects,

software reliability modeling is still in its infancy. In many cases, software cannot be treated as an isolated element. A complex system contains hardware subsystems and software subsystems both interacting with each other. Unfortunately, very few researchers have studied this issue. This research investigates methods for optimizing system reliability involving software and hardware.

Traditionally, the reliability of a hardware system is improved by adding redundant components or by using better components. Determining the number of redundancies at each stage or the reliability level at each stage under available resources is the major concern in reliability optimization. Since the available options of an identical function component are finite and the number of redundancies is an integer, the growth of reliability, in either case, is discrete. Numerous techniques have been proposed for reliability optimization problems. The Lagrange multiplier method, dynamic programming method, branch-and-bound method, maximum principle method, and heuristic method are popular techniques.

When software is involved, the techniques for hardware reliability optimization have to be reevaluated. First, redundancy of software can not be treated the same as hardware. The failure of a hardware component is primarily due to random failures and material deterioration. Parallel redundancy is based on the assumption of independent failure of components. Software failure is due to incorrect logic or incorrect statements in the program. An input state

which causes one copy of software to fail will do so for all copies. Although some people may argue that a redundant copy of software can be produced by an independent group, research has indicated that people make the same mistakes in software development. The degree of dependency among independent groups remains unanswered. Therefore, the issue of software redundancy is much more complicated. Secondly, the improvement of software reliability is primarily through debugging rather than redundancy. Even though the "number of faults" in a program is countable and the actual improvement of software reliability is discrete, most software reliability models are continuous models as opposed to the discrete growth in hardware redundancy. Therefore, the traditional method of integer programming for hardware redundancy allocation is not appropriate for software.

When reliability optimization involves software and hardware, two types of decision variables need to be determined. For hardware, the decision variable is the number of redundancies which is an integer. For software, the decision variable is the reliability level which is a real number. When both types of decision variables are involved, the problem becomes a mixed-integer programming problem. Moreover, the reliability function and the constraint functions for software and hardware systems are nonlinear functions. Mixed integer programming for linear function is better known. But very few methods have been proposed for nonlinear mixed-integer programming problem.

This dissertation uses the alternate format. It is composed of five self-explanatory, yet related papers. In Sections II and III, two methods are proposed for mixed-integer reliability optimization problems. Section IV is a review and classification of software reliability models. It focuses on the nature of software, assumptions of software reliability modeling, factors affecting software reliability modeling, and modeling techniques. This review paves the way for future research in software reliability modeling and applications of software reliability model. Section V is a software life-cycle cost model for the optimal release time. The motivation is to point out the issue of software reliability cost and emphasize the life cycle approach to the problem. Section VI integrates the material from Sections II through V. The purpose is to incorporate software into the reliability optimization problem. An abstract of each paper appears below.

Abstract of Section II

Section II, "A Comparative Study of Heuristic Reliability Optimization Methods" investigates the effectiveness of a mixed-integer programming method. This method is a combination of the heuristic redundancy method and the sequential search method. The heuristic method determines the integer variables (number of redundancies) by assuming that the real variables (reliability level) are known. The sequential search method determines the real variables. At each

iteration, a point in the multi-dimensional real space is chosen. Once the real variables are fixed, the heuristic method is applied to find the integer variables. When both types of decision variables are determined, the objective function can be computed. The next iteration moves to a new point according to the search strategy. As the search proceeds, the current best solution is continuously updated.

Heuristic redundancy methods and sequential search methods were developed independently for different types of problems. Many heuristic redundancy methods have been proposed for the integer programming problem. Also, many sequential search methods have been proposed for real-variable peak-finding problems. This paper investigates their relative merits in obtaining the optimum solution. Four heuristic methods and two sequential search methods were studied. Simulation was used to test these eight combinations on 100 simulated problems. The test problem is based on a bridge structure with three nonlinear constraints.

Results of this simulation show that when heuristic methods are used to solve pure integer programming problem, the quality of the answer is proportional to the CPU time required to obtain the answer. When the sequential search technique is added to the heuristic method to solve the mixed-integer programming problem, the sequential search method is more significant in obtaining the optimal solution. This method is slow. But it takes advantage of the existing search methods and heuristic methods, and can solve a variety of problems.

Abstract of Section III

Section III, "Reliability Optimization with Lagrange Multiplier and Branch-and-Bound," presents a new method for the mixed-integer reliability optimization problem by using the Lagrange multiplier method and the branch-and-bound method. The Lagrange multiplier method solves a constrained problem by introducing Lagrange multipliers. By multiplying Lagrange multipliers to each constraint and adding to the objective function, the constrained problem becomes unconstrained. According to the Kuhn-Tucker conditions, the necessary condition for an optimum to exist is that the first derivative vanishes. By taking derivative with respect to the number of components at each stage, the reliability level of the components, and Lagrange multipliers, a set of simultaneous equations are formed. The solutions to the set of simultaneous equation are stationary points to the problem. Since this method is based on differentiation, all variables are treated as real variables. A solution obtained by this method is a real number solution.

The branch-and-bound method is then used to obtain the integer solution for integer variables. The branch-and-bound for integer programming divides the solution space by imposing a lower bound constraint to one problem and an upper bound constraint to another problem. For example, a constraint $x \leq 3$ is added to one problem and $x \geq 4$ is added to another when an integer variable takes value between 3 and 4. The process continues until all the integer variables become integer and no better solutions can be found.

The results show that this method is superior to the method presented in Section I. The reasoning process is more logical than the heuristic method in obtaining the optimal solution.

Abstract of Section IV

Section IV, "A Review and Classification of Software Reliability Models," focuses on how a software reliability model is derived and how the reliability of software can be measured.

Hardware reliability models are normally based on failure data. If a particular distribution fits very well to the failure data of a particular hardware, this distribution is used to estimate and predict the reliability of that hardware. However, this approach is not appropriate for software. Although many software reliability models have been proposed, very few of them have been tested on a variety of software products and very few of them have proven to be effective for a variety of software products. One of the difficulties is that each software is a new product. Past experiences can only serve as a reference point.

Most software reliability models are theoretical models derived from assumptions. Software reliability theoreticians believe that there are some factors governing the failure process. Depending upon the assumptions imposed, dozens of software reliability models have been proposed. These models and related materials from about 300 papers are reviewed and classified in Section IV. Attributes of

software reliability models are also discussed. Special attentions is given to the probabilistic models which can further be divided into the binomial model, Poisson model, and Markov process.

Abstract of Section V

The reliability optimization problem in Sections II and III implied that a functional relationship between software reliability and cost existed. Section V, "Reliability Cost in Software Life Cycle Models," investigates this relationship. It is recognized that 60% of the software life-cycle costs are incurred after release and the maintenance cost depends heavily upon the reliability at the release time. Thus, an optimum release time model based on the nonstationary birth-and-death process is proposed. The trade-off between debugging cost and maintenance cost is studied.

Abstract of Section VI

Section VI, "Reliability Optimization with Software Components," integrates reliability-redundancy allocation techniques, software reliability-cost function, and software redundancy into a system reliability optimization problem. A series system with hardware components and software components is studied. The failure of hardware redundancies are independent of each other, while the failure of software redundancies are partially independent.

The unknown variables of this reliability optimization problem are the number of hardware redundancies, the number of software redundancies, the hardware reliability levels and the software reliability levels. The mixed-integer programming techniques in Sections II and III are used to solve this problem. Software reliability model in Section IV and software reliability cost in Section V are adapted to formulate the objective and constraint functions of this problem.

SECTION II. A COMPARISON OF HEURISTIC RELIABILITY OPTIMIZATION METHODS

INTRODUCTION

Many optimization techniques have been proposed to allocate redundancy or reliability level for a system of series configuration [15]. But more important than optimizing system reliability with respect to a single type of variable, both redundancies and reliability levels are usually determined simultaneous. The purpose of this study is to investigate methods to maximize the reliability of a complex system subject to nonlinear constraints. Sequential search techniques and reliability optimization heuristics commonly used for optimizing a single type of variable are combined for solving a mixed-integer programming problem. Performance of these heuristics is accomplished through simulation.

In this study, the system reliability is based on the following definition of a system. A system is composed of one or more stages (or subsystems). A stage is a unique functional unit of the system and may be composed of one or more components. Cost functions at different stages are assumed to be additive. The system reliability is the probability of successful operation of a system for a specified period of time under given conditions. It is usually expressed in terms of the reliabilities of both the stages and components. In evaluating the system reliability, it is necessary to specify the system structure, the component failure process, and the definitions of failures.

For a series system, the system is operational only when all the stages are operational. For a parallel system, the system is

operational if one or more stages are operational. A general system (or complex system) is a nonparallel nonseries system, whose reliability can be evaluated by probability theory once the system structure is clearly defined. Other types of structures are the parallel-series system and the series-parallel system. In this study, a nonparallel-nonseries system is investigated. It is assumed that its components are independent of each other and the component reliability is deterministic. Figure 2.1 lists the system structures and their system reliabilities.

The reliability of a system can be improved by increasing the component reliability or adding redundant components. The first method determines the component reliability levels to maximize the system reliability or minimize the total cost. However, this approach may not be economical because of the exponential increase of the reliability-cost function. Also, the highly reliable component may not be available. The second method determines the number of redundancies at each stage, which means that if more components are used, the system gets voluminous, heavy, and costly. Quite frequently, optimization problems refer to only one of these two options. In the design stage, however, the reliability optimization methods should consider the tradeoff between reliability and redundancy with respect to cost and performance requirements. The component reliability is a real number between 0 and 1, while the number of redundancies is an integer number. To optimize both decision variables simultaneously, a mixed-integer programming problem is involved.

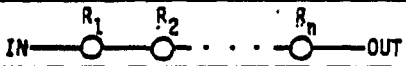
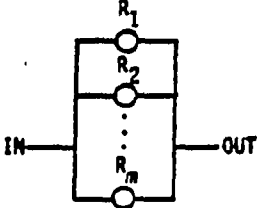
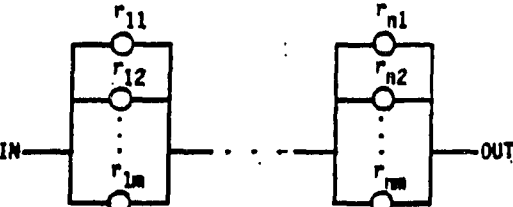
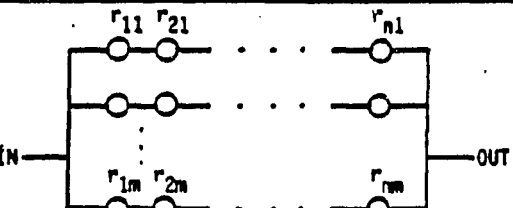
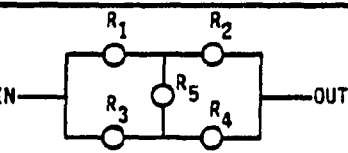
SYSTEM	STRUCTURE	SYSTEM RELIABILITY
SERIES SYSTEM		$R_s = \prod_{j=1}^n R_j$
PARALLEL SYSTEM		$R_s = 1 - \prod_{j=1}^m (1 - R_j)$
PARALLEL SERIES SYSTEM		$R_s = \prod_{j=1}^n (1 - \prod_{i=1}^m (1 - r_{ji}))$
SERIES PARALLEL SYSTEM		$R_s = 1 - \prod_{j=1}^m (1 - \prod_{i=1}^n r_{ij})$
BRIDGE SYSTEM		$R_s = R_1 R_2 + R_3 R_4 + R_1 R_4 R_5 + R_2 R_3 R_5 - R_1 R_2 R_3 R_4 - R_1 R_2 R_3 R_5 - R_1 R_2 R_4 R_5 - R_1 R_3 R_4 R_5 - R_2 R_3 R_4 R_5 + 2 R_1 R_2 R_3 R_4 R_5$

FIGURE 2.1. Structure diagrams

This mixed-integer reliability optimization problem was first given by Misra and Ljubojevic [9] to solve a four-stage series system using the Lagrange multiplier technique. Another method was given by Tillman et al. [13] that combines the sequential search method with a heuristic redundancy method proposed by Aggarwal et al. [2]. The sequential search method moves from point (a combination of decision variables) to point in the solution space to find the optimal solution of a multivariable function. When it is applied to the reliability optimization problem, the component reliabilities are the decision variables and the system reliability is the objective function to be maximized. For each move (change in the component reliability), the heuristic redundancy method is applied to determine the number of redundancies at each stage. Once the component reliabilities and the number of redundancies are determined, the system reliability is calculated and compared to the current best solution. If the solution is better, the current best solution is updated. The search continues until the stopping rule is reached. This method takes advantage of both the existing heuristic redundancy allocation methods and the sequential search methods. The algorithm is shown in Fig. 2.2. The third method, which modified some of the existing heuristics [4,13], was presented by Gopal et al. [4]. In their approach, component reliabilities are sequentially increased in order of descending value of a predefined sensitivity function. For every change in component reliability, redundancies are allocated to determine the new system reliability. At each stage, an inferior solution is rejected.

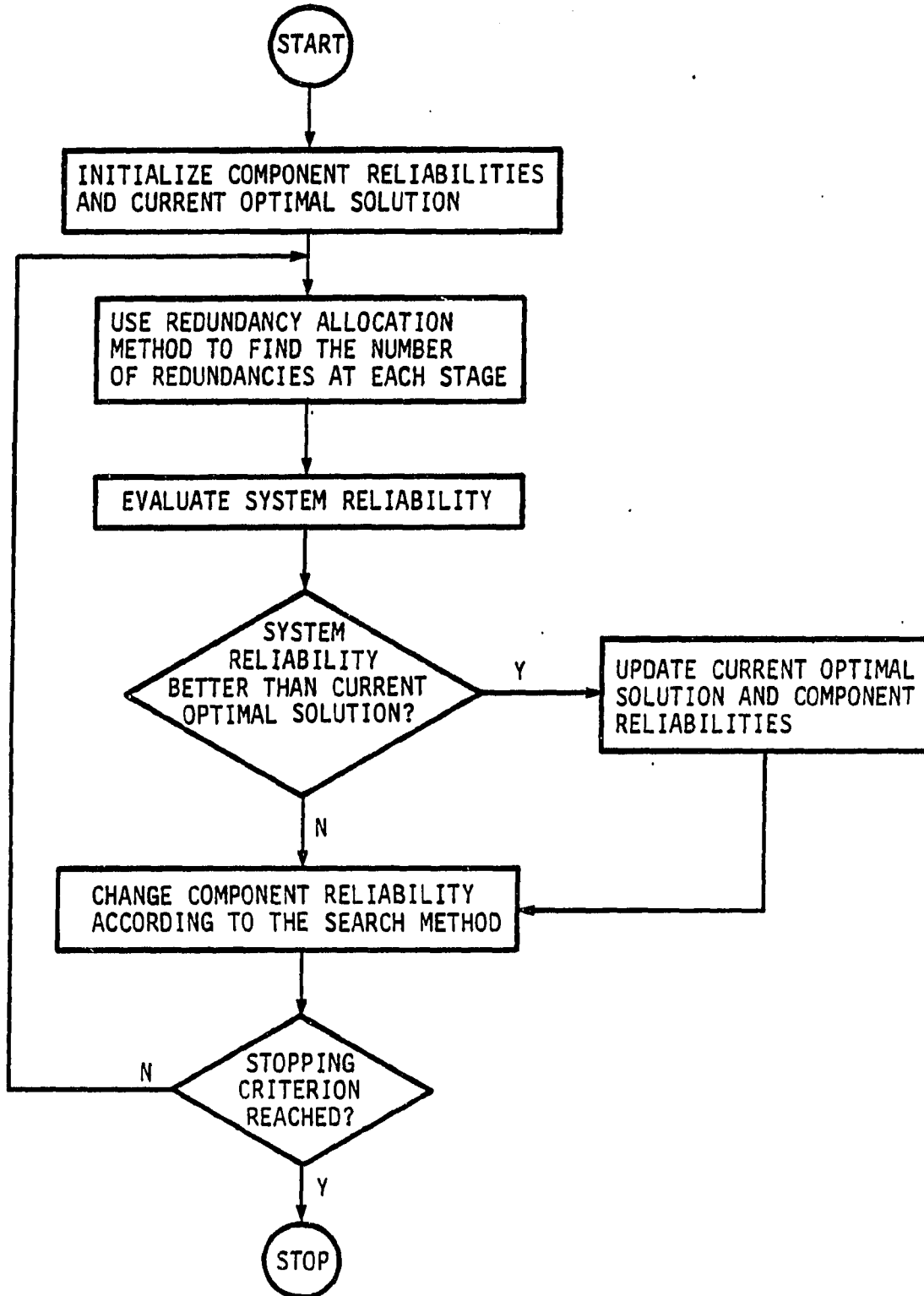


FIGURE 2.2. Combination of heuristic and search methods

Previous studies use a specific redundancy heuristic and a specific search method to solve a specific problem [4,13]. The heuristics' performance has not been investigated. This study intends to combine each of the two search methods [4,5] with each of the four major heuristic redundancy methods [3,6,11,12]. Comparison is based on a nonparallel-nonseries bridge system subject to three nonlinear constraints. One-hundred test problems are randomly generated. Each is tested by eight combinations of the methods.

REVIEW OF THE HEURISTIC REDUNDANCY ALLOCATION METHODS

Notation:

R_s, Q_s	reliability and unreliability of the system
R_j, Q_j	reliability and unreliability of the j th stage
r_j, q_j	reliability and unreliability of component j
x_j	number of components at stage j
\bar{R}	(r_1, \dots, r_N) ; vector of component reliability
\bar{X}	(x_1, \dots, x_N) ; vector of the number of components used at each stage
g_i	i th constraint
g_{ij}	amount of resource i consumed at stage j
b_i	amount of resource i available
b_i^c	$b_i - \sum_{j=1}^N \sum_{k=1}^{x_j} \Delta g_{ij}(k)$
$C_j(r_j)$	cost function of the j th component reliability
C_s	system cost function
h	step size; amount of increment in component reliability
$L+1$	set of all the stages whose reliabilities can be increased.
X^{*p}	current optimal solution
X^c	current solution
$X(\pm j)$	$(x_1, \dots, x_j \pm 1, \dots, x_N)$; add/subtract 1 at stage j
$X(-j, +s)$	$(x_1, \dots, x_j - 1, \dots, x_s + 1, \dots, x_N)$; subtract 1 from x_j

$$\begin{aligned}
 & \text{and add 1 to } x_s \\
 S(j) & \quad \Delta Q_s(x_j); \text{ decrement in } Q_s \text{ by increasing} \\
 & \quad x_j \text{ by 1} \\
 \Delta F_j(x_j) & \quad \ln R_j(x_j) - \ln R_j(x_j - 1) \\
 \Delta g_{ij}(x_j) & \quad g_{ij}(x_j) - g_{ij}(x_j - 1) \\
 \Delta Q_j(x_j) & \quad \text{decrement in } Q_j \text{ by increasing } x_j \text{ by 1.}
 \end{aligned}$$

The redundancy optimization problem can be formulated as

$$\text{Max } R_s(\bar{X}|\bar{R})$$

subject to

$$\sum_{j=1}^N g_{ij}(x_j) \leq b_i \quad \text{for all } i.$$

Assuming that the component reliabilities, $\bar{R} = (r_1, \dots, r_N)$, are given constants, the above problem is to determine the number of redundancies at each stage, $\bar{X} = (x_1, \dots, x_N)$.

Depending upon the type of structure, failure mode, and constraint functions, the above problem may be converted into different forms [7]. Since 1956, numerous techniques have been proposed to solve a variety of reliability redundancy optimization problems [14,15]. Yet, none of them can effectively solve a large-scale general system with multiple nonlinear constraints. Those techniques are restricted to one or more of the following.

- Limited to a special type of system configuration, normally the series system,
- Limited to a special type of constraint function, for example, the linear constraints,
- Limited by the dimension of the problem,
- Nonguaranteed global optimal solution,
- Complicated computation.
- Treated the problem as a nonlinear noninteger optimization, then approximated the optimal decision variables to an integer number through extensive discussion.

Therefore, heuristic redundancy allocation methods have been suggested. The heuristic method has the properties of simplicity, generality, and efficiency [7,15]. Many algorithms have been able to solve the series system with multiple nonlinear constraints [2,3,6,7,11], although global optimality is not yet guaranteed.

Most redundancy allocation heuristics are based on the following steps.

1. Initialize each stage with one component.
2. Evaluate the sensitivity function to determine the stage to which a redundant component is to be added. A sensitivity function is defined as the trade-off between the increment of system reliability and the resources consumption.
3. Increase the number of redundancies by one at the proposed stage and repeat steps 2 and 3 until the constraint is violated.

Four major heuristic redundancy methods compared in this study are summarized below.

Sharma and Venkateswaran Method (S-V)

The S-V method [12] is based on a series system of small Q_j 's. For a series system, the unreliability of the system can be expressed as follows.

$$\text{Min } Q_s = 1 - \prod_{j=1}^N (1 - Q_j) \approx \sum_{j=1}^N Q_j = \sum_{j=1}^N q_j^{x_j}$$

By this approximation, the maximization of R_s is equal to the minimization of the sum of Q_j 's. Therefore, adding a redundancy to the stage having the largest Q_j would increase the system reliability by the largest amount. Although this method is simple and efficient, it does not incorporate the constraint functions into the selection criteria. In general, it does not yield the optimal solution as shown by Nakagawa and Miyazaki [10] and this study. The relative error increases and the optimality rate decreases as the number of decision variables increase.

The Gopal, Aggarwall, and Gupta Method (G-A-G)

The G-A-G method [3] is an improved version of their previous works [1,2]. A relative increment of resource is defined as

$$\Delta g_{ij}^r(x_j) = \Delta g_{ij}(x_j) / \max_j \{ \Delta g_{ij}(x_j) \}.$$

A selection factor evaluates the ratio of relative increment of resource over the decrement of stage unreliability. A redundant component is proposed to be added to the stage having the least value of selection factor. The selection factor is defined as

$$F_j(x_j) = \max_i \{ \Delta g_{ij}^F(x_j) \} / \Delta Q_j(x_j).$$

For the series,

$$\Delta Q_j(x_j) = q_j^{x_j} - q_j^{x_j+1} = r_j q_j^{x_j}.$$

The selection factor, $F_j(x_j)$, can be written in the following forms.

a. The series system with linear constraints:

$$F_j(x_j) = F_j(x_j-1) / q_j \quad \text{for } x_j > 1$$

$$F_j(1) = \max_i \{ \Delta g_{ij}^F(1) \} / r_j q_j \quad \text{for } x_j = 1.$$

b. The series system with nonlinear constraints:

$$F_j(x_j) = \max_i \{ \Delta g_{ij}^F(x_j) \} / f_j(x_j)$$

$$f_j(x_j) = q_j f_j(x_j-1) \quad \text{for } x_j > 1$$

$$f_j(1) = r_j q_j \quad \text{for } x_j = 1.$$

c. The complex system:

$$\Delta Q_s(x_j) = Q_s(Q_1, \dots, (Q_j = q_j^{x_j}), \dots, Q_N)$$

$$\begin{aligned}
& - Q_s(Q_1, \dots, (Q_j = q_j^{x_j+1}), \dots, Q_N) \\
& = \frac{\partial Q_s}{\partial Q_j} (q_j^{x_j} - q_j^{x_j+1}) \\
& = r_j Q_j \frac{\partial Q_s(\bar{X})}{\partial Q_j(x_j)} \\
F_j(x_j) & = \frac{\text{Max } \{\Delta g_{ij}^F(x_j)\}}{r_j Q_j \{\partial Q_s(\bar{X}) / \partial Q_j(x_j)\}}.
\end{aligned}$$

The G-A-G method is simple, fast, and easily programmable. It can be applied to a series or a general system with multiple nonlinear constraints. For a series system, the recursive representation of the selection factor simplifies the computation.

The Extended Nakagawa and Nakashima Method (N-N)

Originally, the N-N method [11] was based on the series system.

The problem was formulated as

$$\text{Max } R_s = \prod_{j=1}^N R_j(x_j)$$

subject to

$$\sum_{j=1}^N g_{ij}(x_j) \leq b_i \quad \text{for all } i.$$

With the assumption of monotonically nondecreasing constraints, the above problem can be transformed into

$$\text{Max } \ln R_s(\bar{X}) = \sum_{j=1}^N \sum_{k=1}^{x_j} \Delta f_j(k)$$

subject to

$$\sum_{j=1}^N \sum_{k=1}^{x_j} \Delta g_{ij}(k) \leq b_i \quad \text{for all } i$$

$$\Delta f_j(k) \geq 0 \quad \text{for all } j \text{ and } k$$

$$\Delta g_{ij}(k) \geq 0 \quad \text{for all } i, j \text{ and } k.$$

A balancing coefficient, a , balances the weights between the increment of system reliability and the resource consumption. The sensitivity function is defined as

$$S_j = \Delta f_j(x_j+1) \left[(1-a) \cdot \min_{k \in L+1} \{\Delta x_k\} + a \Delta x_j \right]$$

where

$$\Delta x_j = \min_i \{b_i^c / \Delta g_{ij}(x_j + 1)\}.$$

A redundant component is then proposed to be added to the stage having the largest S_i . Fourteen balancing coefficients (0.0, 0.1, ..., 1.0, 1/0.9, 1/0.6, 1/0.3) are evaluated. The best solution is the final solution.

This method was later extended to the general system by redefining $\Delta f_j(x_j) = \Delta Q_s(x_j)$ [7]. According to Nakagawa and Miyazaki [10] and the results of this Section, the N-N method is the most accurate heuristic redundancy optimization method, but it requires the longest execution time because of the repetitive computation of 14 balancing coefficients.

The Kohda and Inoue Method (K-I)

The previous three methods improve system reliability by adding redundancy one-by-one to the selected stage. The algorithm stops when any constraint is violated. The K-I method [6] further examines the solution by adding a redundancy to one stage and subtracting a redundancy from another stage to determine whether the new solution is feasible and better.

For every $X^{*P}(-j)$, the maximum $S(k_j)$ over $\{j | X^{*P}(-j, +s) \text{ is feasible}\}$ is obtained. Then the deviation,

$$D(j) = [S(k_j) - S(j)] | X^{*P}(-j)$$

is calculated. If the maximum deviation, $D(\ell)$, is greater than zero, the system reliability can be improved by adding one redundant unit to stage ℓ and subtracting one from stage k_ℓ . If the constraint is not monotonically nondecreasing, two redundancies are added to the stages to see if the solution is feasible and better. This method serves as an improved step to the solution obtained by any redundancy allocation method.

THE SEQUENTIAL SEARCH METHODS

Without considering redundancy, the reliability allocation problem can be formulated as

$$\text{Max } R_s(\bar{R})$$

subject to

$$\sum_{j=1}^N g_{ij}(\bar{R}) \leq b_i \quad \text{for all } i$$

or

$$\text{Min } C_s(\bar{R})$$

subject to

$$\begin{aligned} R_s(\bar{R}) &\geq R^* \\ r_j &\geq r_j^* \end{aligned} \quad \text{for all } j$$

where

R^* and r_j^* are given lower bounds.

This problem, a typical nonlinear programming problem, restricts the decision variables, $\bar{R} = (r_1, \dots, r_N)$, between 0 and 1. To solve such a problem, numerous search techniques can be utilized. There are two basic types of search techniques: the simultaneous search and the sequential search. The simultaneous search, also called the exhaustive

search, evaluates the function value at predetermined points. The results of an experiment are not used to determine the next experiment. On the other hand, the results of a sequential search provide information for the next experiment.

The exhaustive search method cannot be applied to a problem of moderate or large size. The use of the sequential search technique to handle the real part of the mixed-integer reliability optimization problem was first presented by Tillman et al. [13] and later by Gopal et al. [4]. These two sequential search methods are simple and efficient compared to the other search methods. Neither requires a differentiable objective function. They can be easily understood and implemented without any special mathematical background. Two search techniques proposed are summarized below.

The Hooke and Jeeves Pattern Search Method (H-J)

The H-J pattern search [5] begins with an arbitrarily selected base point. The search is composed of the exploratory move and the pattern move. An exploratory move finds a new pattern (direction) by adding and subtracting a step size to the current base point. A pattern move actually makes an improvement toward the optimal solution by adding two times the difference between the previous base point and the current base point. For each move (change in the decision variables), the function value is evaluated and compared with the current optimal solution. If a move gives a better solution, the base

point and current optimal solution are updated. Otherwise, the step size is reduced by half. The search ends when the step size is smaller than a predetermined minimum step size and the functional value sees a limited improvement. The algorithm is shown in Fig. 2.3.

The Gopal, Aggarwal, and Gupta Search Method (G-A-G)

The G-A-G search method simplifies the search procedure by simply adding a step size to the component reliability. A sensitivity function was introduced to determine the order of adding a step size to the component reliability. The sensitivity function is defined as

$$S_j(r_j, \bar{X}) = [R_s(r_1, \dots, r_{j+h}, \dots, r_N; \bar{X}) - R_s(\bar{R}, \bar{X})] / [C_j(r_{j+h}) - C_j(r_j)]$$

The algorithm is shown in Fig. 2.4. This method, although very simple and efficient, does not yield satisfactory solutions.

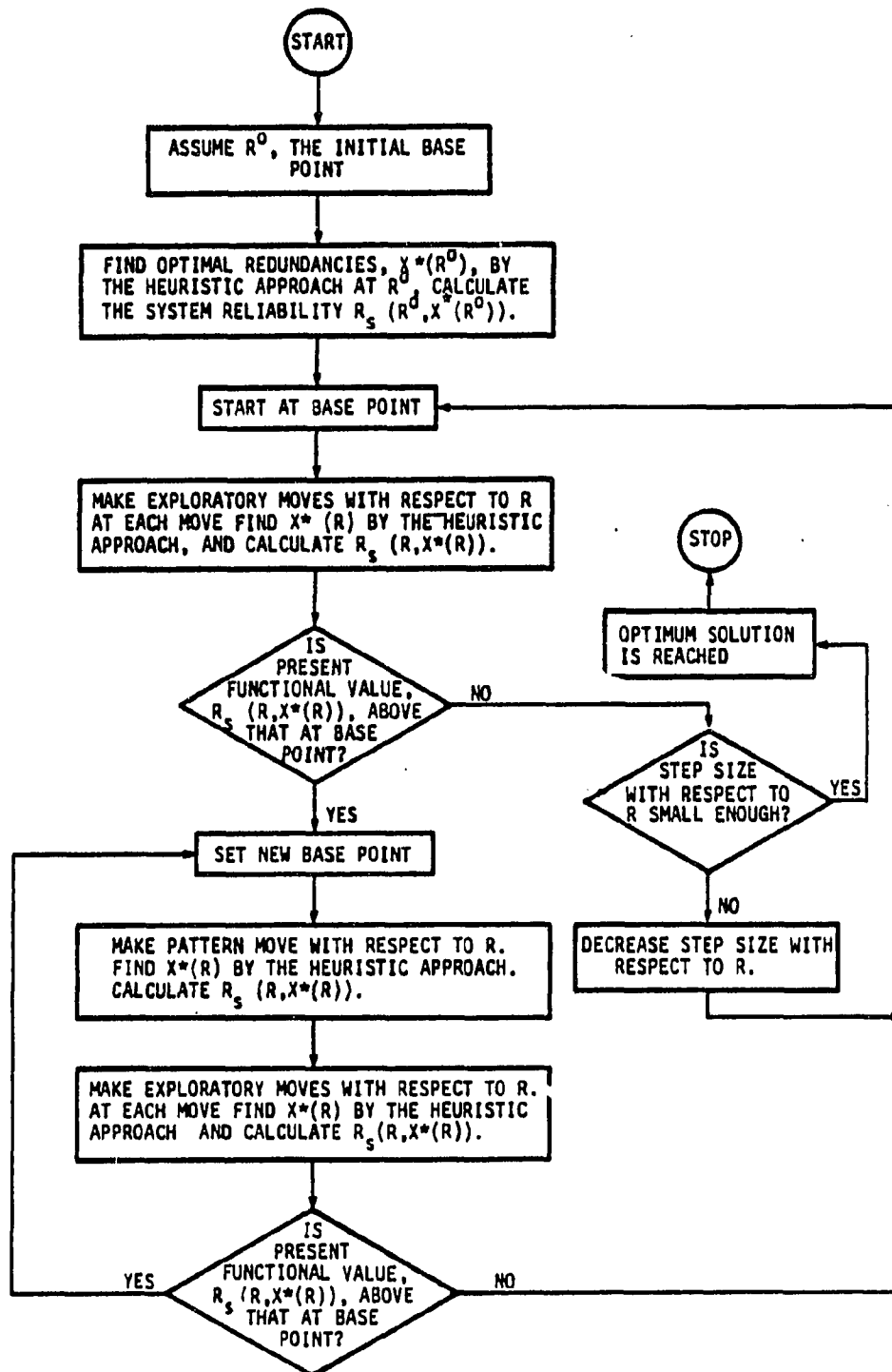


FIGURE 2.3. Combination of H-J search and heuristic methods

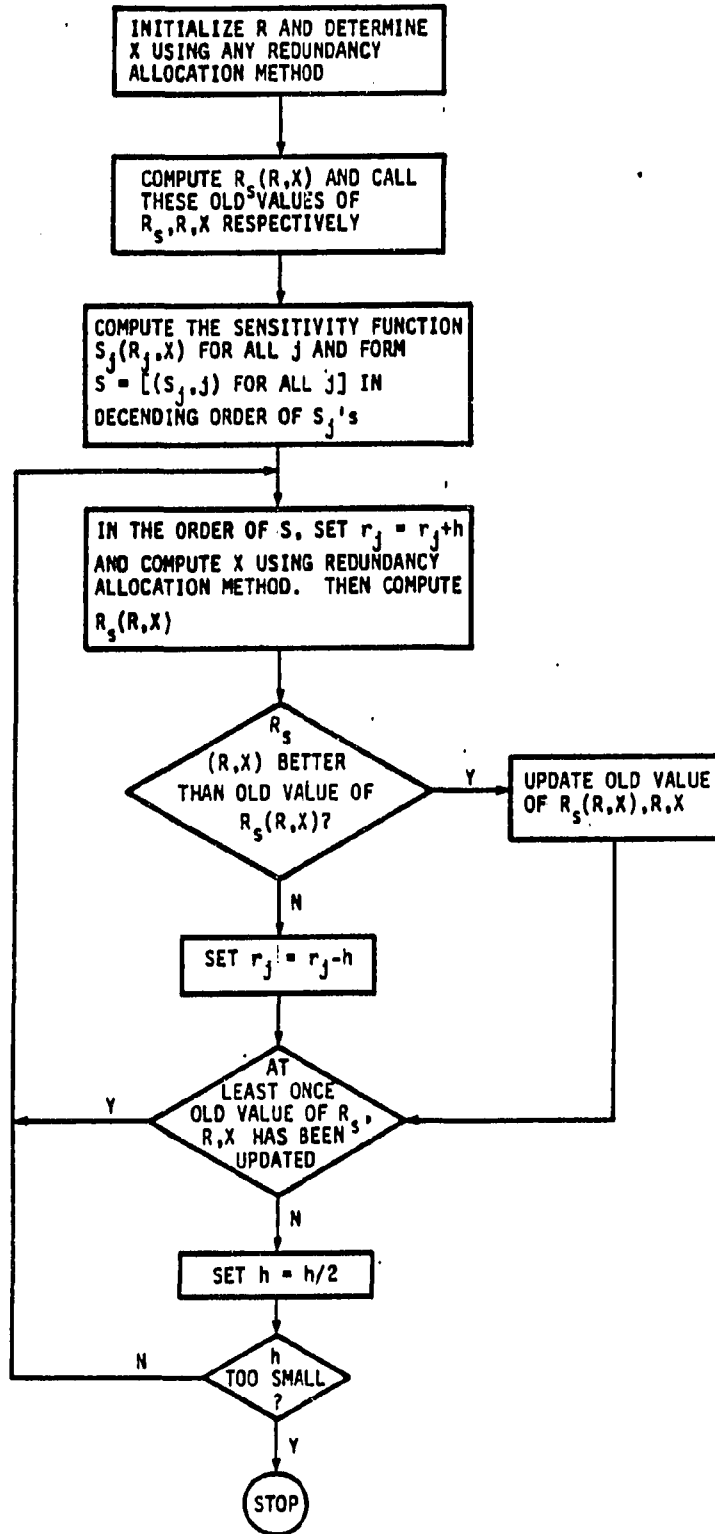


FIGURE 2.4. Combination of G-A-G search and heuristic methods

FORMULATION OF THE RELIABILITY-REDUNDANCY ALLOCATION PROBLEM

In the design stage, it must be decided whether or not to use highly reliable components or to add redundancies. For a system such as a space shuttle, a system reliability near 1.0 is desirable. Yet to minimize the shuttle weight, adding redundancy would be a real burden. On the other hand, in an ordinary industrial product, adding redundancies can be a good solution, since the cost of a reliable component is at least an exponential function of its reliability measure. The following mixed-integer reliability optimization problem is formulated to allow flexibility in the decision process for the systems falling between these two extremes.

$$\text{Max } R_s(\bar{R}, \bar{X})$$

subject to

$$\sum_{j=1}^N g_{ij}(r_j, x_j) \leq b_i \quad \text{for all } i.$$

$\bar{R} = (r_1, \dots, r_N)$ and $\bar{X} = (x_1, \dots, x_N)$ are to be determined for given g_{ij} 's and b_i 's.

The system studied is the bridge system shown in Fig. 2.1. Its system reliability can be evaluated as follows.

$$R_s = P_r\{\text{system is good} \mid \text{component 5 is good}\}$$

$$\times P_r\{\text{component 5 is good}\}$$

$$\begin{aligned}
& + P_r \{ \text{system is good} \mid \text{component 5 is failed} \} \\
& \times P_r \{ \text{component 5 is failed} \} \\
& = R_1 R_2 + R_3 R_4 + R_1 R_4 R_5 + R_2 R_3 R_5 - R_1 R_2 R_3 R_4 - R_1 R_2 R_3 R_5 \\
& - R_1 R_2 R_4 R_5 - R_1 R_3 R_4 R_5 - R_2 R_3 R_4 R_5 + 2 R_1 R_2 R_3 R_4 R_5.
\end{aligned}$$

The three nonlinear constraint functions from [13] are

$$g_1(\bar{X}) = \sum_{j=1}^5 P_j x_j^2 - P \leq 0$$

$$g_2(\bar{X}, \bar{R}) = \sum_{j=1}^5 a_j \left(-t / \ln r_j \right)^{\beta_j} (x_j + \exp(x_j/4)) - C \leq 0$$

$$g_3(\bar{X}) = \sum_{j=1}^5 W_j x_j \exp(x_j/4) - W \leq 0$$

x_j 's ≥ 1 are integers.

$0 < r_j$'s < 1 .

The first constraint models the combination of volume and weight, which is a function of the number of redundancies. The second constraint models the cost, which is a function of the number of redundancies as well as the component reliability. The third constraint models the weight, which is a function of the number of redundancies only. The reliability function and cost function of the second constraint are

$$r_j = \exp(-\lambda_j t)$$

and

$$C_j(r_j) = a_j (1/\lambda_j)^{\beta_j} = a_j (-t/\ln r_j)^{\beta_j}$$

where λ_j is the failure rate, t is the time, and a_j and β_j are cost coefficients.

COMPUTATION AND RESULTS

One-hundred sets of constraint coefficients of specified ranges as listed in Table 2.1 are randomly generated from uniform distribution. Each set of coefficients represents a test problem.

TABLE 2.1. Ranges for coefficients of constraint functions

Coefficient	Range	Coefficient	Value	Coefficient	Value
P_j	1.0 - 10.0	P	100	t	1000
W_j	5.0 - 15.0	W	200	β_j	1.5
$a_j \times 10^5$	0.3 - 9.0	C	200	-	-
where $j=1,2,3,4,5$.					

In order to investigate the effects of combining or not combining with the search method, the problems were first tested on the four heuristic redundancy methods with constant component reliability. Two component reliability levels, 0.7 and 0.85, were tested. The same problems were then tested by combining two sequential search methods with four heuristic redundancy methods using the algorithm outlined in Figs. 2.3 and 2.4. The extended N-N method used three values (0.5, 1.5, and 2.5) instead of 14 values to reduce execution time. The K-I method utilized Aggarwal's redundancy method [1] to find a solution before the perturbation method is applied. Since the initial base

point may affect the final answer, two initial base points, 0.5 and 0.7 were tested on each combination.

The results are compared in the following criteria:

1. Optimality rate (O): the number of times the method yields the highest system reliability.
2. Maximum error rate (M): the number of times the method yields the lowest system reliability.
3. Average absolute error (A): $\frac{100}{\sum_{j=1}^{100}} |R_s^{ij} - R_s^{*j}|/100$

where R_s^{ij} is the system reliability of method i at the j th run and $R_s^{*j} = \max_i \{ R_s^{ij} \}$

4. Average relative error (R): $\frac{100}{\sum_{j=1}^{100}} |R_s^{ij} - R_s^{*j}| / (100 \times R_s^{*j})$.
5. Average execution time (T): average CPU time of 100 runs.

Table 2.2 summarizes the effects on the heuristic redundancy methods by relaxing the assumption of constant reliability. Data in each row of Table 2.2 are based on that particular experiment. In the case of constant component reliability, significant differences exist among the heuristic redundancy methods. The solution depends heavily upon the type of algorithm used. The quality of the result is proportional to the execution time required to obtain the answer. The simulation results are consistent over all criteria.

Having combined sequential search methods, the relative performance changed considerably. Using the G-A-G search method, G-A-G/G-A-G is the best for optimality rate. G-A-G/N-N is the best for maximum error rate, while G-A-G/K-I is the best for average absolute error. Using the H-J search method, the differences among G-A-G, N-N, and K-I heuristic redundancy methods are leveled out. Except for the fact that K-I has a higher maximum error rate, the results are quite consistent.

In Table 2.3, the results of eight combinations are compared. Data at each entry of Table 2.3 are the results compared over eight combinations of the method. Comparison of all criteria and all heuristic redundancy methods shows that the H-J search method is significantly better than the G-A-G search method. The differences among H-J/G-A-G, H-J/N-N, and H-J/K-I are not significant. It can be concluded that the search method is the dominant factor in solving this type of problems. The relative importance of the heuristic redundancy methods are leveled out when combined with the search methods.

The algorithms were coded in Fortran 77 and run on an IBM PC/AT with a mathematical co-processor. The computation was done in double precision to avoid round-off errors. Because of the structure of the system, a low component reliability will yield a high system reliability. Therefore, the absolute errors are small and the relative errors are closed to the corresponding absolute errors.

TABLE 2.2. Simulation results I

Heuristic Method		Base Point 1 (0.5) or $r_j = 0.7$			
Search Method		S-V	G-A-G	N-N	K-I
Constant r_j 's	O*	36	53	68	55
G-A-G	O	6	50	40	36
H-J	O	8	35	31	43
Constant r_j 's	M†	55	43	31	45
G-A-G	M	76	11	9	12
H-J	M	72	7	7	17
Constant r_j 's (%)	A‡	0.416	0.311	0.101	0.532
G-A-G (10^{-5})	A	45	11	12	10
H-J (10^{-5})	A	86	15	17	22
Constant r_j 's (sec)	T§	0.213	0.844	2.06	1.15
G-A-G (sec)	T	30.8	123.3	288.5	166.1
H-J (sec)	T	58.1	228.2	501.5	298.1

*O = optimality rate.

†M = maximum error rate.

‡A = average absolute error.

§T = execution time.

Base Point 2 (0.7) or $r_j = 0.85$				Average			
S-V	G-A-G	N-N	K-I	S-V	G-A-G	N-N	K-I
42	57	71	70	39	55	69.5	62.5
8	52	44	39	7	51	42	37.5
10	36	41	31	9	35.5	36	37
61	43	33	38	58	43	32	41.5
76	11	8	15	76	11	8.5	13.5
61	11	14	17	62.5	9	10.5	17
0.1	0.086	0.016	0.05	--	--	--	--
45	12	12	9	45	11.5	12	9.5
50	19	21	17	68	17	19	19.5
0.204	0.82	1.94	1.20	0.208	0.832	2.0	1.18
20.3	82.2	188.8	112	25.6	102.7	238.6	139
53.9	199.6	446.4	273.6	56	214	474	285.9

TABLE 2.3. Simulation results II

Heuristic Method		Base Point 1 (0.5)			
Search Method		S-V	G-A-G	N-N	K-I
G-A-G	0 [*]	2	10	7	6
H-J	0	5	30	28	37
G-A-G	M [†]	41	9	4	6
H-J	M	30	4	5	8
G-A-G (10 ⁻⁵)	A [‡]	62	27	29	26
H-J (10 ⁻⁵)	A	87	17	19	23

^{*}0 = optimality rate.
[†]M = maximum error rate.
[‡]A = average absolute error.

Base Point 2 (0.7)				Average			
S-V	G-A-G	N-N	K-I	S-V	G-A-G	N-N	K-I
0	12	5	7	1	11	6	6.5
7	32	36	24	6	31	32	30.5
43	7	4	9	42	8	4	7.5
25	8	6	6	27.5	6	5.5	7
61	27	27	24	61.5	27	28	25
52	21	23	18	68.5	19	22	20.1

REFERENCES

1. Aggarwal, K. K. "Redundancy optimization in general system." IEEE Transactions on Reliability, R-25, 1976, 330-332.
2. Aggarwal, K. K., J. S. Gupta, and K. B. Misra. "A new heuristic criterion for solving a redundancy optimization problem." IEEE Transactions on Reliability, R-24, 1975, 86-87.
3. Gopal, K., K. K. Aggarwal, and J. S. Gupta. "An improved algorithm for reliability optimization." IEEE Transactions on Reliability, R-27, 1978, 325-328.
4. Gopal, K., K. K. Aggarwal, and J. S. Gupta. "A new method for solving reliability optimization problem." IEEE Transactions on Reliability, R-29, 1980, 36-38.
5. Hooke, R. and T. A. Jeeves. "Direct search solution of numerical and statistical problems." J. Assoc. Comp., 8, 1961, 212-224.
6. Kohda, T. and K. Inoue. "A reliability optimization method for complex systems with the criterion local optimality." IEEE Transactions on Reliability, R-31, 1982, 109-111.
7. Kuo, W., C. L. Hwang, and F. A. Tillman. "A note on heuristic methods in optimal system reliability." IEEE Transactions on Reliability, R-27, 1978, 320-324.
8. Misra, K. B. "A simple approach for constrained redundancy optimization problem." IEEE Transactions on Reliability, R-21, 1972, 30-34.
9. Misra, K. B. and M. D. Ljubojevic. "Optimal reliability design of a system: a new look." IEEE Transactions on Reliability, R-22, 1973, 255-258.
10. Nakagawa, Y. and S. Miyazaki. "An experimental comparison of the heuristic methods for solving reliability optimization problems." IEEE Transactions on Reliability, R-30, 1981, 181-184.
11. Nakagawa, Y. and K. Nakashima. "A heuristic method for determining optimal reliability allocation." IEEE Transactions on Reliability, R-26, 1977, 156-161.

12. Sharma, J. and K. V. Venkateswaran. "A direct method for maximizing the system reliability." IEEE Transactions on Reliability, R-20, 1971, 256-259.
13. Tillman, F. A., C. L. Hwang, and W. Kuo. "Determining component reliability and redundancy for optimum system reliability." IEEE Transactions on Reliability, R-26, 1977, 162-165.
14. Tillman, F. A., C. L. Hwang, and W. Kuo. "Optimization Techniques for system reliability with redundancy -- a review." IEEE Transactions on Reliability, R-26, 1977, 148-155.
15. Tillman, F. A., C. L. Hwang, and W. Kuo. Optimization of Systems Reliability. Marcel Dekker, New York, 1985.

SECTION III. RELIABILITY OPTIMIZATION WITH THE LAGRANGE MULTIPLIER AND
BRANCH-AND-BOUND TECHNIQUES

INTRODUCTION

In the past two decades, numerous reliability optimization techniques have been proposed [12]. These techniques can be classified as the exact method and the iterative method. The exact method obtains the solution analytically. In general, it involves more mathematics and generate a more accurate solution. The Lagrange multiplier with Kuhn-Tucker conditions [8,9] and dynamic programming [12] are examples of the exact method. The iterative method obtains the solution by repeating the algorithm or enumerating the solutions. It does not require an extensive mathematical background. The branch-and-bound technique [10] and the heuristic method [13] are examples of the iterative method.

In most reliability optimization problems, the decision variables are the number of redundancies that are integer (integer programming or redundancy allocation problems), the component reliabilities that are real numbers (real programming or reliability allocation problems), or a combination of both (mixed-integer programming or reliability-redundancy allocation problems). In the methods that are based on differentiation, the decision variables must be continuous. Earlier studies treat the number of redundancies as real variables [8,9]. The real number answer is rounded off and the neighboring integer solutions are evaluated. The best feasible solution among the trials is taken as the final solution. This method works well if the problem is simple and the constraints are linear [9]. As the problem gets complicated,

the rounding off and trial-and-error procedure become inefficient and inaccurate. Furthermore, this approach provides no theoretical reasoning and has difficulties in extending the integer programming problem to the mixed-integer programming problem, which is frequently needed for reliability optimization.

Other methods treat the redundancy allocation problem as an integer allocation process from the very beginning. Heuristics are the popular techniques [12] but offer the users little feeling about optimality. In addition, it is both inefficient and difficult to justify the methods to solve the reliability-redundancy allocation problem. The combination method studied in Section II provides one of a few ways to optimize the reliability-redundancy allocation problem. Unfortunately, it suffers numerical instability and low computational efficiency.

A method combining the Lagrange multiplier technique with the branch-and-bound technique is proposed. The Lagrange multiplier technique quickly reaches an exact real number solution that is close to the optimal solution. Next, the branch-and-bound method is used to obtain the integer solution. This proposed method can solve both the redundancy allocation problem and the reliability-redundancy problem. When dealing with the latter problem, only branching and bounding the integer variable is necessary.

THE LAGRANGE MULTIPLIER AND KUHN-TUCKER CONDITIONS

Notation:

$[x_j]$	integer part of x_j
λ_i	the i th Lagrange multiplier
L	Lagrange multiplier function
$\bar{\lambda}$	$(\lambda_1, \dots, \lambda_M)$
g_i	the i th constraint.

The constrained reliability optimization problem can be formulated as follows:

$$\text{Max } R_s(\bar{X}, \bar{R})$$

subject to

$$g_i(\bar{X}, \bar{R}) \leq b_i \quad i=1, \dots, M. \quad (3.1)$$

The Lagrange multiplier technique transforms the constrained optimization problem into the unconstrained problem by introducing the Lagrange multipliers, λ_i 's. The unconstrained optimization problem, called the Lagrangian, becomes

$$\text{Max } L(\bar{X}, \bar{R}, \bar{\lambda}) = R_s(\bar{X}, \bar{R}) - \sum_{i=1}^M \lambda_i [g_i(\bar{X}, \bar{R}) - b_i] \quad (3.2)$$

$$\lambda_i \text{'s} \geq 0.$$

According to the Kuhn-Tucker conditions [7], the necessary conditions for a maximum to exist are

$$\frac{\partial L}{\partial r_j} = 0 \quad (3.3)$$

$$\frac{\partial L}{\partial x_j} = 0 \quad j=1, \dots, N \quad (3.4)$$

$$\lambda_i \frac{\partial L}{\partial \lambda_i} = \lambda_i [g_i(\bar{X}, \bar{R}) - b_i] = 0 \quad (3.5)$$

$$\lambda_i \geq 0 \quad (3.6)$$

$$g_i - b_i \leq 0 \quad i = 1, \dots, M \quad (3.7)$$

Equations (3.3), (3.4), and (3.5) form a system of $2N + M$ simultaneous equations. The solutions to these simultaneous equations subject to Eqs. (3.6) and (3.7) are extreme points in Eq. (3.1).

The nonlinear simultaneous equations can be solved by Newton's method, which expresses the multi-variable root-finding problem as follows [2].

$$\bar{X}_{k+1} = \bar{X}_k - VJ(\bar{X}_k)^{-1} F(\bar{X}_k) \quad (3.8)$$

where

\bar{X}_k \bar{X} at the k th iteration

V a positive scalar

$F(\bar{X})$ $(f_1(\bar{X}), \dots, f_N(\bar{X}))^T$

$J(\bar{X}_k)$ Jacobian matrix of $F(\bar{X}_k)$.

The scalar, V , controls the rate of convergence. If V is greater than one, the convergence is faster. If V is greater than zero and less than one, the convergence is slower. For the reliability optimization problem, the scalar, V , is taken to be less than one, since the upper and lower bounds of x_j and r_j are not imposed on the constraints. This conservative measure avoids x_j and r_j converging in an infeasible region.

Newton's method requires the evaluation of partial derivatives of the simultaneous equations. In some applications, the exact evaluation of the partial derivatives is inconvenient or even impossible. This difficulty can be overcome by using a finite difference approximation to the partial derivative [2], i.e.,

$$\frac{\partial f_i(\bar{X})}{\partial x_j} \approx \frac{f_i(\bar{X} + \bar{e}_j h) - f_i(\bar{X})}{h} \quad (3.9)$$

where h is a small value and \bar{e}_j is a vector with one at the j th element and zero elsewhere. Other methods such as the secant approximation to the derivative in Newton's method [14], i.e.,

$$f'(\bar{X}_k) = \frac{f(\bar{X}_k) - f(\bar{X}_{k-1})}{\bar{X}_k - \bar{X}_{k-1}} \quad (3.10)$$

and the quasi-Newton method [2] are popular ways of solving nonlinear simultaneous equations without having to evaluate partial derivatives

explicitly. Subroutines for solving nonlinear simultaneous equations are available in many mathematical libraries. Examples are ZSCNT and ZSPOW of IMSL [6], and ZONE of PORT mathematical library [11]. These subroutines are accurate, convenient, and efficient. However, they may not converge, and the solution may be infeasible. In this study, the ZSCNT subroutine was used to verify solutions obtained by Newton's method.

THE BRANCH-AND-BOUND TECHNIQUE IN INTEGER PROGRAMMING

The branch-and-bound technique of integer programming for reliability optimization is stated as follows [3]:

1. Solve the problem as if all the variables were real numbers. This solution is the upper bound (for maximization problem) of the integer programming problem.
2. Choose one variable at a time that has a noninteger value, say x_j , and branch that variable to the next higher integer value for one problem and to the next lower integer value for the other. The real value solution of the j th variable can be expressed as $x_j = [x_j] + x_j^*$, where $[x_j]$ is the integer part of x_j and $0 < x_j^* < 1$. The lower and upper bound constraints of the two mutually exclusive problems are $x_j \geq [x_j] + 1$ and $x_j \leq [x_j]$, respectively. Add these two constraints to both branched problems (called the process of the j th branch-and-bound). Solve both problems by the Lagrange multiplier method. Now the j th variable becomes an integer in either branch.
3. Fix the integer of x_j for the following steps of branch-and-bound. Select the branch that results in a higher system reliability. Then repeat step 2 using another variable $x_k \neq x_j$ for each of the new problems until all variables becomes integers.

4. Stop branching the problem if the solution is worse than the current best integer solution. Stop the iteration when all the desired integer variables are obtained.

In step 2, there are many criteria for selecting the variable for branching [4]. This paper selects the variable x_j that minimizes $\min(x_i^*, 1-x_i^*)$.

These steps can be directly applied to the mixed-integer programming problem. For mixed-integer programming problem, only the integer variables need to be enumerated by the branch-and-bound procedure. The real variables are free of restriction after each step of the branch-and-bound technique. Then by using the Lagrange multiplier technique, their new optimal values are obtained. Stop the branch-and-bound process whenever all the integer variables find integer values.

NUMERICAL EXAMPLES

Example 1

A four-stage series system with two linear constraints is formulated as a pure integer programming problem. The decision variables, $\bar{X} = (x_1, x_2, x_3, x_4)$, are the number of redundancies at each stage. The problem is formulated as

$$\text{Max } R_s(\bar{R}, \bar{X}) = \prod_{j=1}^4 [1 - (1 - r_j)^{x_j}]$$

subject to

$$\sum_{j=1}^4 c_{ij} x_j \leq b_i \quad i = 1, 2 \quad (3.11)$$

x_j 's ≥ 1 are integers.

Using the data given in Table 3.1, the real solution, $\bar{X} = (5.11672, 6.30536, 5.23536, 3.90151)$, was obtained using the Lagrange multiplier method and the Kuhn-Tucker conditions proposed by Misra [9]. By rounding the solution to the nearest integer, the solution becomes (5, 6, 5, 4). This paper suggests that the real solution be further elaborated by the branch-and-bound technique. As shown in Fig. 3.1, the final answer after the branch-and-bound process is also (5, 6, 5, 4), which is globally optimal. Newton's method was programmed in

Fortran and run on the NAS 9160. It took three seconds of CPU time to solve the problem. Even if both Misra's method [9] and this method draw the same conclusion about the decision variables, this method provides a logical reasoning in obtaining the solution.

TABLE 3.1. Data for example 1

Stage, j	1	2	3	4
r_j	0.80	0.70	0.75	0.85
c_{1j}	1.2	2.3	3.4	4.5
c_{2j}	5	4	8	7
$b_1 = 56$				
$b_2 = 120$				

Example 2

A five-stage series system with three nonlinear constraints is formulated as a mixed-integer programming problem. Both the number of redundancies, x_j , and the component reliability, r_j , are to be determined. The problem from Ref. [12] is as follows:

$$\text{Max } R_s(\bar{R}, \bar{X}) = \prod_{j=1}^5 [1 - (1 - r_j)^{x_j}]$$

subject to

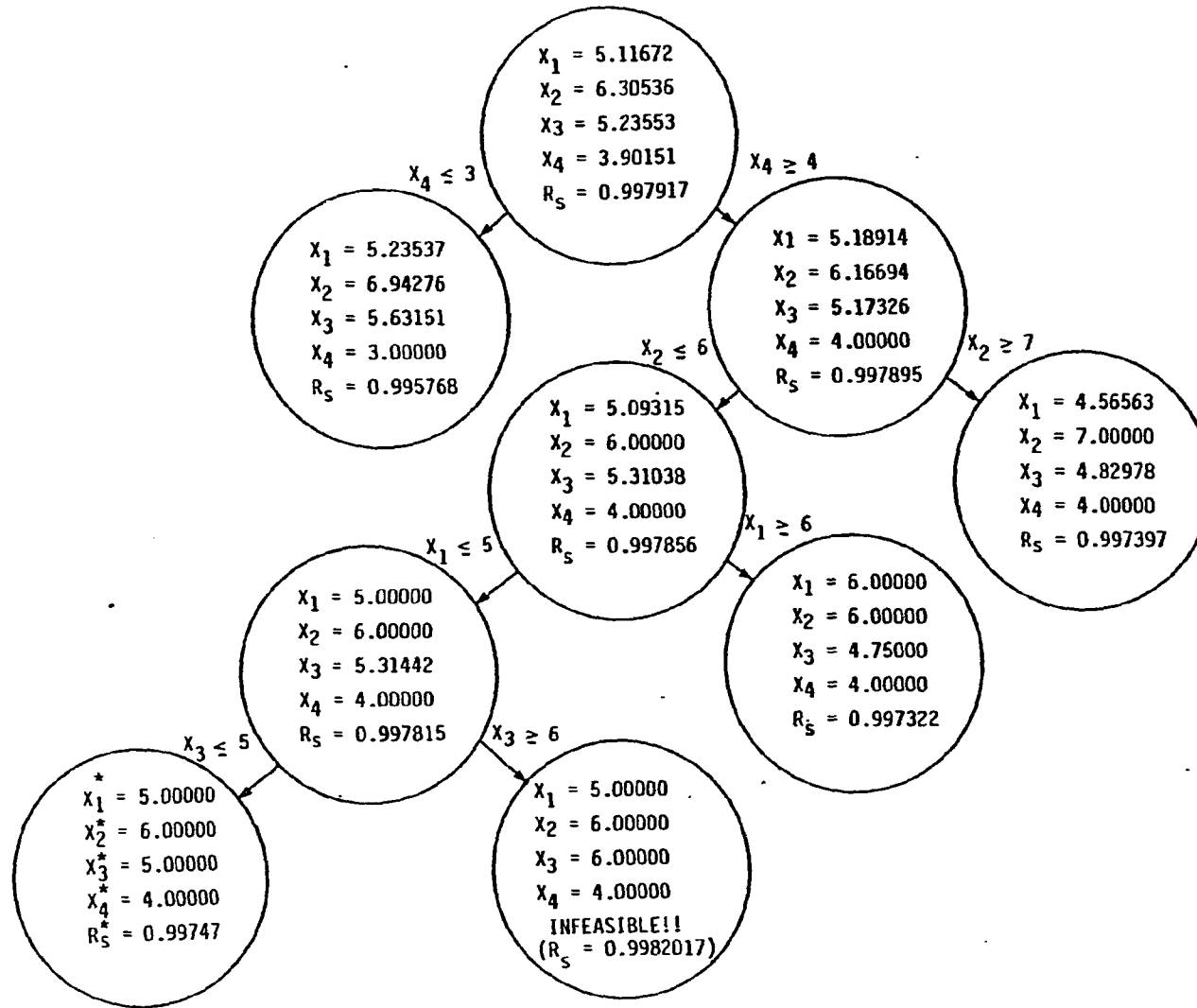


FIGURE 3.1. Branch-and-bound of example 1

$$g_1(\bar{X}) = \sum_{j=1}^5 p_j x_j^2 - P \leq 0$$

$$g_2(\bar{X}, \bar{R}) = \sum_{j=1}^5 a_j \left(-t / \ln r_j \right)^{\beta_j} (x_j + \exp(x_j/4)) - C \leq 0$$

$$g_3(\bar{X}) = \sum_{j=1}^5 w_j x_j \exp(x_j/4) - W \leq 0 \quad (3.12)$$

$$0 < r_j \text{'s} < 1$$

$x_j \text{'s} \geq 1$ are integers.

By taking the logarithm to the objective function, the Lagrangian can be written as

$$L(\bar{X}, \bar{R}, \bar{\lambda}) = \sum_{j=1}^N \ln \left[1 - (1-r_j)^{x_j} \right] - \sum_{i=1}^3 \lambda_i g_i(\bar{X}, \bar{R}). \quad (3.13)$$

The Kuhn-Tucker conditions to the problem are

$$\begin{aligned} \frac{\partial L}{\partial r_j} &= \frac{\ln q_j q_j^{x_j}}{1 - q_j^{x_j - 1}} - \lambda_2 a_j \left(\frac{-t}{\ln r_j} \right)^{\beta_j} \\ &\quad \cdot [1 + \exp(x_j/4)/4] - 2\lambda_1 x_j p_j - \\ &\quad \lambda_3 w_j \exp(x_j/4) (1 + x_j/4) = 0 \end{aligned} \quad (3.14)$$

$$\frac{\partial L}{\partial r_j} = \frac{x_j q_j^{x_j - 1}}{1 - q_j^{x_j - 1}} + \lambda_2 a_j \left(\frac{-t}{\ln r_j} \right)^{\beta_j} \cdot \beta_j$$

$$\cdot (x_j + \exp(x_j/4)) / (r_j \ln r_j) = 0 \quad j=1,2,\dots,5 \quad (3.15)$$

$$\lambda_i \frac{\partial L}{\partial \lambda_i} = \lambda_i g_i(\bar{X}, \bar{R}) = 0 \quad (3.16)$$

$$\lambda_i \geq 0 \quad (3.17)$$

$$g_i \leq 0 \quad i=1,2,3. \quad (3.18)$$

Using the data given in Table 3.2, the system of simultaneous equations in Eqs. (3.14), (3.15), and (3.16) was solved by Newton's method. After the real number solution was obtained, the branch-and-bound technique was used to find the integer variables while leaving the other variables free of restriction, except the previously investigated integer variables. The enumeration tree is shown in Fig. 3.2. Newton's method was programmed in Fortran and run on the NAS 9160. It took 16.2 seconds of CPU time to solve the problem.

TABLE 3.2. Data for example 2

j	a_j	P_j	W_j	P	C	W
1	2.33×10^{-5}	1	7			
2	1.45×10^{-5}	2	8			
3	5.41×10^{-6}	3	8	110	175	200
4	8.05×10^{-5}	4	6			
5	1.95×10^{-5}	2	9			
	$\beta_j = 1.5,$	$j = 1,2,3,4,5$		$t = 1000$		

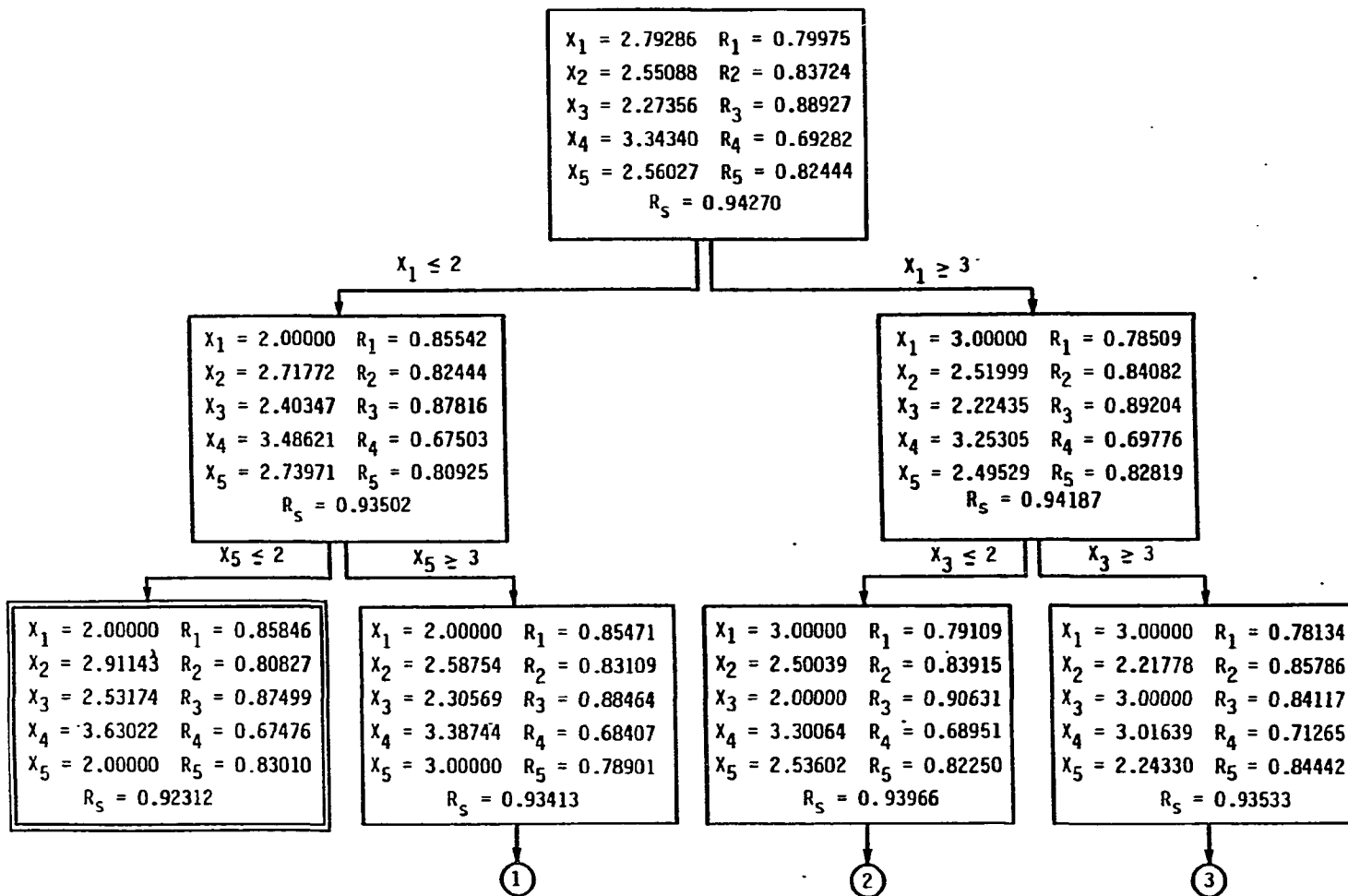


FIGURE 3.2. Branch-and-bound of example 2

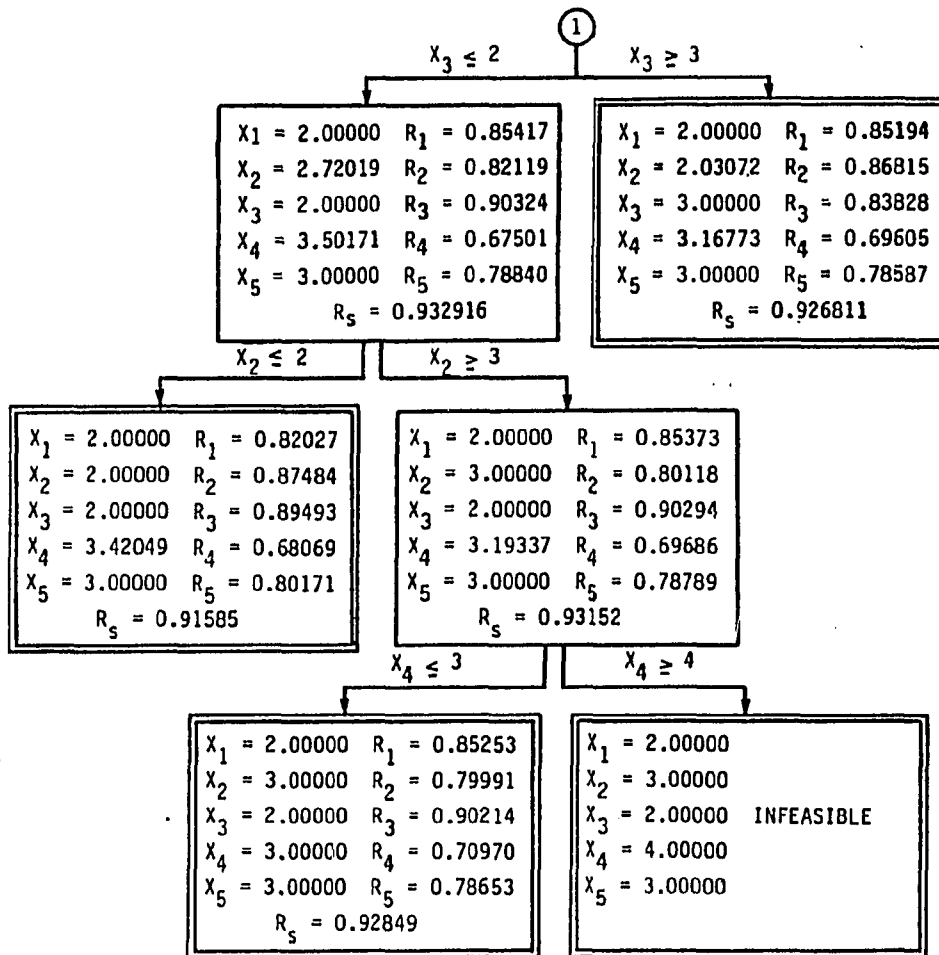


FIGURE 3.2 (Continued)

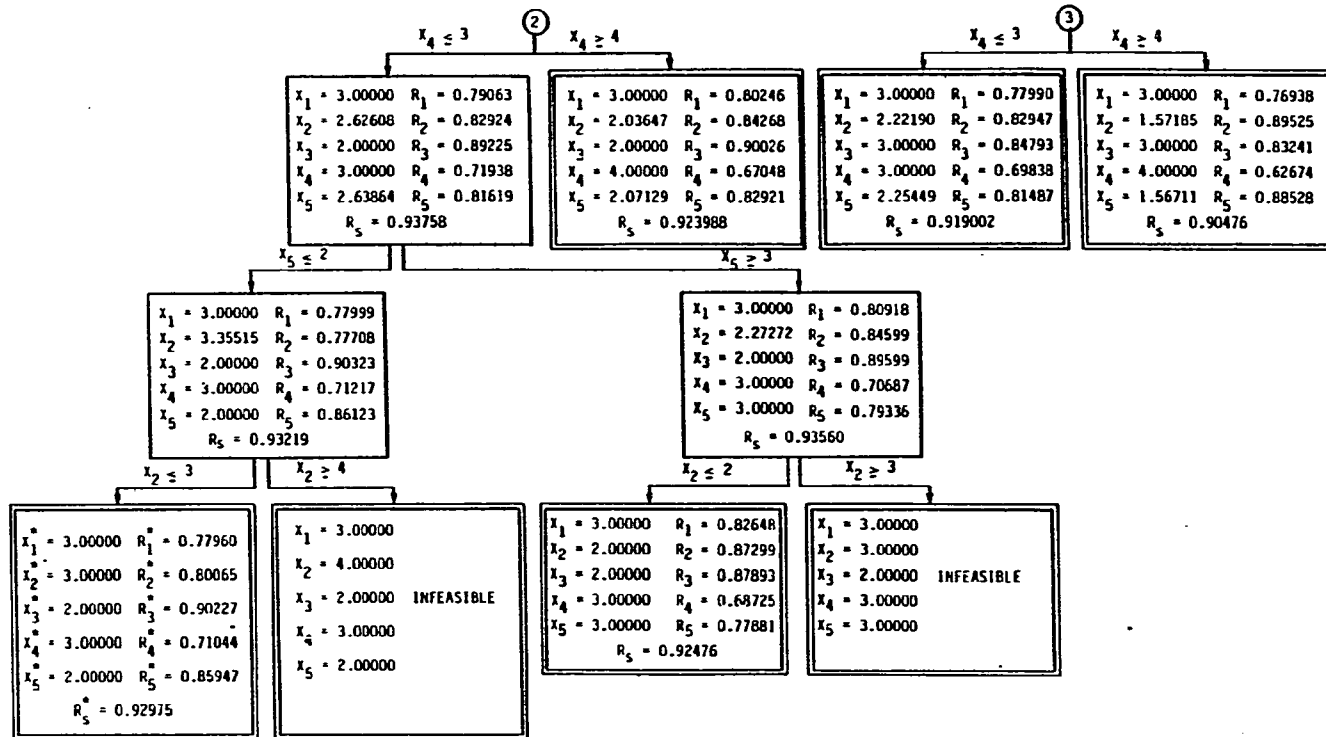


FIGURE 3.2 (Continued)

This same problem was solved using a combination of the sequential search and the heuristic redundancy allocation methods investigated in Section II. The results, summarized in Table 3.3, show that the proposed method is superior to the combination of the two iterative methods. A higher system reliability is obtained with less resource consumed. Experience show that this mixed-integer programming problem has many local optimums. The search technique discussed in Section II has the drawback of being trapped by a local optimum and not being able to get out of it. The proposed method overcomes this drawback and has been shown to be quite effective, especially for the mixed-integer programming problem.

TABLE 3.3. Comparison of two methods

	Lagrange Multiplier and the Branch-and- Bound Method	Hooke and Jeeves Pattern Search and Heuristic Method
\bar{X}	(3, 3, 2, 3, 2)	(3, 3, 2, 2, 3)
\bar{R}	(0.77960, 0.80065, 0.90227, 0.71044, 0.85947)	(0.7582, 0.8000, 0.9000, 0.8000, 0.7500)
R_s	0.92975	0.91494
g_1	27	28
g_2	0.00001	0.033727
g_3	10.57248	1.4118

CONCLUSION

The combination of the Lagrange multiplier and the branch-and-bound techniques takes advantage of the exact method and the enumerative method. The analytical method quickly reaches a solution that is close to optimum, and the enumerative method finds the integer solution. Since a good approximation is obtained by the former method, it does not take many iterations for the latter one to reach the optimal solution. In addition, the branch-and-bound method generates many sets of solutions. The competitive alternatives provide management with different options and flexibility. This general method can be applied to any twice differentiable constrained optimization problem. Nonlinear root-finding subroutines and numerical approximation can be used to eliminate the need of evaluating partial derivatives.

REFERENCES

1. Aggarwal, K. K., J.S. Gupta, and K. B. Misra. "A New Heuristic Criterion for Solving a Redundancy Optimization." IEEE Trans. on Reliability, R-24, No. 1, 1975, 86-87.
2. Burden, R. L. J. D. Faires, and A. C. Reynolds. Numerical Analysis. 2nd edition. Prindle, Weber and Schmidt, Boston, MA., 1981.
3. Garfinkel, R. S. and G. L. Nemhouse. Integer Programming. John Wiley & Sons, New York, 1972.
4. Gupta, O. K. and A. Ravindran. "Branch-and-bound experiments in convex nonlinear programming." Management Science, 31, No. 12, 1985, 1533-1546.
5. Hooke, R., and T. A. Jeeves. "Direct Search Solution of Numerical and Statistical Problems." J. Assoc. Comp. Math., 8, 1961, 212-224.
6. IMSL Library Reference Manual. International Mathematical and Statistical Libraries, Inc., Houston, Texas, 1984.
7. Kuhn, N. W. and A. W. Tucker. "Nonlinear programming." Proc. Second Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, CA, 1951.
8. Misra, K.B., and M.D. Ljubojevic. "Optimal Reliability Design of a System: a New Look." IEEE Trans. on Reliability, R-22, No. 4, 1973, 255-258.
9. Misra, K.B. "Reliability Optimization of a Series-parallel System." IEEE Trans. on Reliability, R-21, No. 4, 1972, 230-238.
10. Nakagawa, Y. K. Nakashima, and Y. Hattori. "Optimal reliability allocation by branch-and-bound technique." IEEE Trans. Reliability, R-27, No. 1, (1978), 31-38.
11. PORT Mathematical Subroutine Library. AT&T Bell Laboratories, Inc., Murray Hill, New Jersey, 1984.
12. Tillman, F.A., C.L. Hwang, and Way Kuo. Optimization of Reliability. Marcel Dekker, New York, 1985.

13. Tillman, F.A., C.L. Hwang, and Way Kuo. "Determining Component Reliability and Redundancy for Optimum System Reliability." IEEE Trans. on Reliability, R-26, No. 3, 1977, 162-165
14. Wolf, P. "The secant method for simultaneous nonlinear equations." Communications of the ACM, 12, 1959, 12-13.

SECTION IV. A REVIEW AND CLASSIFICATION OF SOFTWARE RELIABILITY MODELS

INTRODUCTION

Since the invention of the computer, computer software has gradually become an important part of a system. In the 1970s, the cost of software has surpassed the cost of hardware as being the major cost of a system [209]. In addition to the cost of developing a software, the penalty costs of software failures are even more significant. As missions accomplished by human beings are becoming 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 usually involves very high costs, human lives, and a social impact. Therefore, how to measure and predict the reliability of a software becomes an important issue.

In the past 15 years, more than 300 papers have been published in the areas of software reliability modeling, software reliability characteristics, and software reliability model validation. Since software is an interdisciplinary science, software reliability models are also developed from different perspectives of a software and different applications of the model. In order to pave the way for the future development and evaluation of highly reliability software and systems involving software and hardware, a detailed review of the existing software reliability models and the assumptions behind those models is of value. In this Section, a classification scheme for software reliability models is proposed. Software reliability models along with the characteristics of software and factors affecting software reliability are discussed.

CHARACTERISTICS OF SOFTWARE RELIABILITY MODELS

In hardware reliability, the mechanism of failure occurrence is treated as a black box. It's the failure process that is of interest to the reliability engineers. The emphasis is on the analysis of failure data and the design of experiment. 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 of how software is produced, how errors are introduced, how software is tested, how errors occur, the types of errors, and the environmental factors can help us in justifying the reasonableness of the assumptions, the usefulness of the model, and the applicability of the model under given user environment.

General description of software and software reliability, software life cycle, the bug-counting concept, hardware reliability versus software reliability, time index, error analysis, error size, user environment, and flowgraph representation of a program are discussed below.

General Description of Software and Software Reliability

Similar to the definition of hardware reliability, time-domain software reliability is defined as the probability of failure-free operation of a software for a specified period of time under specified conditions [209]. 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 mapping the input space to the output space ($P: I \rightarrow O$), 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.

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

It should be noted that "error" and "bug" are loosely used by many authors to represent fault and sometimes failure. Failure and fault customarily defined above [137] will be used through this section. Error and bug will also be used when the distinction between the two is not critical.

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 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 imperfect debugging assumption, i.e., faults may be removed, introduced, or no change at each debugging, the number of remaining faults may increase or decrease. This bug-counting process can be represented by the binomial model, Poisson model, compound Poisson process, Markov process, and doubly stochastic process.

Error Size

Error size of a fault is defined as the probability that an input state randomly selected from the input space will execute that fault and result in a failure [56]. It can be expressed in the following form.

$$S_i = \frac{1}{N} \sum_{j=1}^N e_{ij}$$

where

$$e_{ij} = \begin{cases} 1 & \text{if input state } j \text{ executes fault } i \text{ and fails} \\ 0 & \text{otherwise} \end{cases}$$

S_i error size of the i th fault
 N number of input states.

One hypothesis about error says that a large sized fault is easier to detect and will be detected earlier. A small sized fault is more

subtle and will be detected later. Although this hypothesis is hard to validate, the idea of nonidentical size of error conforms with the assumption of nonconstant per-fault failure rate postulated by many software reliability models.

User Environment

The reliability of a software is subject to the user environment. The failure rate of a Fortran compiler for instruction is expected to be lower than for sophisticated applications. Operational profile and system load are two environmental factors discussed below.

Operational profile is the distribution of input state execution. Depending upon the application, an input state could mean a typical transaction of daily operations, a partition of input space, or a combination of input variables. Since the relationship of input, fault, and failure is deterministic, how inputs are selected determines how failures occur. In other words, if the assumption says that faults are detected equal likely, it implies that input states are selected randomly. In testing, the test cases should be generated randomly according to the operational profile, so that the testing strategy will conform with the assumptions of the model. In the operational phase, some input states are executed more frequently than the others. This must be considered when evaluating the reliability of the software.

The system load consideration is derived from the phenomenon that a software is more likely to fail at peak hours than at the normal

operational hours [25,94]. In other words, the failure rate is not only a function of time (CPU time or operational time), but also a function of system load. This observation leads to a correction factor added to the software reliability model.

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" have been suggested as indices of the software reliability function.

Execution time - CPU time; time when the CPU is busy.

Operational time - Time the software is in use. This is usually referred to 8 working hours per day.

Calendar time - This index is used for software running 24 hours a day.

Run - A run is a job submitted to the CPU.

Instruction - Number of instructions executed.

Path - A path is the execution sequence of an input.

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

Although it may seem that software reliability models do not have a unified index, the unification can be achieved through unit conversion. For example, Musa et al. [155] have proposed methods of converting their execution time model to the calendar time model. Input-domain model can also be converted into time-domain model through a factor of "number of runs or paths executed per unit time."

Software Life Cycle

Software life cycle is normally divided into the 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. 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. This classification is based on the functional point of view rather than a strict time sequence. In reality, software life cycle phases overlap each other.

The factors governing the failures, the types of models applicable for reliability assessment, the purpose of reliability assessment, and the data available for parameter estimation vary from phase to phase [155,178]. In the early phase of 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, 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.

In addition to the relationship between software reliability model and software life cycle, the study of the type and percentage of errors introduced and removed within the software life cycle is also of interest to software reliability engineers.

Graph Representation of a Program

A program can very well be represented by a directed graph where decisions are the nodes, statements between two decisions is the arc, and execution sequence is the direction of the arc. This representation is also called the flowgraph of a program. Using flowgraph representation, the execution sequences of a program can be

traced through the paths of the flowgraph. In addition, the analysis of control flow and data flow of a flowgraph set the ground for many complexity metrics which, in turn, are used to estimate the number of errors in a program.

Another view of the flowgraph treats a program as a reliability network. Each node represents a module or a subroutine. As the reliability of each module and the transition probabilities among the modules are determined, the reliability of the program can be evaluated by the techniques of reliability network [7,33]. Some other graph properties like connectivity and reachability can also be applied to represent software properties.

Software Reliability versus Hardware Reliability

Since the emergence of software reliability, reliability theoreticians and practitioners have discussed the issue of software reliability versus hardware reliability in terms of similarity, differences, modeling techniques, etc. [85,217]. 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 4.1 lists the differences and similarities between the two.

TABLE 4.1. 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 stage is similar to the software debugging stage.
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 is treated as a black box.
CPU time and "run" are two popular indices for the reliability function.	Calendar time is a universally accepted index for the reliability function.
Most models are analytical models derived from assumptions. Emphasis is placed on the development of 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 the analysis of failure data.
Failures are caused by incorrect logic, incorrect statements, or incorrect input data. This is similar to the design errors of the complex hardware system.	Failures are caused by material deterioration, random failures, design errors, misuse, and environmental factors.
Failures are reproducible because the relationship between input state, program, and output is deterministic.	Failures are not reproducible.

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 provide information in the selection of testing strategies and the development of new methodologies. To facilitate our study, error analysis is studied by severity, error type, special errors, origination in the software life cycle, and uncovered destination in the software life cycle.

Classification by severity

In practice, it is often necessary to classify failures by their impact on the organization. As pointed out by Musa et al. [155], 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, and system breakdown could be a criterion for service impact.

To estimate the failure rate of each severity level, Musa et al. [155] suggest the following approaches.

1. Classify the failures and estimate failure rate separately for each class.
2. 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.
3. 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 [62].

Some 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 for too short a time to be isolated [209]. 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 [105]. This 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 the

redundant codes and not observed in the final output. When setting up the reliability objective, decisions must be made to either count the internal error or to simply count the observable errors.

Hardware caused software errors are errors if not carefully investigated will be regarded as a common software error [95]. For example, a program may be terminated during execution and receive an error message of operating system error. Without careful investigation, this error may be classified as software error while the operating system error was actually caused by the hardware. In software failure data collection, hardware caused software errors should be excluded from software errors.

Previously fixed errors are old errors which have happened before, but were not removed by debugging. Generated errors are new errors introduced by debugging [209]. These two types of errors conform with the assumption of imperfect debugging which allows errors to be introduced or no change in the fault count at each debugging.

Classification by the type of error

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. [239] includes the following error types.

- Computational errors
- Logical errors
- Input/output errors
- Data handling errors
- Operating system/system support errors

- Configuration errors
- Routine/routine interface errors
- Tape/processing interface errors
- User interface errors
- Data base interface errors
- User requested change
- Present data base errors
- Global variable/compool definition errors
- Recurrent errors
- Documentation errors
- Requirement compliance 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 Refs. [56,66].

Classification by error introduced in the software life cycle phase

Within the software life cycle, errors can be introduced in the following phases [20,239].

- Requirement and specification
- Design
 - Functional design
 - Logical design
- Coding
- Documentation
- Maintenance

For each phase, the frequency of occurrence can be obtained from failure data. It's recognized that errors introduced in the early phase of the software life cycle is more costly to remove [20].

Classification by error removed in the software life cycle phase

Errors are removed through testing which can be divided into the following stages [239].

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

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

Classification by the techniques of error removal

Some techniques of error removal given in Refs. [100,239] are summarized below.

- Automated requirement aids
- Functional specification review
- Simulation
- Design language
- Design standard
- Logic specification review
- Module logic inspection
- Module code inspection
- Code standards auditor
- Set/use analyzer
- Unit test
- Component test
- Subsystem test
- System test

This type of study gives us information in the selection and validation of software design and testing techniques.

CLASSIFICATION OF SOFTWARE RELIABILITY MODELS

Software reliability models can be classified into the deterministic model and the probabilistic model. The deterministic model studies 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, 3) the data flow of a program by studying the data sharing and data passing, and 4) other deterministic properties of a program.

Performance measures of the deterministic model are obtained by analyzing the program texture and do not involve any random event. The deterministic model can be further divided into software science, information content, software complexity, and software quality attributes. In general, these models empirically measure the qualitative attributes of a software and are used in the early phases of the software life cycle to predict the number of errors in a program or used in the maintenance phase for assessing and controlling the quality of a software.

The probabilistic model represents the failure occurrences and the fault removal as probabilistic events. It can be further divided into the error seeding model, curve fitting model, reliability growth model, execution path model, program structure model, input domain model, failure rate model, nonhomogeneous Poisson process model, Markov model, Bayesian model, and unified model.

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

The curve fitting model 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.

The reliability growth model 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.

The execution path model 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 program structure model views 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.

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

The failure rate model studies the functional forms of per-fault failure rate and the program failure rate at the failure intervals. Since mean-time-between-failure is the reciprocal of failure rate, models based on time-between-failure also belong to this category.

The Markov model 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-failure 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 rate of each fault.

The Bayesian model assume a prior distribution of the failure rate. This model is used when the software reliability engineer has a good feeling about the failure process and the failure data are rare. The unified model includes many models as special cases. Besides the continuous time discrete state Markov chain, the exponential order statistics [142], and the shock model [113] are two other general models.

The Deterministic Models

The deterministic model studies the elements of software and their interrelationship. It is also called software metrics or complexity metrics. With these metrics, programs can be measured and compared on the same basis. Software metrics are defined by analyzing the texture of the program or the flowgraph of the program rather than analyzing the failure process of the program as the probabilistic models do. These static models predict the number of errors in the program and do not involve time-dependent variables. Deterministic models are discussed below.

Software science

Developed by Halstead [77], software science defines software metrics based on the number of distinct operators and the number of distinct operands in a program. Program length, volume, effort, level, difficulty, mental discrimination, and moments are defined and related to program size, program development time, program development effort, and the number of errors in a program [63]. Among these metrics, program length and volume have been used to estimate the number of errors in a program.

Notation:

η_1	number of distinct operators
η_2	number of distinct operands
N_1	total number of operators
N_2	total number of operands
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 defines

$$N_1 = \eta_1 \log_2 \eta_1$$

$$N_2 = \eta_2 \log_2 \eta_2$$

$$N = N_1 + N_2$$

$$V = N \log_2(\eta_1 + \eta_2).$$

Previous studies have shown that a high correlation exists between the number of machine instructions and the number of errors in the program [209]. Since program length N is proportional to the number of machine instructions ($I=N/2$ if we assume that one machine instruction contains one operator and one operand), the number of errors in a program is also proportional to Halstead's program length. The relationship can be written as

$$(B \propto I) \wedge (I = N/2) \rightarrow B \propto N.$$

Halstead also derived a formula to estimate B from V . The formula is

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

Entropy function (information content)

The use of entropy function to estimate the number of errors in a program originates from Shannon's information theory [201].

Notation:

$X=(x_1, \dots, x_N)$	a set of messages
x_i	the i th message in X
p_i	probability of x_i
f_i	self-information of x_i
H	entropy of X

H_T	entropy of each token
H_P	entropy of the program
$I=(I_1, \dots, I_N)$	input space
I_i	the i th partition of I
NI_i	number of inputs in I_i
NI	total number of input
W	software work

Let $X=(x_1, \dots, x_N)$ be a set of messages from which a message is chosen. Then the self-information of any message, x_i , is defined as

$$f_i = -\log_2 p_i$$

If the probability of a message is 1, its self-information equals zero. If the probability of a message approaches 0, its self-information tends to infinity. The expected value of self-information is called the "entropy" (a measure of disorderness) or information content of that message and is defined as

$$H = -\sum_{i=1}^n p_i \log_2 p_i.$$

To set up an analogy between entropy of a set of messages and entropy of a program, we assume there is an entropy associated with each token (operator or operand) of the program, and program entropy is the sum of all the token entropies. Each token is a set of messages consisting of all the distinct operators and distinct operands. If the occurrence of each distinct operator and operand is equal likely, then

$$p_i = 1/(\eta_1 + \eta_2) \quad i=1, \dots, \eta_1 + \eta_2$$

$$f_i = -\log_2 p_i = \log_2(\eta_1 + \eta_2).$$

The entropy of each token is

$$H_T = \sum_{i=1}^{\eta_T} p_i f_i = \log_2(\eta_1 + \eta_2)$$

where

$$\eta_T = \eta_1 + \eta_2.$$

And the entropy of the program is

$$H_P = NH_T = N \log_2(\eta_1 + \eta_2) = V$$

where N and V are Halstead's program length and program volume, respectively. Since Shannon's program entropy is equal to Halstead's program volume, the formula of estimating the number of errors from program volume is also applicable to program entropy.

The idea of entropy metric can be applied to input classes as well as program tokens. Let the input space I of a program be partitioned into n classes, the entropy function of the program can be defined as [197]

$$H = \sum_{i=1}^n \frac{NI_i}{NI} \log_2 \frac{NI}{NI_i}.$$

Since a different design will result in a different partition of input space and a different entropy value, this entropy function can serve as a metric of measuring design complexity.

Another variation of entropy function called software work [88] is defined as

$$W = \sum_{i=1}^n NI_i \log_2 \frac{NI}{NI_i} .$$

Software quality attributes

The applicability of time-domain or input-domain software reliability models so far developed are limited to the testing phase. These models use failure rate or the number of remaining faults as a measure of software reliability. In the specification phase, design phase, and maintenance phase, the characteristics of a software can better be represented by software quality attributes rather than failure rate and the number of remaining errors. Although the correlation between software reliability and software quality attributes at a specific time point is difficult to be justified, they interact with each other in a long-term complicated manner. Poor quality attributes of today will lead to poor reliability in the future.

Software quality attributes include, but are not limited to, the ones listed below. They are grouped into the specification and design

phase, initial operation phase, revision phase, and transition phase. Depending upon the original authors, the definitions of these attributes may differ slightly and the meaning of two attributes may duplicate. The detailed definitions of these software quality attributes can be found in Refs. [21,24,140,148,249]. Software quality attributes from different sources are summarized as follows.

Initial operation phase

- Reliability
 - Correctness
 - Accuracy
 - Completeness
 - Integrity
 - Resilience
- Usability
 - Validity
 - Completeness
 - Documentation
- Efficiency
- Economy

Specification and design phase

- Modularity, structureness
- Clarity, conciseness
- Consistency, stability

Revision phase

- Maintainability
- Understandability
 - Clarity
 - Documentation
- Testability
 - Traceability
 - Accessibility

- Flexibility
 - Modifiability
 - Expandability

Transition phase

- Portability
- Reusability
- Modularity
- Interoperability

In addition to the descriptive definition, some software quality attributes have been expressed quantitatively. For example, consistency of requirement specification has been represented by a connectivity matrix and a reachability matrix [58], and maintainability has been represented by a connectivity matrix [185]. In addition, complexity metric is another quantitative way of representing software quality attributes. Although the correlation between software quality attributes and complexity metrics has not been widely studied, numerous complexity metrics have been suggested for their empirical relationship [21,216]. A detailed discussion of complexity metrics is given in the next section.

To measure and control the quality of a software, the software quality attributes and their highly correlated complexity metrics can be measured after specification and design phase, during operational phase, and after each major change in the maintenance phase. Anomalies reflected by the quality attributes can be identified and corrected. Since many attributes and complexity metrics are involved, a decision

table can be used to keep track of the conditions of each attribute and the actions to control the quality of the software. More work should be done in this area to find out quantitative metrics highly correlated to software quality attributes, and attributes highly correlated to reliability costs, resources, and productivity.

Complexity metrics

Complexity metrics in the context of software engineering is a measure of sophistication of a software program as opposed to the time complexity in algorithm analysis, which measures the running time as a function of problem size. The ultimate purposes of complexity are to 1) estimate the costs, resources, and time required to develop, test, and maintain a software, 2) measure the reliability of a software and the productivity of software development, and 3) serve as a quantitative representation of software quality attributes.

Although the relationship between complexity metrics and software quality, reliability, and productivity is empirical, complexity metrics have been widely used by practitioners because of their simplicity, intuition, and ease of automation. Once the program of measuring a complexity metric is written, this metric can be measured repeatedly with only the cost of computer time. Numerous complexity metrics have been proposed from the standpoint of program size, Halstead's software science, information content, data flow analysis, control flow analysis, and program syntax. In this review, only those metrics

related to reliability, error counting, and software quality attributes are discussed.

Lines of code Lines of code is the most widely used metric of estimating the number of errors in a program, the resources required to develop a program, and the productivity of programmers. Depending upon the authors, lines of code may mean the number of machine instructions, the number of executable source statements with or without data declaration, or the total number of source statements (including comments). It has been shown that the number of errors in a program is proportional to the size of the program. This linear relationship can be written as

$$B = KI$$

where

- B Number of errors in the program before debugging
- I Number of instructions
- K Constant of proportionality.

The value of K is about 0.02 error/machine instruction [209].

Program change Program change [49] is the textual change in the source code of a module during the development phase. It includes changes to statements, insertion of statements, and changes followed by the insertion of new statements. A program change represents a conceptual change to the program. It has been shown that a high

correlation exists between the total number of changes and the total error occurrences.

Takahashi and Kamayachi [236] studied the changes in program specification rather than textual changes. They also found a high correlation with the number of errors in the program.

Job step A job step is a programmer activity at the operating system command level [49]. Typical examples are editing texts, compiling source modules, link object modules, and executing entire program. This metric quantifies the frequency of computer system activities and can be used to estimate the requirements of computer resources, programmer's time, and programmer's efforts as well as software reliability.

Data binding Defined by Basili and Turner [12], a data binding occurs when a procedure/function P modifies a global variable X and a procedure/function Q access X. When the execution sequence of P proceeds that of Q, data binding denoted by (P,X,Q) occurs. A higher number of data binding increases the possibility of causing error when procedures/functions are changed.

Data span Data span is a measure of locality of data references. It is defined as the number of statements between two references to the same identifier with no intervening references to that identifier [55].

Cyclomatic number McCabe's cyclomatic number [139] originated from graph theory. The cyclomatic number $V(G)$ of a graph G with n nodes, e edges, and p connected components is

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

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 an unique entry node and an unique exit node, this flowgraph becomes a strongly connected graph if a dummy edge from the exit 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 number of each connected component. Another simple way of computing the cyclomatic number is as follows.

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

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 (GO TO statement is normally excluded from the

structured program). The cyclomatic number has been widely used in predicting the number of errors and as a measure of software quality.

Maximum intersection number In contrast to the cyclomatic number which measures the number of decisions, the maximum intersection number (MIN) proposed by Chen [31] measures the levels of nested decisions. MIN is obtained by cutting a strongly connected graph such that each region is entered exactly once. Given a program of n decisions, the upper bound of MIN is $n+1$ when n -level nested structure occurs, and the lower bound of MIN is 2 when none of the decisions is nested.

Knot count Knot count was suggested by Woodward et al. [251]. It measures the number of crossings of control flow in a program.

Calls and jumps An early experiment by Akiyama and Fumio [3] shows that the number of errors is proportional to the number of subroutine calls plus the number of jumps (decisions). A simplified metric of this type considers only the number of subroutine calls or the number of jumps.

Maintainability Haney [79] proposed a method of predicting maintainability by using a transition probability matrix. The expected number of changes at each module can be predicted from the initial number of changes of each module and a transition probability matrix of module change.

$$T = A(I + P + P^2 + \dots) = A(I - P)^{-1}$$

where

$P = (p_{ij})$	transition probability matrix of module changes
p_{ij}	probability of changing module i will result in changing module j
$A = (a_i)$	vector of initial changes
a_i	number of initial changes in module i
$T = (t_i)$	vector of total changes
t_i	expected number of changes in module i
I	identity matrix.

For a different design, the transition probability of module change, P , and the vector of total changes, T , are different. Given that P and A are available for alternative designs, T can be computed for each design and serves as a measure of maintainability.

By letting $a_i = 1$ for all i , a metric of design complexity is defined as [185]

$$m = \frac{1}{n} \sum_{i=1}^n (t_i - 1)$$

where

m design complexity

n matrix size.

Notice that the series $I + P + P^2 + \dots$ converges when the eigenvalue of P is greater than 0 and less than 1.

Accessibility Mohanty [148] defines the accessibility of a node as

$$A_{kl} = \sum_{ij} A_{ij} Q_{ijkl} P_{ij}$$

where

N_{ij} node ij ; the j th node of the i th level in the graph

A_{ij} accessibility of N_{ij}

P_{ij} probability of successfully executing N_{ij}

Q_{ijkl} probability of entering N_{kl} after executing N_{ij} .

Mohanty also suggests that P_{ij} can be estimated by

$$P_{ij} = k_p / C_{ij}$$

where

k_p constant of proportionality

C_{ij} some measure of complexity.

Since P_{ij} is the reliability of node N_{ij} , the complexity metric chosen must have a high correlation with reliability.

Testability Based on accessibility, Mohanty [148] further defines testability as

$$T_{ij} = A_{ij} P_{ij}$$

$$TP_i = \left[\sum_{S_i} (1/T_{ij}) \right]^{-1}$$

$$T = \begin{bmatrix} 1 & M & 1 \\ - & \Sigma & \\ M & i=1 & TP_i \end{bmatrix}^{-1}$$

where

T_{ij}	testability of N_{ij}
TP_i	testability of path i
T	testability of the program
S_i	set of node of path i
M	minimum number of paths in the program that cover all the nodes.

Testedness Also based on accessibility, Mohanty [148] defines testedness as

$$W_{ij} = 1 - \exp\left(-\frac{F_{ij}}{A_{ij}q_{ij}}\right)$$

$$W = \sum_S W_{ij} / |S|$$

where

W	testedness of the program
W_{ij}	testedness of node N_{ij}
$q_{ij}=1-p_{ij}$	unreliability of node N_{ij}
F_{ij}	number of times N_{ij} is executed
S	set of nodes in the program.

From the above formula, the testedness of node N_{ij} is an exponentially increasing function of the number of times N_{ij} is executed with rate $1/(A_{ij}-T_{ij})$, and bounded by 1. As F_{ij} approaches from 0 to infinity, W_{ij} increases from 0 to 1.

Program evolution The program evolution model proposed by Belady and Lehman [17] describes the phenomenon of continuing changes, continuing growth, and increasing entropy of a program after release. A complexity metric for module changes is defined as

$$C_R = MH_R/M_R$$

where

R release sequence number R
 M_R number of modules at release R
 MH_R number of modules handled in release interval R (I_R).

To predict C_R , two formulas have been suggested.

$$C_R = K_0 + K_1R + K_2R^2 + S + \epsilon$$

$$C_R = K_0 + K_1R + K_2R^2 + K_3HR_R + S + \epsilon$$

where

K_0, K_1, K_2, K_3 coefficients
 S cyclic component
 ϵ stochastic component; error
 I_R release interval R
 $HR_R = MH_R/I_R$ handle rate of release R .

Another complexity metric called fault class is defined as

$$C_i = 2^{i-1}$$

where C_i is the fault complexity at release i . At each release, the remaining faults are either faults generated at that release or residual faults. Therefore, the total number of combinations (fault classes) at release i is 2^{i-1} .

Schneider model Schneider [195] uses development effort in man-month and the number of subroutines to estimate the expected number of software problems. The empirical formula is given as

$$\begin{aligned} E(N) &= 7.6E^{0.667} S^{0.333} \\ &= K \left(\frac{S/K}{0.047} \right)^{1.667} \end{aligned}$$

where

$E(N)$	expected number of problems
E	efforts in man-month
S	number of subroutines
K	thousand of source codes
$E(N_r)$	expected remaining errors.

By assuming a ratio of 100:15 between detected errors and remaining errors, the author gives

$$E(N_r) = 0.15E(N).$$

Hybrid model The hybrid model uses more than one complexity metric discussed above to estimate the number of errors in the program. The types of complexity metrics included can be studied by regression analysis.

Environmental factors and error estimation Methods of error estimation discussed above are all based on complexity metrics. A different approach taken by Takahashi and Kamayachi [236] studies the correlation between error rate and environmental factors. They considered the type of program, the frequency of specification change (CHG), the average number of programmer experience, the difficulty of programming (DIF), the amount of programming effort (EFF), the level of programming technology, the volume of design documentation (DOC), and the percentage of reused modules. The authors have shown a close relationship between error rate and CHG, DIF, EFF, and DOC.

The Probabilistic Models

The probabilistic models treat software failures and errors removal as random events. They can be broken down into the error seeding model, curve fitting model, reliability growth model, execution path model, program structure model, input domain model, failure rate model, nonhomogeneous Poisson process model, and Markov chain. Among those, curve fitting model and reliability growth model are traditional techniques used in hardware reliability and other areas. The others were developed specifically for software.

The probabilistic model is the mainstream of software reliability study because it can be integrated with the hardware reliability theory. As systems are getting more and more complex, more will involve both hardware component and software component. This common framework makes it possible to evaluate the reliability of a hardware-software system.

Error seeding model

Originated from the idea of estimating the size of an animal population from recapture data [57], Mills [144] proposed an error seeding method to estimate the number of errors in a program by introducing pseudoerrors 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 of N can be shown to be

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

This method was criticized for the inability of determining the type, the location, and the difficulty level of the induced errors such that they will be detected equal likely as the indigenous errors. Basin [14] 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

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

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 [121] 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.

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

$$N \geq r-k \geq 0, n_1 \geq k \geq 0, \text{ and } m \geq r.$$

The interval estimate of N can be found in Huang [90] and Ramzan [180].

Reliability growth model

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 [42] summarizes 10 reliability growth models developed for hardware systems. This empirical approach is also adapted for predicting the progress of 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 failures
t	total time
λ_c	cumulative failure rate
a, β	parameters

The above formula shows that $\log \lambda_c$ is inversely proportional to $\log t$.

This model was adapted by Coutinho [36] 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 [42].

Weibull growth model Wall and Ferguson [247] proposed a model similar to the Weibull growth model for predicting the failure rate of a software during testing.

Notation:

$N(t)$	cumulative number of failures at time t
$M(t)$	maturity (man-month of testing, CPU time, calendar time, or number of tests)
M_0	scaling constant
N_0	parameters to be estimated
$\lambda(t)$	failure rate at time t
λ_0	initial failure rate; a constant
$G(t)$	$M(t)/M_0$

The model is summarized as follows:

$$N(t) = N_0 [G(t)]^\beta$$

$$\lambda(t) = N'(t) = N_0 G'(t) [G(t)]^{\beta-1}.$$

Let $N_0 G'(t) = \lambda_0$, then

$$\begin{aligned} \lambda(t) &= \lambda_0 [G'(t)]^{\beta-1} \\ &= \frac{\lambda_0}{\beta} \beta [G'(t)]^{\beta-1}. \end{aligned}$$

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 follows 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 Ref. [42].

Wall and Feguson tested this model on 6 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 [246] 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 i th time interval
$F(t)$	n_1/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 Ref. [246].

Logistic growth curve model

Suggested by Yamada and Osaki

[252], the logistic growth curve model has been used to represent the cumulative number of errors detected during debugging. The expected cumulative number of errors 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 [165] 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
- $N(0)$ number of corrections made before the first test interval is completed
- $N(t)$ cumulative number of errors corrected at time t
- A $N(0)/a$
- $\ln \gamma$ correction rate.

The above formula can be written as

$$\ln[\ln(N(t)/a)] = \ln[\ln(N(0)/a)] + t \ln \gamma$$

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 [42].

Hyperbolic reliability growth model Sukert [229] 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 following model.

Notation:

j	testing stage
R_j	reliability at the j th stage
γ	growth rate
R_∞	upper bound of the software reliability.

Then the reliability of the software at stage j is

$$R_j = R_\infty - \frac{\gamma}{j}$$

and the least squares estimates of R_∞ and a are in Ref. [125].

This model is a special case of a more general growth model for reliability improvement with a sequence of testing stages [254]. The model is

$$R_j = R_\infty - \gamma f(j).$$

By setting $f(j)=1/j$, the hyperbolic model is obtained.

Curve fitting model

The curve fitting model 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 The number of errors in a program can be estimated by a linear [9,176], or quadratic [93] 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 the i th error factors
 a_i, b_i coefficients.

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

Estimation of change Belady and Lehman [17] use time series analysis to study the program evolution process. Some of the models studied by them are

$$M_R = K_0 + K_1R + S + \epsilon$$

$$C_R = K_0 + K_1R + K_2R^2 + S + \epsilon$$

$$C_R = K_0 + K_1R + K_2R^2 + K_3HR_R + S + \epsilon$$

$$HR_R = K_1 + S + \epsilon$$

$$CMH_D = K_0 + K_1D + S + \epsilon$$

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, the operating system.

Estimation of time between failures Crow and Singpurwalla [38] argue that software failure may occur in clusters. Also addressed by Ramamoorthy and Bastani [178], 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 [38]. Data from two software projects were analyzed. Unfortunately, no statistical test was done to assess the adequacy of this model.

Estimation of failure rate Isotonic regression and exponential regression have been proposed to estimate the failure rate of a software.

Isotonic regression Given failure times t_1, \dots, t_n , a rough estimate of failure rate at the i th failure interval is

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

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

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

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 [76] and Miller and Sofer [143]. By imposing different assumptions to the problem, for example, monotonicity and convexity of the failure rate function, or equal spaced time intervals [143], the isotonic regression problem can be formulated into different forms.

Exponential regression Reported in Refs. [25,94], the failure of a program in the operational phase is a function of system load. This functional relationship has been studied by Butner and Iyer [25], using an exponential regression analysis. The probability of utilization-induced failure can be expressed as

$$P(u) = 1 - e^{-\gamma u}$$

where

$P(u)$ probability of utilization-induced failure

γ utilization-induced failure rate (failure/unit-paging²)

u utilization factor (unit-paging²).

Incorporating this function into a constant failure rate model,

$$\begin{aligned} F(t,u) &= 1 - e^{-\lambda t} e^{-\gamma u} \\ &= 1 - e^{-(\lambda t + \gamma u)} \end{aligned}$$

where $F(t,u)$ is the c.d.f. of time-between-failure in terms of time (CPU time or operational time) and system load.

Input-domain model

The input domain model uses "run" (input state) as the index of reliability function as opposed to "time" used by the time-domain model [169]. The basic input-domain model and an input-domain based stochastic model are discussed below.

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, software reliability is defined as the probability of successful run(s) randomly selected from the input space. Therefore, the reliability of one run 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

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

where

- I_i input state i
- p_i probability of running the i th input state
- F_I number of failures in N runs
- N number of 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.

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

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.

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

If a sequence of k runs are 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 [168]

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

where

$R(k)$ reliability over k runs

$R_j(1)$ $R(1)$ of the j th 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 [168] 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.

The values of f_i 's are given by Nelson for a quick estimation of the software reliability. For a partition i , the f_i value is

0.001	if more than one test case belongs to the partition
0.01	if only one test case belongs to the partition
0.05	if no test case belongs to the partition, but all segments and segment pairs executed by that partition have been exercised in the testing
0.1	same as above but not all segment pairs have been exercised in the testing
$0.1+0.2m$	if m segments ($1 \leq m \leq 4$) of that partition have not been exercised in the testing
1	if more than 4 segment of that partition has not been exercised in the testing.

Input-domain based stochastic model The input-domain based stochastic model was proposed by Ramamoorthy and Bastani [178]. 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.

Notation:

j	number of failures occurred
k	number of runs since the j th failure
$T_j(k)$	testing process for the k th run after the j th 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 k th run after j failures; $f(T_j(k))\lambda_j$
$R_j(k \lambda_j)$	probability that no failure occurs over k runs after j failures
$E_{\lambda_j}(\cdot)$	expectation over λ_j
Δ_j	size of the j th error
X	random variable that follows distribution F .

Then

$$\begin{aligned}
 R_j(k|\lambda_j) &= \prod_{i=1}^k [1 - V_j(i)] \\
 &= \prod_{i=1}^k [1 - f(T_j(i))\lambda_j]
 \end{aligned}$$

and

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

Assuming that the testing process is identical to the operational process,

$$f(T_j(k)) = 1$$

$$\lambda_j = v_j(k) \quad \text{for all } k$$

and

$$\Delta_j = \lambda_{j-1} - \lambda_j.$$

Further assume that

$$\Delta_j = \lambda_{j-1}X.$$

Hence,

$$\begin{aligned} R_j(k) &= E[(1 - \lambda_j)^k] \\ &= \sum_{i=1}^k \binom{k}{i} (-1)^i E[\lambda_j^i] \\ &= \sum_{i=1}^k \binom{k}{i} (-1)^i \{E[(1 - X)^i]\}^j. \end{aligned}$$

Other performance measures and parameters estimation can be found in Refs. [177,178].

Execution path model

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 [168,208] 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 [139]. The test cases are then selected from those paths. Since the relationship between input state, partition of input state, and path is not readily available, the execution path model is discussed separately from the input domain model.

Shooman decomposition model The decomposition model proposed by Shooman [208] 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

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

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

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

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

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

Program structure model

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 [130] represents the transitions between program modules during execution as a 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 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

$$\frac{\sum_j \pi_j P_{ij} (\mu'_{ij} \lambda_i + q_{ij})}{\sum_{i,j} \pi_i P_{ij} \mu'_{ij}}$$

$$= \sum_i a_i \lambda_i + \sum_{ij} b_{ij} \cdot q_{ij}$$

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 The Cheung's user-oriented software reliability model [33] estimates the reliability of a program 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 an 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 n th 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 $W(n,1)$

$$G^T = \begin{bmatrix} 0 & \dots & R_n \\ 1-R_1 & \dots & 1-R_n \end{bmatrix} \quad 2 \times n$$

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

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

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

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

Failure rate models

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 how failure rate changes at the failure time and the functional form of the failure rate during the failure intervals. Figure 4.1 shows a realization of failure process with failure times and failure intervals.

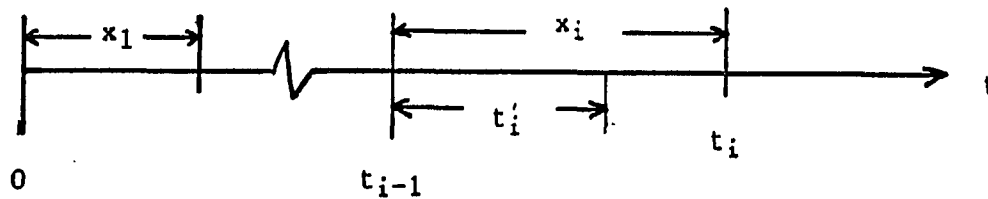


FIGURE 4.1. Failure process

The program failure rate during a failure interval is normally assumed to be dependent upon one or more of the following variables.

- number of remaining faults in the program
- failure rate of each fault

- time since the last failure
- debugging time
- number of testing stages
- probability of removing or introducing a fault at each debugging

Different assumptions lead to a different program failure rate and a different failure rate model. Once the program failure rate at the i th failure interval $\lambda(t'_i | t_{i-1})$, $0 \leq t'_i \leq x_i$ is determined, the i th failure time interval follows exponential distribution with rate $\lambda(t'_i | t_{i-1})$. In other words,

$$X_i \sim F_p(t'_i | t_{i-1}) = 1 - \exp\left[- \int_0^{t'_i} \lambda(s | t_{i-1}) ds\right]$$

where F_p is the c.d.f. of program time-between-failure. And the reliability function given that $i-1$ faults have been removed at time t_{i-1} is

$$R(t'_i | t_{i-1}) = \exp\left[- \int_0^{t'_i} \lambda(s | t_{i-1}) ds\right].$$

Most failure rate models belong to the binomial type model with the following assumptions.

1. The program contains N initial faults.
2. Each fault has the same c.d.f. of time to failure.
3. Whenever a failure occurs, a corresponding fault is removed with certainty.
4. The failure rate of the program is proportional to the number of faults remaining in the program.
5. Time spent in correcting the fault is negligible.

6. Faults are discovered independently.

Assumption 1 says that the number of initial faults is an unknown constant to be estimated. Assumption 2 means that each fault has the same failure rate or equivalently each fault has the same probability to be detected. Assumption 3 implies perfect debugging with which the number of failures occurred is equal to the number of faults removed. Assumption 4 establishes a linear relationship between program failure rate and the number of remaining faults. Assumptions 5 and 6 simplify the problem and make it workable. The binomial type model [155] lays the basis for more complex models. The above simplified assumptions will be relaxed gradually as this review proceeds.

The binomial type model treats the removal of faults as sampling without replacement from N initial faults, each having a time to failure distribution of $F(t)$. Let $X(t)$ be the number of failures occurred at time t , the probability of removing K faults at time t is

$$P_r\{X(t)=K\} = \binom{N}{K} [F(t)]^K [1 - F(t)]^{N-K}.$$

The c.d.f. of time to failure of each fault can be expressed in terms of the failure rate of each fault, $\phi(s)$.

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

The mean value function and variance of the failure process can be expressed as follows.

$$\mu(t) = E[X(t)] = NF(t)$$

and

$$\text{Var}\{X(t)\} = NF(t)[1 - F(t)].$$

By definition, the program failure rate is

$$\lambda(t) = \mu'(t) = Nf(t).$$

Let $V(t)$ be the remaining number of failures at time t ,

$$\begin{aligned} P_r\{V(t)=K\} &= P_r\{N - X(t) = K\} \\ &= P_r\{X(t) = N - K\} \\ &= \binom{N}{K} [F(t)]^{N-K} [1 - F(t)]^K \end{aligned}$$

and the expected number of remaining failures

$$\nu(t) = E[V(t)] = N[1 - F(t)].$$

Let T_i be the random variable of the i th failure time, the c.d.f. of T_i can be expressed as

$$\begin{aligned} P_r\{T_i \leq t\} &= P_r\{X(t) \geq i\} \\ &= \sum_{j=i}^N P_r\{X(t) = j\} \\ &= \sum_{j=i}^N \binom{N}{j} [F(t)]^j [1 - F(t)]^{N-j} \end{aligned}$$

and the c.d.f. of T_i given the $(i-1)$ th failure occurred at t_{i-1} is

$$\begin{aligned}
 P_r\{T_i > t_i | t_{i-1}\} &= [1 - F(t_i | t_{i-1})]^{N-i+1} \\
 &= \exp\left[-(N-i+1) \int_{t_{i-1}}^{t_i} \phi(s) ds\right].
 \end{aligned}$$

Finally, the conditional reliability function is

$$R(t'_i | t_{i-1}) = \exp\left[-(N-i+1) \int_{t_{i-1}}^{t'_i+t_{i-1}} \phi(s) ds\right]$$

and the program hazard rate is

$$Z(t'_i | t_{i-1}) = (N-i+1)\phi(t'_i+t_{i-1}).$$

By specifying a different per-fault failure rate function, a different class of binomial type model can be derived. Seven failure rate models are discussed below. It should be noted that not all of them follow exactly the assumptions postulated in the binomial model. The differences will be pointed out as needed. The following models list only the program failure rate or time-between-failure distribution. Other performance measures can be derived by following the procedure given in the binomial type model.

Jelinski and Moranda De-Eutrophication Model The Jelinski and Moranda De-Eutrophication model [96] is one of the earliest software reliability models. Although simple, it is the most often cited model. Many probabilistic software reliability models are either a variant or an extension of this basic model. By assuming a constant failure rate of each fault, the program failure rate at the i th failure interval is

$$\lambda(t'_i | t_{i-1}) = \phi[N - (i-1)], \quad 0 \leq t'_i < x_i$$

and the c.d.f. of the i th failure interval is

$$F_p(t_i' | t_{i-1}) = 1 - \exp\{-\phi[N - (i-1)]t_i'\}.$$

The reliability function is

$$R(t_i' | t_{i-1}) = \exp[-\phi(N-i+1)t_i'].$$

The above model was modified by Lipow [122] to allow more than one failure in a time interval. The failure rate at the i th time interval becomes

$$\lambda(t_i' | t_{i-1}) = (N - n_{i-1})\phi$$

where n_{i-1} is the number of failures occurred up to the $(i-1)$ th interval. In this formulation, the failure time can be interpreted as the debugging effort which may include more than one failure.

Extension of J-M model for varying program size The above J-M model assumes that the number of initial errors is an unknown constant. However, the integration testing is usually performed in a stepwise manner. Moranda [149] incorporates this changing program size debugging process into the original J-M model by further assuming that

1. the indigenous error is proportional to the number of statements under testing,
2. the number of statements at any time is known, and
3. the failure rate of each fault is unaffected when new statements are added.

Let

E_p errors per statement
 $G(t)$ number of statements at time t
 $X(t)$ number of failures occurred up to time t .

Then

$$\lambda(t'_i | t_{i-1}) = \phi[G(t)E_p - X(t'_i + t_{i-1})]$$

or

$$\lambda(t'_i | t_{i-1}) = \phi[G(t)E_p - (i-1)].$$

Jelinski-Moranda geometric De-Eutrophication model The J-M geometric De-Eutrophication model [150] assumes that the program failure rate decreases geometrically at failure times. Notice that this model deals with program failure rate rather than per-fault failure rate. The program failure rate and c.d.f. of time-between-failure at the i th failure interval can be expressed as

$$\lambda(t'_i | t_{i-1}) = \lambda_0 K^{i-1} \quad 0 \leq t'_i < x_i$$

and

$$F_p(t'_i | t_{i-1}) = 1 - \exp[-\lambda_0 K^{i-1} t'_i]$$

where

λ_0 initial program failure rate
 K parameter of geometric function ($0 < K < 1$).

A modified version of J-M geometric model was suggested by Lipow [122] to allow multiple error removal in a time interval. The program failure rate becomes

$$\lambda(t'_i | t_{i-1}) = \lambda_0 K^{n_{i-1}}$$

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

Moranda geometric Poisson model The Moranda geometric Poisson model [130] assumes that at fixed time T, 2T, ... of equal length interval, the number of failures occurred at interval i, n_i , follows a Poisson distribution with intensity rate $\lambda_0 K^{i-1}$. The probability of getting m failures at the ith interval is

$$P_r \{n_i = m\} = \frac{e^{-\lambda_0 K^{i-1}} (\lambda_0 K^{i-1})^m}{m!}$$

Schick and Wolverton model The Schick and Wolverton (S-W) model [193] 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. The program failure rate can be expressed as

$$\lambda(t'_i | t_{i-1}) = \phi [N - (i-1)] t'_i$$

A variation of the above model, also proposed by Schick and Wolverton [193], uses a parabolic function of time since the last debugging. The failure rate function becomes

$$\lambda(t'_i | t_{i-1}) = \phi [N - (i-1)] [at'_i{}^2 + bt'_i + c]$$

where a , b , and c are coefficients to be estimated.

Modified Schick and Wolverson model Sukert [229] modifies the S-W model to allow more than one failure at each time interval. The program failure rate becomes

$$\lambda(t'_i | t_{i-1}) = \phi[N - n_{i-1}]t'_i$$

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

Lipow [122] also modifies the S-W model by assuming that the program failure rate at the i th failure interval is a function of the $(i-1)$ th failure time and debugging time since the last failure. It can be expressed as

$$\lambda(t'_i | t_{i-1}) = \phi[N - n_{i-1}](t'_i/2 + t_{i-1}).$$

Goel and Okumoto imperfect debugging model Goel and Okumoto [72] 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 i th failure interval is

$$\lambda(t'_i | t_{i-1}) = \phi[N - p(i-1)].$$

According to the functional form of the per-fault failure rate, the failure rate models can be classified into the exponential class, Weibull class, C1 class, Pareto class, and others [155].

Nonhomogeneous Poisson process model

Based on the bug-counting concept, the nonhomogeneous Poisson process model (NHPP) represents the number of failures experienced up to time t as an NHPP, $\{X(t), t \geq 0\}$. 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's basic execution time model [164] and Goel and Okumoto NHPP model [70] belong to this class. Other types of mean value function suggested by Ohba [170] are the S-shaped models and hyperexponential model.

The NHPP model has the following assumptions [238].

1. The failure process has an independent increment, i.e., for any time points $t_0=0 < t_1 < \dots < t_n$, the process increments

$$X(t_1)-X(t_0), \dots, X(t_n)-X(t_{n-1})$$

are independent variables. Or equivalently, 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.

2. The failure rate of the process is

$$P_r\{X(t+\Delta t)-X(t) = 1\} = \lambda(t)\Delta t + o(\Delta t).$$

3. During a very short time interval Δt , the probability of more than one failures is negligible, i.e.,

$$P_r\{X(t+\Delta t)-X(t) > 1\} = o(\Delta t).$$

4. Initial condition is $X(0)=0$.

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

$$P_r\{X(t)=m\} = \frac{[\mu(t)]^m}{m!} e^{-\mu(t)}.$$

By definition, the mean value function of the cumulative number of failures can be expressed in terms of the failure rate of the program, i.e.,

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

And the expected number of initial faults is equal to the expected number of failures eventually experienced. The number of failures eventually experienced has a Poisson distribution with mean N_0 , i.e.,

$$E\{X(\infty)\} = \mu(\infty) = X_0 = E\{N(0)\} = N_0$$

and

$$P_r\{X(\infty)=k\} = \frac{e^{-N_0} N_0^k}{k!}.$$

The NHPP model treats $N(0)$ and $X(\infty)$ as random variables rather than constants as the binomial model does.

Due to the property of independent increment, the conditional probability can be derived as

$$\begin{aligned} P_r\{X(t) = n | X(t_i) = n_i\} &= P_r\{X(t) - X(t_i) = n - n_i\} \\ &= \frac{[\mu(t) - \mu(t_i)]^{n-n_i}}{(n - n_i)!} \exp\{-[\mu(t) - \mu(t_i)]\}. \end{aligned}$$

Also, define the distribution of the number of remaining faults as

$$\bar{X}(t) = X(\infty) - X(t).$$

Then

$$\begin{aligned} P_r\{\bar{X}(t) = k\} &= P_r\{X(\infty) - X(t) = k\} \\ &= \frac{[\mu(\infty) - \mu(t)]^k}{k!} \exp\{-[\mu(\infty) - \mu(t)]\}. \end{aligned}$$

And the c.d.f. of the i th failure interval can be expressed as

$$\begin{aligned} P_r\{T_i \leq t\} &= P_r\{X(t) \geq i\} \\ &= \sum_{j=i}^{\infty} \frac{[\mu(t)]^j}{j!} \exp[-\mu(t)]. \end{aligned}$$

Finally, the reliability function and the conditional reliability function of the program are

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

and

$$R(t_i' | t_{i-1}) = \exp\{-[\mu(t_i' + t_{i-1}) - \mu(t_{i-1})]\}.$$

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

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

A special case of the exponential class NHPP model is to let

$$\phi(t) = \phi.$$

Then

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

$$F(t) = 1 - e^{-\phi t},$$

and

$$R(t_i' | t_{i-1}) = \exp\{-N[F(t_{i-1} + t_i') - F(t_{i-1})]\}.$$

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

Musa exponential model Musa exponential model [164] can be summarized as

$$\phi(t) = \phi$$

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

and

$$\begin{aligned} \lambda(t) &= X_0 \phi B e^{-\phi B t} \\ &= \phi B [X_0 - \mu(t)]. \end{aligned}$$

Goel and Okumoto NHPP model The Goel-Okumoto model [70] has mean value function of

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

and

$$\lambda(t) = N\phi e^{-\phi t}.$$

An extension of the exponential mean value function model has been suggested by Yamada and Osaki [252]. They assume that faults comes 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

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

S-shaped growth model Most bug-counting models assume that each fault has the same probability to be detected. This assumption of independency in failure occurrence leads to an exponential growth of the cumulative number of failures. Ohba [170] observed an S-shaped growth which he claimed is due to the mutual dependency of faults. He argues that the detection of a fault will lead to the detection of its dependent faults. Therefore, in the early stage of debugging, as faults are detected, more dependent faults become detectable. This results in an increasing growth rate. As undetected faults decrease, the growth rate slows down gradually and finally approaches zero. Two types of S-shaped growth models, the delayed S-shaped growth model and the inflection S-shaped growth model have been proposed.

Delayed S-shaped growth model The delayed S-shaped model [170,256] 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

$h(t)$	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

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

This function becomes the mean value function of the NHPP model. Other performance measures can be derived following the procedure discussed in the NHPP model.

Based on the assumptions, the above model is not appropriate when 1) the time delay between fault detection and fault removal is negligible, 2) the effort spent in failure detection and failure removal is not constant, and 3) new faults are generated during the debugging process.

Inflection S-shaped growth model Ohba [170] models the dependency of faults by postulating the following assumptions.

1. Some of the faults are not detectable before some other faults are removed.
2. The detection rate is proportional to the number of detectable faults in the program.

3. Failure rate of each detectable fault is constant and identical.
4. All faults can be removed.

Then, the program failure rate during the i th 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 have been removed and $\mu_i [N - (i-1)]$ is the number of detectable faults at the i th 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 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 .

Markov chain

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 process. 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 [155].

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 [231]. In practice, there were not enough failure data to estimate all the parameters of the transition probability matrix. Some models have been developed as special cases of Markov chain. They are the stationary linear death model with perfect debugging, stationary linear death model with imperfect debugging, nonstationary linear death model with imperfect debugging, and the nonstationary linear birth-and-death model. These models are discussed below.

Linear death model with perfect debugging The Jelinski and Moranda model [96] 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

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

And the transition rate diagram is shown in Fig. 4.2.

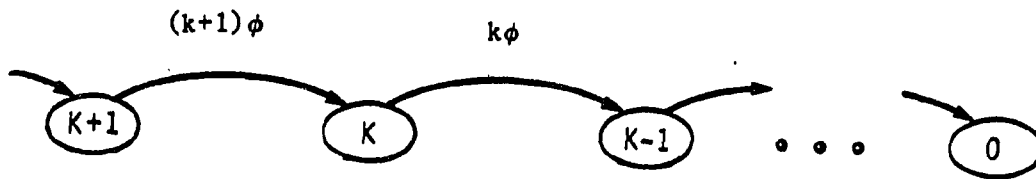


FIGURE 4.2. Linear death with perfect debugging

The differential-difference equation of $P_k(t)$ 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 [71,72], the transition probabilities of the linear death model with imperfect debugging can be expressed as

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

where p is the probability of successful debugging. And the transition rate diagram is shown in Fig. 4.3.

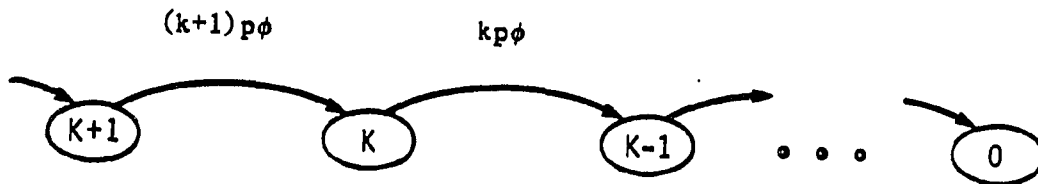


FIGURE 4.3. Linear death with imperfect debugging

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

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

The expected number of imperfect debugging errors by time t is

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

Reliability function of the k th 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 that [71]

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

Nonstationary linear death model with perfect debugging

Suggested by Shanthikumar [203,204], the transition probabilities of the nonstationary linear death model with perfect debugging can be expressed as

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

and the transition rate diagram is shown in Fig. 4.4.

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 $N(0)=N$,

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

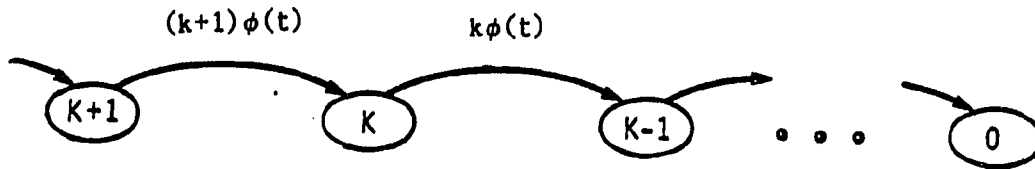


FIGURE 4.4. Nonstationary linear death with perfect debugging

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 that section.

Nonstationary linear birth-and-death model The adaptation of nonstationary linear birth-and-death process was given by Kuo [111] and Kremer [106]. At each debugging, a fault was removed with probability p , a fault was introduced with probability q , and no change with probability $1-p-q$. Kuo approaches the problem using a compound Poisson model while Kremer starts with a Markov chain. However, both approaches lead to the same conclusion.

The transition probabilities can be expressed as

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

and the transition rate diagram is shown in Fig. 4.5.

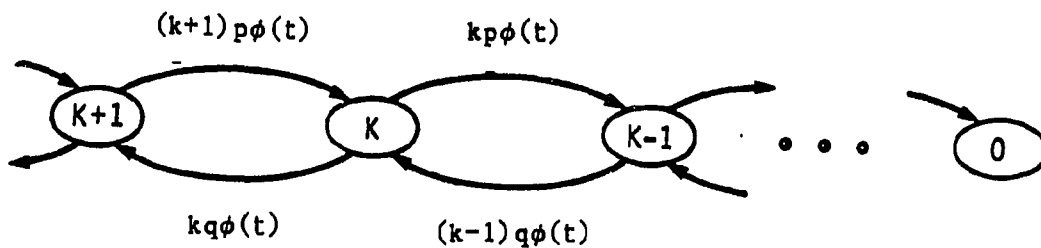


FIGURE 4.5. Nonstationary birth-and-death

Hence, the differential-difference equation for the above process is

$$P_k'(t) = (k+1)p\phi(t)P_{k+1}(t) + (k-1)q\phi(t)P_{k-1}(t) - k\phi(t)(p+q)P_k(t)$$

with initial condition

$$P_N(0) = 1.$$

The mean value function of $N(t)$ is derived in Section V. The state probabilities and performance measures can be found in Refs. [106,111].

The random variable of this Markov process is the number of remaining faults. Similarly, the number of transitions or the number of failures experienced can also be represented as a Markov process. During the debugging process, keeping track of the number of failures experienced is more practical than keeping track of the number of faults remaining, since the number of remaining faults is normally unknown without further estimation. Combining the two processes, it becomes a bivariate Markov process. The transition rate diagram of this bivariate process is shown in Fig. 4.6.

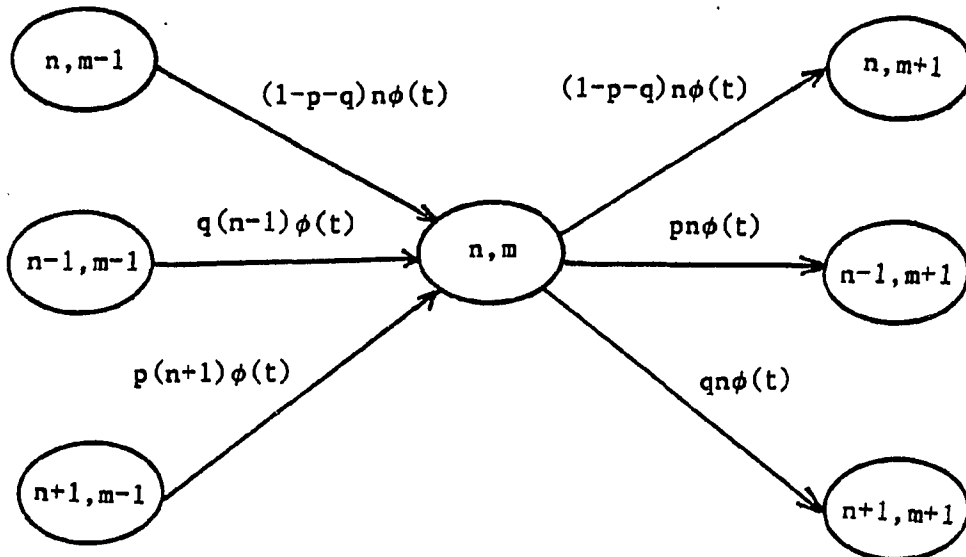


FIGURE 4.6. Bivariate process of fault-count and failure count

The differential-difference equation of the above process is

$$P'_{n,m}(t) = (1-p-q)n\phi(t)P_{n,m-1}(t) + q(n-1)\phi(t)P_{n-1,m-1}(t) + \\ p(n+1)\phi(t)P_{n+1,m-1}(t) - n\phi(t)P_{n,m}.$$

where $P_{n,m}(t)$ is the probability of n remaining faults and m failures occurred at time t .

Performance measures of this bivariate Markov model can be found in Kremer [106] and Kuo [111].

Other types of probabilistic models

The Bayesian model and unified model are two other types of probabilistic software reliability models. The Bayesian approach has been discussed by Jewell [97, Serra and Barlow(200)], Kuo [111], Littlewood [124,125], Littlewood and Verrall [136], and Langberg and Singpurwalla [113]. Besides the nonstationary birth-and-death model [106,111], other unified models are the exponential order statistics model by Miller [142] and Scholz [198] and the shock model by Langberg and Singpurwalla [113].

This review classifies the software reliability models mainly by the modeling techniques. Other types of classification, for instance, by the usage in software life cycle phases or by the types of applications, can also be investigated. Table 4.2 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, systems design,

reliability cost model, hardware-software system, and project management are areas which software reliability models can be applied.

TABLE 4.2. Summary of References

Models	References
General Software Reliability Models	1, 8, 30, 39, 41, 43, 48, 67, 74, 75, 84, 87, 91, 92, 105, 116, 117, 129, 137, 155, 159, 183, 189, 202, 209, 218, 229, 234, 239, 245, 254.
Error Analysis	3, 56, 66, 95, 153, 154, 181, 190, 243, 244.
System Load Effects	25, 27, 94, 188, 244.
Software Science	63, 77, 118, 195, 206, 207.
Software Quality Attributes	21, 24, 140, 175, 184, 249.
Complexity Metrics	9, 10, 11, 12, 13, 31, 49, 58, 64, 93, 100, 119, 139, 148, 176, 184, 185, 201, 206, 216, 236, 243, 248, 251, 257.
Error Seeding Models	10, 14, 51, 90, 180, 193, 194, 242.
Reliability Growth Models	36, 37, 42, 134, 165, 247.
Curve Fitting Models	17, 18, 25, 38, 76, 81, 143, 215, 236.
Input Domain Models	15, 168, 169, 177, 178, 217, 219, 224, 235, 250.
Execution Paths Models	46, 47, 50, 208.
Program Structure Models	7, 33, 130, 133, 196, 232, 233.
Failure Rate Models	4, 10, 29, 40, 44, 68, 80, 96, 98, 99, 104, 123, 124, 126, 127, 128, 131, 135, 149, 150, 151, 166, 167, 204, 211, 221.
NHPP Models	32, 70, 78, 111, 145, 156, 157, 162, 164, 170, 171, 187, 222, 22, 254, 255, 256.
Markov Chain Models	71, 72, 106, 114, 132, 203, 213, 231.
Bayesian Models	2, 97, 124, 125, 136, 192, 240, 241.

TABLE 4.2. (Continued)

Models	References
Other Unified Models	7,103,113,142,198.
Model Validation	4,5,147,162,182,199,225,226,227,228.
Cost Models and Stopping Rule	26,45,59,60,62,69,102,107,110,119,161,172,173,186,205,253.
Software Management	20,22,23,52,79,88,89,108,109,146,160,161,163,179,185,209,214,220,223.
Hardware-Software Systems	27,73,82,83,85,101,112,217,230,240,241.
Fault Tolerant Systems	16,19,26,28,35,53,86,138,141,191,237,257.

REFERENCES

1. Abdel-Ghaly, A. A., P. Y. Chan, and B. Littlewood. "Evaluation of computing software reliability predictions." IEEE Trans. Software Engineering, SE-12, No. 9, 1986, 950-967.
2. Adams, E. N. "Optimizing preventive service of software product." IBM Journal of Research and Development, 28, No. 1, 1984.
3. Akiyama and Fumio. "An example of software system debugging." IFIP Congress, 1971.
4. Angus, J. E. "The application of software reliability models to a major CCCI system." Proc. Annual Reliability and Maintainability Symposium, 1984.
5. Angus, J. E. "Software reliability model validation." Proc. Annual Reliability and Maintainability Symposium, 1980.
6. Bailey, N. T. J. The Element of Stochastic Process: John Wiley & Sons, New York, 1964.
7. Barlow, R. E. and N. D. Singpurwalla. "Assessing the reliability of computer software and computer networks: an opportunity for partnership with computer scientists." American Statistician, 39, No. 2, 1985, 88-94.
8. Basili, V. R. and 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.
9. Basili, V. R., R. W. Selby, and T. Y. Phillips. "Metric analysis and data validation across Fortran projects." IEEE Trans. Software Engineering, SE-9, No. 6, 1983, 652-663.
10. Basili, V. R. and D. H. Hutchens. "An empirical study of a syntactic complexity family." IEEE Trans. Software Engineering, SE-9, No. 6, 1983, 664-672.
11. Basili, V. R. and R. W. Reiter. "Evaluating automatable measure of software development." Proc. Workshop on Quantitative Software Models, 1979, 107-116.
12. Basili, V. R. and A. J. Turner. "Iterative Enhancement: a practical technique for software development." IEEE Trans. Software Engineering, SE-1, 1985, 390-396.

13. Basili, V. R. and B. T. Perricone. "Software errors and complexity: an empirical investigation," Communications of the ACM, 27, No. 1, 1984 42-45.
14. Basin, S. L. Estimation of software error rates via capture-recapture sampling. Science Application, Inc., Palo Alto, CA., 1973.
15. Bastani, F. B. An Input Domain Based Theory of Software Reliability and Its Application. Ph.D. dissertation, University of California, Berkeley, 1980.
16. Beaudry, M. D. "Performance related reliability measures for computing systems." Proc. Intl. Conf. on Fault-Tolerant Computing, 1977, 16-21.
17. Belady, L. A. and M. M. Lehman. "A model of large development." IBM Systems Journal, 15, No. 3, 1976, 225-252.
18. 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.
19. Bhargava, B. "Software reliability in real-time systems." Proc. COMPCON, 1981, 297-309.
20. Boehm, B. W. Software Engineering Economics. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1981.
21. Boehm, B. W., J. R. Brown, and M. Lipow. "Quantitative evaluation of software quality." Proc. Second Int'l Conf. on Software Engineering, 1976, 592-605.
22. Boehm, B. W. "Software and its impact: a quantitative assessment," Datamation, 19, No. 5, 48-59.
23. Brooks, F. P. Jr. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975.
24. Buckley, F. J. and R. Poston. "Software Quality Assurance." IEEE Trans. Software Engineering, SE-10, No. 1, 1984, 36-41.
25. Butner, S. E. and R. K. Iyer. "A statistical study of reliability and system load at SLAC." Proc. Int'l Conf. on Fault-Tolerant Computing, 1980, 207-209.
26. Caspi, P. A. and E. F. Kouka. "Stopping rules for a debugging process based on different software reliability models." Proc. Int'l Conf. on Fault-Tolerant Computing, 1984, 114-119.

27. Castillo, X. and Siewiorek. "A workload dependent software reliability prediction model." Proc. Int'l Conf. on Fault-Tolerant Computing, 1982, 279-286.
28. Castillo, X. and Siewiorek. "A performance-reliability model for computing systems." Proc. Int'l Conf. on Fault-Tolerant Computing, 1980, 187-192.
29. Catuneanu, V. and A. Mihalache. "Improving the accuracy of the Littlewood-Verrall model." IEEE Trans. Reliability, R-34, No. 5, 1985, 418-421.
30. Cavano, J. P. "Toward high confidence software." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1449-1455.
31. Chen, E. T. "Program complexity and program productivity." IEEE Trans. Software Engineering, SE-4, No. 2, 1978, 187-194.
32. Chenoweth, H. B. "Modified Musa theoretic software reliability." Proc. Annual Reliability and Maintainability Symposium, 1981, 353-356.
33. Cheung, R. C. "An user-oriented software reliability model." IEEE Trans. Software Engineering, SE-6, No. 2, 1980, 118-125.
34. Chiang, C. L. Introduction to Stochastic Processes in Biostatistics. Wiley, New York, 1968.
35. Cifersky, J. "Generalized Markov model for reliability evaluation of functionally degradable systems." Proc. Int'l Conf. on Fault-Tolerant Computing, 1982, 275-278.
36. Coutinho, J. S. "Software reliability growth." Proc. Int'l Conf. on Reliable Software, 1973, 58-64.
37. Crow, L. H. "Methods for assessing reliability growth potential." Proc. Annual Reliability and Maintainability Symposium, 1984, 484-489.
38. Crow, L. H. and N. D. Singpurwalla. "An empirically developed Fourier series model for describing software failure." IEEE Trans. Reliability, R-33, No. 2, 1984, 176-183.
39. Culpepper, L. M. "A system for reliable engineering software." IEEE Trans. Software Engineering, SE-1, No. 2, 1975, 174-178.
40. Currit, P. A., M. Dyer, and H. D. Miller. "Certifying the reliability of software." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 3-11.

41. Daniels, B. K. "Software reliability." Reliability Engineering, 4, 1983, 199-234.
42. Dhillon, B. S. Reliability Engineering in Systems Design and Operation. Van Nostrand Reinhold Co., New York, 1983.
43. Dhillon, B. S. "Software reliability - bibliography." Microelectronics and Reliability, 22, No. 3, 1982, 625-640.
44. Dickson, J. C. "Quantitative analysis of software reliability." Proc. Annual Reliability and Maintainability, 1972, 148-157.
45. Donelson, J. III. "Cost model for testing program based on nonhomogeneous Poisson failure model." IEEE Trans. Reliability, R-26, No. 3, 1977, 189-194.
46. Downs, T. "Extension to an approach to the modeling of software testing with some performance comparisons." IEEE Trans. Software Engineering, SE-12, No. 9, 1986, 979-987.
47. Downs, T. "An approach to the modeling of software testing with some applications." IEEE Trans. Software Engineering, SE-11, No. 4, 1985, 375-386.
48. Dunham, J. R. "Experiments in software reliability: life-critical applications." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 110-123.
49. Dunsmore, H. E. and J. D. Gannon. "Experimental investigation of programming complexity." Proc. ACM/NBS 16th Annual Technical Symposium: Systems and software, 1977, 117-125.
50. Duran, J. W. and J. Wiorkowski. "Quantifying software validation by sampling." IEEE Trans. Reliability, R-29, No. 2, 1980, 141-144.
51. Duran, J. W. and J. Wiorkowski. "Capture-recapture sampling for estimating software error content." IEEE Trans. Software Engineering, SE-7, No. 1, 1981, 147-148.
52. Duvall, L. "Data needs for software reliability modeling." Proc. Annual Reliability and Maintainability Symposium, 1980, 200-208.
53. Eckhardt, D. E. Jr. and L. D. Lee. "A theoretical basis for the analysis of multiversion software subject to coincident errors." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1511-1517.

54. Elliott, R. W. "Measuring computer software reliability." Computer and Industrial Engineering, 2, 1978, 141-151.
55. Elshoff, J. L. "An analysis of some commercial PL/1 programs." IEEE Trans. Software Engineering, SE-2, 1976, 113-120.
56. Endres, A. "An analysis of errors and their causes in system program." IEEE Trans. Software Engineering, SE-1, No. 2, 1975, 140-149.
57. Feller, W. An Introduction to Probability Theory and Its Application. Vol. I. Wiley, New York, 1968.
58. Fischer, K. F. and M. G. Walker. "Improved software reliability through requirements verification." IEEE Trans. Reliability, R-28, No. 3, 1979, 233-240.
59. Forman, E. H. and N. D. Singpurwalla. "Optimal time interval for testing hypothesis on computer software errors." IEEE Trans. Reliability, R-29, No. 4, 1979, 250-253.
60. Forman, E. H. and N. D. Singpurwalla. "An empirical stopping rule for debugging and testing computer software." Journal of the American Statistical Association, 72, No. 360, 1977.
61. Fragola, J. R. and J. F. Spahn. "The software error effects analysis: a quantitative design tool." Proc. Int'l Conf. on Reliable Software, 1973, 90-93.
62. Friedman, M. "Modeling the penalty cost of software failure." Proc. Annual Reliability and Maintainability Symposium, 1987, 359-363.
63. Funami, Y. and M. H. Halstead. "A software physics analysis of Akiyama's debugging data." Proc. Computer Software Engineering Symposium, 1976, 133-138.
64. Gilb, T. Software Metrics. Winthrop, Cambridge, Mass., 1977.
65. Girard, E. and J. C. Rault. "A programming technique for software reliability." Proc. Int'l Conf. on Reliable Software, 1973, 44-48.
66. Glass, R. L. "Persistent software errors." IEEE Trans. Software Engineering, SE-7, No. 2, 1981, 162-168.
67. Goel, A. L. "Software reliability models: assumptions, limitations, and applicability." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1411-1423.

68. Goel, A. L. "A summary of the discussion on an analysis of computing software reliability models." IEEE Trans. Software Engineering, SE-6, No. 5, 1980, 501-502.
69. Goel, A. L. and K. Okumoto. "When to stop testing and start using software?" Performance Evaluation Review, 1, No. 10, 1981, 131-138.
70. Goel, A. L. and K. Okumoto. "Time-dependent error detection rate model for software reliability and performance measures." IEEE Trans. Reliability, R-28, No. 3, 1979, 206-211.
71. Goel, A. L. and K. Okumoto. "A Markovian model for reliability and other performance measures of software systems." Proc. COMPCON, 1979, 769-774.
72. Goel, A. L. and K. Okumoto. "An analysis of recurrent software errors in a real-time control system." Proc. ACM Conf., 1978, 496-501.
73. Goel, A. L. and J. Soenjoto. "Models for hardware-software system operational performance evaluation." IEEE Trans. Reliability, R-30, No. 3, 1981, 232-239.
74. Govil, K. K. "Philosophy of a new structure of software reliability modeling." Microelectronics and Reliability, 24, No. 3, 1984, 407-409.
75. Greenspan, S. J. and C. L. McGowan. "Structuring software development for reliability." Microelectronics and Reliability, 17, No. 1, 1978, 77-84.
76. Gubitz, M. and K. O. Ott. "Quantifying software reliability by a probabilistic model." Reliability Engineering, 5, 1983, 157-171.
77. Halstead, M. H. Elements of Software Science. Elsevier, New York, 1977.
78. Hamilton, P. A. and J. D. Musa. "Measuring reliability of computation center software." Proc. 3rd Int'l Conf. on Software Engineering, 1978, 29-36.
79. Haney, F. M. "Module connection analysis - a tool for scheduling software debugging activities." Proc. AFIPS Conf., 41, part I, 1972, 173-179.
80. Hansen, G. A. "Measuring software reliability." Mini-Micro Systems, Aug. 1977, 54-57.

81. Hart, G. "The software integrity of a computer system installed in Royal Navy Frigate." Microelectronics and Reliability, 22, No. 6, 1982, 1061-1066.
82. 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, 109-122.
83. Haynes, R. D. and W. E. Thompson. "Combined hardware and software availability." Proc. Annual Reliability and Maintainability Symposium, 1981, 365-370.
84. Hecht, H. "Mini-tutorial on software reliability." Proc. COMPSAC, 1980, 383-385.
85. Hecht, H. "Can software benefit from hardware experience?" Proc. Annual Reliability and Maintainability Symposium, 1975, 480-485.
86. Hecht, H. "Fault-tolerant software." IEEE Trans. Reliability, R-28, No. 3, 1979, 227-232.
87. Hecht, H. and M. Hecht. "Software reliability in the system context." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 51-58.
88. Hellerman, L. "A measure of computational work." IEEE Trans. Computer, C-21, No. 5, 1972, 439-446.
89. Howden, W. E. "Empirical studies of software validation." Microelectronics and Reliability, 19, No. 1, 1979, 39-47.
90. Huang, X. Z. "The hypergeometric distribution model for predicting the reliability of software." Microelectronics and Reliability, 24, No. 1, 1984, 11-20.
91. Iannino, A. "Criteria for software reliability model comparison." IEEE Trans. Software Engineering, SE-10, No. 6, 1984, 687-691.
92. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard 729, 1983.
93. Islam, M. M. and F. Lombardi. "Estimation of total errors in software." Microelectronics and Reliability, 22, No. 2, 1982, 281-285.

94. Iyer, R. K. and D. J. Rossetti. "Effect of system workload on operating system reliability: a study on IBM 3081." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1438-1448.
95. Iyer, R. K. and P. Velardi. "Hardware-related software errors: measurement and analysis." IEEE Trans. Software Engineering, SE-11, No. 2, 1985, 223-231.
96. Jelinski, Z. and P. B. Moranda. "Software reliability research." in Statistical Computer Performance Evaluation. Academic Press, New York, 1972.
97. Jewell, W. S. "Bayesian extension to a basic model of software reliability." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1465-1471.
98. Joe, H. and N. Reid. "On the software reliability models of Jelinski-Moranda and Littlewood." IEEE Trans. Reliability, R-34, No. 3, 1985, 216-228.
99. Joe, H. and N. Reid. "Estimating the number of faults in a system." American Statistical Association, 80, No. 389, 1985, 222-226.
100. Jones, T. C. "Measuring programming quality and productivity." IBM Systems Journal, 17, No. 1, 1978, 39-63.
101. Kline, M. B. "Software and hardware R&M: what are the differences?" Proc. Annual Reliability and Maintainability Symposium, 1980, 179-185.
102. Koch, H. and P. Kubat. "Optimal release time of computer software." IEEE Trans. Software Engineering, SE-9, No. 3, 1983, 323-327.
103. Koch, G. and P. J. C. Spreij. "Software reliability as an application of martingale and filtering theory." IEEE Trans. Reliability, R-32, No. 4, 1983, 342-345.
104. Koch, H. S. and P. Kubat. "Quick and simple procedures to assess software reliability and facilitate project management." Journal of Systems and Software, 1981, 271-276.
105. Kopetz, H. Software Reliability. Macmillan Press Ltd., London, 1979.
106. Kremer, W. "Birth-death and bug counting." IEEE Trans. Reliability, R-32, No. 1, 1983, 37-47.

107. Krten, O. J. and D. A. Levy. "Software Modeling for optimal field entry." Proc. Annual Reliability and Maintainability Symposium, 1980, 410-414.
108. Kubat, P. and H. S. Koch. "Managing test procedure to achieve reliable software." IEEE Trans. Reliability, R-32, No. 3, 1983, 299-303.
109. Kubat, P. and H. S. Koch. "Pragmatic testing protocols to measure software reliability." IEEE Trans. Reliability, R-32, No. 4, 1983, 338-341.
110. Kuo, W. "On optimal burn-in modeling and its application to an electronic product." Proc. 3rd Int'l Conf. on Reliability and Maintainability, France, 1982, 18-22.
111. Kuo, W. "Software reliability estimation: a realization of competing risk." Microelectronics and Reliability, 23, No. 2, 1983, 249-260.
112. Landrault, C. J. C. Laprie. "Reliability and Availability modeling of systems featuring hardware and software faults." Proc. Int'l Conf. on Fault-Tolerant Computing, 1977, 10-15.
113. Langberg, N. and N. Singpurwalla. "A unification of some software reliability models." SIAM Journal on Scientific and Statistical Computing, 6, No. 3, 1985, 781-790.
114. Laprie, J. C. "Dependability evaluation of software systems in operation." IEEE Trans. Software Engineering, SE-10, No. 6, 1984, 701-714.
115. Lawless, J. F. Statistical Models and Methods for Lifetime Data, John Wiley & Sons, New York, 1982.
116. Leveson, N. G. and P. R. Harvey. "Analyzing software safety." IEEE Trans. Software Engineering, SE-9, No. 5, 1983, 569-579.
117. Levy, L. S. "A metaprogramming method and its economic justification." IEEE Trans. Software Engineering, SE-12, No. 2, 1986, 272-277.
118. Lipow, M. "Number of faults per line of code." IEEE Trans. Software Engineering, SE-8, No. 4, 1982, 437-439.
119. Lipow, M. "Prediction of software failures." Journal of Systems and Software, 1, No. 1, 1979, 71-75.

120. Lipow, M. and T. A. Thayer. "Prediction of software failures." Proc. Annual Reliability and Maintainability Symposium, 1977, 489-494.
121. Lipow, M. "Estimation of software package residual errors." TRW Software Series, Report TRW-SS-72-09, Redondo Beach, CA., 1972.
122. Lipow, M. "Some variation of a model for software time-to-failure." TRW Systems Group, Correspondence, ML-74-2260.1.9-21, Aug. 1974.
123. Littlewood, B. "A critique of the Jelinski-Moranda model for software reliability." Proc. Annual Reliability and Maintainability Symposium, 1981, 357-364.
124. Littlewood, B. "Stochastic reliability growth: a model for fault removal in computer program." IEEE Trans. Reliability, R-30, No. 4, 1981, 313-320.
125. Littlewood, B. "A Bayesian differential debugging model for software reliability." Proc. COMPSAC, 1980, 511-517.
126. Littlewood, B. "Theories of software reliability: how good are they and how can they be improved." IEEE Trans. Software Engineering, SE-6, No. 5, 1980, 489-500.
127. Littlewood, B. "What makes a reliable program - few bugs or a small failure rate?" Proc. COMPCON, 1980, 707-713.
128. Littlewood, B. "The Littlewood-Verrall model for software reliability compared with some rivals." Journal of Systems and Software, 1980, 251-258.
129. Littlewood, B. "How to measure software reliability and how not to." IEEE Trans. Reliability, R-28, No. 2, 1979, 103-110.
130. Littlewood, B. "Software reliability models for modular program structure." IEEE Trans. Reliability, R-28, No. 3, 1979, 241-245.
131. Littlewood, B. "Validation of a software reliability model." Proc. Software Life Cycle Management Workshop, 1978, 146-152.
132. Littlewood, B. "A semi-Markov model for software reliability with failure cost." Proc. Symposium on Computer Software Engineering, 1976, 281-300.
133. Littlewood, B. "A reliability model for systems with Markov structure." Applied Statistics, 24, No. 2, 1975, 172-177.

134. Littlewood, B. and P. A. Keiller. "Adaptive software reliability modeling." IEEE 0731-3071/84.
135. Littlewood, B. and J. L. Verrall. "Likelihood function of a debugging model for computer software reliability." IEEE Trans. Reliability, R-30, No. 2, 1981, 145-148.
136. Littlewood, B. and J. L. Verrall. "A Bayesian reliability model with stochastically monotone failure rate." IEEE Trans. Reliability, R-23, No. 2, 1974, 108-114.
137. Lloyd, D. K. and M. Lipow. Reliability: Management, Methods, and Mathematics. Lloyd, Redondo Beach, CA., 1977.
138. Makam, S. V. and A. Avizienis. "ARIES 81: a reliability and life cycle evaluation tool for faulttolerant systems." Proc. Int'l Conf. on Fault-Tolerant Computing, 1982, 267-274.
139. McCabe, T. J. "A complexity measure." IEEE Trans. Software Engineering, SE-2, 1976, 308-320.
140. McCall, J. A., P. Pierce, R. Hartley, and R. Thonerfelt. "Measuring technology for software life cycle support," COMPCON, 1985, 310-320.
141. Migneault, G. E. "Software reliability and advanced avionics." Proc. COMPCON, 1980, 715-720.
142. Miller, D. R. "Exponential order statistics models of software reliability growth." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 13-24.
143. Miller, D. R. and A. Sofer. "Completely monotone regression estimation of software failure rate." Proc. Int'l Conf. on Software Engineering, 1985, 343-348.
144. Mills, H. D. "On the statistical validation of computer programs." IBM FSD, unpublished paper, July 1970.
145. Misra, P. N. "Software reliability analysis." IBM Systems Journal, 22, No. 3, 1983, 262-270.
146. Miyamoto, I. "Reliability evaluation and management for an entire software life cycle." Software Life Cycle Management Workshop, 1978.
147. Moawad, R. "Comparison of current software reliability models." Proc. Int'l Conf. on Software Engineering, 1984, 222-229.

148. Mohanty, S. N. "Models and measurements for quality assessment of software." Computing Surveys, 11, No. 3, 1979, 251-275.
149. Moranda, P. B. "An error detection model for application during software development." IEEE Trans. Reliability, R-30, No. 4, 1981, 309-312.
150. Moranda, P. B. "Event-altered rate models for general reliability analysis." IEEE Trans. Reliability, R-28, No. 5, 1979, 376-381.
151. Moranda, P. B. "A comparison of software error-rate models." Proc. Texas Conf. on Computing Systems, 1975, 6.1-6.9.
152. Moranda, P. B. "Prediction of software reliability during debugging." Proc. Annual Reliability and Maintainability Symposium, 1975, 327-332.
153. Morey, R. C. "Estimating and improving the quality of information in a MIS." Communications of the ACM, 25, No. 5, 1982, 337-342.
154. Mourad, S. and D. Andrews. "The reliability of the IBM MVS/XA operating system." Proc. Int'l Conf. on Fault-Tolerant Computing, 1985, 93-98.
155. Musa, J. D., A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application, to be published by McGraw-Hill, New York, 1987.
156. 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.
157. Musa, J. D. and K. Okumoto. "A logarithmic Poisson execution time model for software reliability measurement." Proc. 7th Int'l Conf. on Software Engineering, 1984, 230-238.
158. Musa, J. D. and K. Okumoto. "A comparison of time domains for software reliability models." Journal of Systems and Software, 4, No. 4, 1984, 277-287.
159. Musa, J. D. and K. Okumoto. "Software reliability models: concepts, classification, comparison, and practice." Proc. Electronic Systems Effectiveness and Life Cycle Cost Conf., NATO Advanced Study Series. Springer Verlag, Heidelberg, 1983, 395-424.

160. Musa, J. D. "Software reliability measurement." Journal of Systems and Software, 1, No. 3, 1980, 223-241.
161. Musa, J. D. "Software reliability measures applied to system engineering." Proc. COMPCON, 1979, 941-946.
162. Musa, J. D. "Validation of execution time theory of software reliability." IEEE Trans. Reliability, R-28, No. 3, 1979, 181-191.
163. Musa, J. D. "The use of software reliability measures in project management." Proc. COMPSAC, 1978, 493-498.
164. Musa, J. D. "A theory of software reliability and its application." IEEE Trans. Software Engineering, SE-1, No. 3, 1975, 312-327.
165. Nathan, I. "A deterministic model to predict error-free status of complex software development." Workshop on Quantitative Software Models, 1979, 159-169.
166. Nayak, T. K. "Software reliability: statistical modeling and estimation." IEEE Trans. Reliability, R-35, No. 5, 1986, 566-570.
167. Nayak, T. K. "Estimating population size by recapture sampling." Report. Department of Statistics, George Washington University, 1986.
168. Nelson, E. "Estimating software reliability from test data." Microelectronics and Reliability, 17, No. 1, 1978, 67-74.
169. Nelson, E. C. "A statistical basis for software reliability assessment." TRW-SS-73-03, TRW, 1973.
170. Ohba, M. "Software reliability analysis models." IBM Journal of Research and Development, 28, No. 4, 1984, 428-443.
171. Ohba, M., S. Yamada, K., K. Takeda, and S. Osaki. "S-shaped software reliability growth curve: how good is it?" IEEE CH1810-1/82, 1982, 38-44.
172. Okumoto, K. "A statistical method for software quality control." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1424-1430.
173. Okumoto, K. and A. Goel. "Optimal release time for software systems based on reliability and cost." Journal of Systems and Software, 1, No. 4, 1980, 315-318.

174. Parzen, E. Stochastic Processes. San Francisco, Holden-Day, 1962.
175. Peercy, D. E. "A software maintainability evaluation methodology." IEEE Trans. Software Engineering, SE-7, No. 4, 1981, 343-351.
176. Potier, D. "Experiments with computer software complexity and reliability." Proc. Int'l Conf. on Software Engineering, 1982, 94-101.
177. Ramamoorthy, C. V. and F. B. Bastani. "Modeling of the software reliability growth process." Proc. COMPSAC, 1980, 161-169.
178. Ramamoorthy, C. V. and F. B. Bastani. "Software reliability - status and perspectives." IEEE Trans. Software Engineering, SE-8, No. 4, 1982, 354-371.
179. Ramamoorthy, C. V. and S. F. Ho. "Testing large software with automated software evaluation systems." IEEE Trans. Software Engineering, SE-1, No. 1, 1975, 46-58.
180. Ramzan, M. T. "Seeded bug volume for software validation." Microelectronics and Reliability, 23, No. 5, 1983, 981-988.
181. Reifer, R. J. "Software failure modes and effects analysis." IEEE Trans. Reliability, R-28, No. 3, 1979, 247-249.
182. Reiss, R. M. "A prediction experience with three software reliability models." Workshop on Quantitative Software Models, 1979, 190-195.
183. Romeu, J. L. and K. A. Dey. "Classifying combined hardware/software R models." Proc. Annual Reliability and Maintainability Symposium, 1984, 282-287.
184. Ronback, J. A. "Software reliability - how it affects system reliability." Microelectronics and Reliability, 14, No. 2, 121-140.
185. Rosene, A. F., J. E. Connolly, and K. M. Bracy. "Software maintainability - what it means and how to achieve it." IEEE Trans. Reliability, 30, No. 3, 1981, 240-245.
186. Ross, S. M. "Software reliability: the stopping rule problem." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1472-1476.
187. Ross, S. M. "Statistical estimation of software reliability." IEEE Trans. Software Engineering, SE-11, No. 5, 1985, 479-483.

188. Rossetti, D. J. and R. K. Iyer. "Software related failures on the IBM 3081: a relationship with system utilization." CHI810-IEEE, 1982, 45-54.
189. Rubey, R. J. "Planning for software reliability." Proc. Annual Reliability and Maintainability Symposium, 1977, 495-498.
190. Rubey, R. J., J. A. Dana, and P. W. Biche. "Quantitative aspect of software validation." IEEE Trans. Software Engineering, SE-1, No. 2, 1975, 150-155.
191. Rutledge, R. A. "The reliability of memory subject to hard and soft failures." Proc. Int'l Conf. on Fault-Tolerant Computing, 1980, 193-198.
192. 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, 259-266.
193. Schick, G. J. and R. W. Wolverton. "An analysis of competing software reliability models." IEEE Trans. Software Engineering, SE-4, No. 2, 1978, 104-120.
194. Schick, G. J. and R. W. Wolverton. "Achieving reliability in large software system." Proc. Annual Reliability and Maintainability Symposium, 1974, 302-319.
195. Schneider, V. "Some experimental estimators for developmental and delivered errors in software development projects." Proc. COMPSAC, 1980.
196. Schneidewind, N. F. "The use of simulation in the evaluation of software." Computer, 10, No. 4, 1977, 47-53.
197. Schutt, D. "On a hypergraph oriented measure for applied computer science." Proc. COMPCON, 1977, 295-296.
198. Scholz, F. W. "Software reliability modeling and analysis." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 25-31.
199. Scott, R. K. "Experimental validation of six fault-tolerant software reliability models." Proc. Int'l Conf. on Fault-Tolerant Computing, 1984, 102-107.
200. Serra, A. and R. E. Barlow, ed. Theory of Reliability. North-Holland, Amsterdam, 1986.
201. Shannon, C. and W. Weaver. The Mathematical Theory of Communication. University of Illinois Press, Urbana, 1975.

202. Shanthikumar, J. G. "Software reliability models: a review." Microelectronics and Reliability, 23, No. 5, 1983, 903-943.
203. Shanthikumar, J. G. "A general software reliability model for performance prediction." Microelectronics and Reliability, 21, No. 5, 1981, 671-682.
204. Shanthikumar, J. G. "A state and time-dependent error occurrence-rate software reliability model with imperfect debugging." Proc. COMPCON, 1981, 311-315.
205. Shanthikumar, J. G. and S. Tufekci. "Application of a software reliability model to decide software release time." Microelectronics and Reliability, 23, No. 1, 1983, 41-59.
206. Shen, V. Y. "Identifying error-prone software - an empirical study." IEEE Trans. Software Engineering, SE-11, No. 4, 1985, 317-324.
207. Shen, V. Y., S. D. Conte, and H. E. Hunsmore. "Software science revisited: a critical analysis of the theory and its empirical support." IEEE Trans. Software Engineering, SE-9, No. 2, 1983, 155-165.
208. Shooman, M. L. "Structure models for software reliability prediction." Proc. Int'l Conf. on Software Engineering, 1984, 268-273.
209. Shooman, M. L. Software Engineering - Design, Reliability, and Management. McGraw Hill, New York, 1983.
210. Shooman, M. L. "Software reliability - analysis and prediction." Integrity in Electronic Flight Control Systems, France, AGARD AG224, 1977.
211. Shooman, M. L. "Software reliability: measurement and models." Proc. Annual Reliability and Maintainability Symposium, 1975, 485-491.
212. Shooman, M. L. "Probabilistic Models for Software Reliability and Prediction." Statistical Computer Performance Evaluation. Academic Press, New York, 1972.
213. Shooman, M. L. and A. K. Trivedi. "A many-state Markov model for computer software performance parameters." IEEE Trans. Reliability, R-25, No. 2, 1976, 66-68.
214. Simkins, D. J. "Software performance modeling and management." IEEE Trans. Reliability, R-32, No. 3, 1983, 293-297.

215. Singpurwalla, N. D. and R. Soyer. "Assessing software reliability growth using a random coefficient autoregressive process and its ramifications." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1456-1464.
216. Soi, I. M. "Software complexity: an aid to software maintainability." Microelectronics and Reliability, 25, No. 2, 1985, 223-228.
217. Soi, I. M. and K. Gopal. "Hardware vs software reliability - a comparative study." Microelectronics and Reliability, 20, 1980, 881-885.
218. Soi, I. M. and K. Gopal. "Some aspects of reliable software packages." Microelectronics and Reliability, 19, 1979, 379-386.
219. Soi, I. M. and K. Gopal. "Error prediction in software." Microelectronics and Reliability, 18, 1978, 433-436.
220. Soi, I. M. and K. Gopal. "Detection and diagnosis of software malfunctions." Microelectronics and Reliability, 18, 1978, 353-356.
221. Spreij, P. "Parameter estimation for a specific software reliability model." IEEE Trans. Reliability, R-34, No. 4, 1985, 323-328.
222. Strandberg, K. and H. Anderson. "On a model for software for software reliability performance." Microelectronics and Reliability, 22, No. 2, 1982, 227-240.
223. Strong, E. J. "Software reliability and maintainability in large scale systems." Proc. COMPSAC, 1978, 755-760.
224. Sugiura, N. "On the software reliability." Microelectronics and Reliability, 13, 1974, 529-534.
225. Sukert, A. "A guidebook for software reliability assessment." Proc. Annual Reliability and Maintainability Symposium, 1980, 186-190.
226. Sukert, A. N. "Empirical validation of three software error prediction models." IEEE Trans. Reliability, R-28, No. 3, 1979, 199-204.
227. Sukert, A. N. "A four-project empirical study of software error prediction models." Proc. COMPSAC, 1978, 577-582.

228. Sukert, A. N. "Analysis of software error model predictions and questions of data availability." Software Life Cycle Management Workshop, 1978, 209-215.
229. Sukert, A. N. "An investigation of software reliability models." Proc. Annual Reliability and Maintainability Symposium, 1977, 478-484.
230. Sumita, U. and Y. Masuda. "Analysis of software availability/reliability under the influence of hardware failures." IEEE Trans. Software Engineering, SE-12, No. 1, 1986, 32-41.
231. Sumita, U. and J. G. Shanthikumar. "A software reliability model with multiple error introduction and removal." IEEE Trans. Reliability, R-35, No. 4, 1986, 459-462.
232. Suri, P. K. and K. K. Aggarwal. "Software reliability of programs with network structure." Microelectronics and Reliability, 21, No. 2, 1981, 203-207.
233. Suri, P. K. and K. K. Aggarwal. "Reliability evaluation of computer programs." Microelectronics and Reliability, 20, 1980, 465-470.
234. Swearingen, D. and J. Donahoo. "Quantitative software reliability models - data parameters: a tutorial." Workshop on Quantitative Software Models, 1979, 143-153.
235. Szabo, S. G. "A schema for producing reliable software." Proc. Int'l Conf. on Fault-Tolerant Computing, 1980, 151-155.
236. Takahashi, M. and Y. Kamayachi. "An empirical study of a model for program error prediction." Proc. Int'l Conf. on Software Engineering, 1985, 330-336.
237. Taylor, D. J. "Redundancy in data structure: improving software fault tolerance." IEEE Trans. Software Engineering, 6, No. 6, 1980, 585-594.
238. Taylor, H. M. and S. Karlin. An Introduction to Stochastic Modeling. Academic Press, New York, 1984.
239. Thayer, T. A., M. Lipow, and E. C. Nelson. Software Reliability. North-Holland Publishing Co., Amsterdam, 1978.
240. Thompson, W. E. and P. O. Chelson. "Software reliability testing for embedded computer systems." Workshop on Quantitative Software Models, 1979, 201-208.

241. Thompson, W. E. and P. O. Chelson. "On the specification and testing of software reliability." Proc. Annual Reliability and Maintainability Symposium, 1980, 379-383.
242. Trachtenberg, M. "The linear software reliability model and uniform testing." IEEE Trans. Reliability, R-34, No. 1, 1985, 8-16.
243. Trachtenberg, M. "Order and difficulty of debugging." IEEE Trans. Software Engineering, SE-9, No. 6, 1983, 746-747.
244. Troy, R. and Y. Romain. "A statistical methodology for the study of the software failure process and its application to the ARGOS center." IEEE Trans. Software Engineering, SE-12, No. 9, 1986, 968-978.
245. Troy, R. and R. Moawad. "Assessment of software reliability models." IEEE CH1810-1, 1982, 28-37.
246. Wagoner, W. L. "The final report on a software reliability measurement study." Report TOR-0074(41221)-1. The Aerospace Corp., El Segundo, CA, 1973.
247. Wall, J. K. and P. A. Ferguson. "Pragmatic software reliability prediction." Proc. Annual Reliability and Maintainability Symposium, 1977, 485-488.
248. Walsh, T. "A software reliability study using a complexity measure." Proc. AFIPS Conf., 48, 1979.
249. Walters, G. F. and J. A. McCall. "Software quality metrics for life-cycle cost reduction." IEEE Trans. Reliability, R-28, No. 3, 1979, 212-220.
250. Williams, M. W. H. "Reliability of large real-time control software systems." Proc. Int'l Conf. on Reliable Software, 1973, 1-6.
251. Woodward, M., M. Hennell, and D. Hedley. "A measure of control flow complexity in program text." IEEE Software Engineering, SE-5, No. 1, 1979, 45-50.
252. Yamada, S. and S. Osaki. "Software reliability growth modeling: models and applications." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1431-1437.
253. Yamada, S. and S. Osaki. "Cost-reliability optimal release policies for software systems." IEEE Trans. Reliability, R-34, No. 5, 1985, 422-424.

254. 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, 91-112.
255. Yamada, S. "Software reliability analysis based on a nonhomogeneous error detection rate model." Microelectronics and Reliability, 24, No. 5, 1984, 915-920.
256. Yamada, S. "S-shaped reliability growth modeling for software error detection." IEEE Trans. Reliability, R-32, No. 5, 1983, 475-478.
257. Yee, J. G. and S. Y. H. Su. "A scheme for tolerating fault data in real-time systems." Proc. COMPSAC, 1978, 663-667.

SECTION V. RELIABILITY COSTS IN SOFTWARE LIFE-CYCLE MODELS

INTRODUCTION

The investigation of trade-off among reliability, schedule, resources, and costs in hardware development is also of interest in software development. Although a functional relationship clearly exists between software life-cycle cost and software reliability, the parameters associated with such a relationship are not readily available. One difficulty in developing reliability-related cost models for software is that, unlike hardware, each software system is a new product, so that previous experiences may at most serve as a reference point.

As software costs have increased over the past two decades, the cost structure of the system has changed dramatically. In 1960, about 20 percents of the system's cost was spent on software. In 1985, that percentage had risen to 80 percents [23]. This change has drawn much attention as to how the software portion of the cost is determined and how it can be minimized. So far, studies of software cost have concentrated on development cost; however, life-cycle cost is more appropriate to study.

In hardware, life-cycle cost is usually studied from a buyer's standpoint. It can be divided into procurement cost, maintenance cost, and disposal cost. Since software development and maintenance are normally performed by the same organization, software life-cycle cost is usually studied from the developer's point of view and is divided into development cost and maintenance cost.

The software development process can be broken down into requirement and specification phase, design phase, coding phase, and testing phase. Among these, testing including unit test, integration test, and field test, accounts for 40 percents or more of the development cost. The maintenance activities include preventive maintenance, corrective maintenance, adaptive maintenance, enhancement, and growth. It is recognized that 60 percents of the software life-cycle cost are maintenance costs [21]. Again, testing is also the major cost factor in the maintenance phase.

For common software projects, reliability cost is mainly incurred by testing. For highly reliable software, such as that used in flight control systems, nuclear power plant control systems, and military systems, additional reliability cost is incurred at every phase of the software life cycle [25]. As indicated by Boehm [4], there is a very high productivity range of 1.87 between very low and very high reliability projects. Indeed, a large portion of the software life-cycle cost is devoted to achieving high reliability. Table 5.1 compares reliability costs incurred at each phase of the software life cycle for common and highly reliable software. Unfortunately, none of the existing quantitative software models can deal with this issue properly. Models that address the relationship between software reliability and cost are surveyed and summarized below. A generalized bug-counting software reliability cost minimization model is proposed.

TABLE 5.1. Reliability cost and software life-cycle phases

Phase	Reliability Cost of Common Software	Additional Reliability Cost for Highly Reliable Software
DEVELOPMENT		
Requirements and specifications	Basic requirement and specification walkthrough	Parallel development of requirement and specification, and detailed validation
Design	Basic design walkthrough	Parallel design, fault-tolerant design, and detailed verification.
Coding	Basic coding walkthrough	Parallel coding of critical modules, fault-tolerant codes, and detailed code walkthrough
MAINTENANCE		
Preventive maintenance	Totally devoted to reliability	Higher frequency of preventive maintenance
Corrective maintenance	Totally devoted to reliability	Immediate correction and extra testing
Adaptive maintenance	Testing	Extra testing
Enhancement	Equivalent to a development subcycle	
Growth	Equivalent to a development subcycle	

REVIEW OF THE RELIABILITY-RELATED SOFTWARE COST MODELS

The software life-cycle cost model and the software reliability model are two quantitative ways of dealing with reliability-related software costs. The software life-cycle model can be subdivided into the cost-estimation model, the resource-allocation model, and the program-evolution model, each describing a different aspect of software life-cycle cost. The cost-estimation model estimates the amount of resources required, the resource-allocation model shows how resources are distributed over the life cycle, and the program evolution model describes the dynamic nature of software and the trade-off between development cost and maintenance cost.

Cost Estimation

The cost-estimation model estimates efforts, including manpower, computer time, documentation, and project duration required at the development phase as well as over the entire life cycle. These estimates are based on cost factors identified from historical data by the regression analysis. Typical cost factors are the number of instructions, percentage of new instructions, number of files, number of reports, number of miles traveled, number of display consoles, pages of documentation, average experience of programmers, etc. [2,4,16,24]. A simple baseline model has only one cost factor, while a complicated model may involve many cost factors. A general formula can be expressed as follows.

$$E = \sum_{i=1}^n a_i X_i$$

where

E efforts
 X_i the ith cost factor
 a_i, b_i coefficients of the ith cost factor
 n number of cost factors.

Coefficients of the above model can be adjusted to reflect the particular application and environment by using a weighting method, a table driven method, or a formula. Adjustment may involve a single attribute or multiple attributes. Typical adjustment attributes are type of application, degree of difficulty, reliability, complexity, development methodology, etc. [5,16,24,25]. For those models that include reliability as one of the cost attributes, the reliability cost can be estimated directly from the model. Otherwise, reliability cost can be estimated from the degree of difficulty, system complexity, and type of application.

Resource Allocation

The resource-allocation model distributes resources to the phases of the software life-cycle according to a manpower utilization curve. Originally, Norden applied the Rayleigh curve to represent the resource allocation in research and development projects [18]. The Rayleigh curve model was later adapted by Putnam to represent the manpower

buildup of the software life cycle [21]. Putnam's Rayleigh curve model can be summarized as follows.

$$y(t) = 2Kae^{-at^2}$$

$$Y(t) = \int_0^t y(s) ds = K \left(1 - e^{-at^2} \right)$$

$$\left. \frac{dy(t)}{dt} \right|_{t_d} = 2Kae^{-at_d^2} \left(-2at_d \right) = 0$$

$$t_d = (2a)^{1/2}$$

$$C_d = Y(t_d) = \int_0^{t_d} y(s) ds = 0.3945K$$

where

$y(t)$ density function of manpower utilization

$Y(t)$ cumulative manpower utilization

K total manpower

t_d development time (release time)

C_d development cost

a constant of proportionality.

Through empirical observation, development cost is defined as the time when the manpower curve reaches its peak, which is close to 40 percents of the total cost. Other quantities such as degree of difficulty, productivity, and technology level are also derived. This

same idea of fitting the staffing curve to a parametric distribution is also used in the Sech-square model by Parr [20], the parabolic model, and the trapezoid model by Basili and Beane [1]. As in the cost-estimation model, reliability is not treated explicitly. However, reliability cost can be traced from difficulty level, testing phase, and total manpower. The 40 to 60 breakdown of development and maintenance costs serves as a guideline for reliability cost allocation.

Program Evolution

The program-evolution model describes the dynamic nature of software. The software is subject to constant change after delivery. Correcting errors, adding new functions, deleting unnecessary functions, adapting to the new environment, and improving performance are among the major activities of the evolution process. As new functions and new codes are added, the reliability of the software decreases. Unless effort is devoted to keeping the reliability under control, further changes will make it even more costly to maintain the desired reliability [3,13]. Resources can be devoted to growth which tends to increase the failure rate, to error removal which will decrease the failure rate, or to routine service which does not affect the failure rate. The ultimate purpose of the evolution model is to consider these conflicting factors under limited resources and to provide a guideline to management for setting up the optimum reliability level and the optimum release time.

The program evolution can be approached by analytical or simulation models [3,14,26]. Reliability can be related to the size of the program (total number of modules, number of modules changed, number of modules added), release number, system load, operational profile, and complexity measures [15]. Unlike the cost-estimation model and the resource-allocation model, which are concerned with the amount of reliability cost, the program-evolution model describes the interactions between reliability and other factors.

SOFTWARE RELIABILITY AND COST

The software-reliability model measures and predicts the reliability of the software during testing and maintenance phases. Software reliability is defined as the probability of failure-free operation of a software program under the specified conditions for a specified period of time. Most software reliability models fall into the category of the "bug-counting" model, which represent the number of remaining faults (or the number of failures experienced) at time t as a stochastic counting process. The following functions are derived to characterized the software failure process.

- the number of faults remaining at time t , $N(t)$
- the mean value function of $N(t)$, $M(t)=E[N(t)]$
- the failure rate of the software, $\lambda(t)$
- the reliability function, $R(t) = \exp[- \int_0^t \lambda(s) ds]$
- The probability of k remaining faults at time t , $P_k(t)$

This counting process can be modeled as a continuous-time, discrete-state Markov chain. Under the following assumptions, the model is reduced to the birth-and-death process with linear birth rate and linear death rate [11,12].

1. The failure rate is proportional to the number of faults remaining,
2. each fault has the same failure rate $\phi(t)$, and
3. whenever a failure occurs, the number of faults is reduced by 1 with probability p , increased by 1 with probability q , and not changed with probability $1-p-q$.

The transition diagram of the $N(t)$ process is shown in Fig. 4.5. The differential equations of $P_k(t)$ is

$$P_k(t+\Delta t) = (k-1)q\phi(t)P_{k-1}(t)\Delta t + [1-k(p+q)\phi(t)\Delta t]P_k(t) + (k+1)p\phi(t)P_{k+1}(t)\Delta t + o(\Delta t) \quad (5.1)$$

with the initial condition

$$P_k(0) = \begin{cases} 1 & \text{for } k = N \\ 0 & \text{for } k \neq N. \end{cases}$$

Rearranging Eq. (5.1), dividing by Δt , and taking the limit as $\Delta t \rightarrow 0$ gives

$$P_k'(t) = (k-1)q\phi(t)P_{k-1}(t) - k(p+q)\phi(t)P_k(t) + (k+1)p\phi(t)P_{k+1}(t) \quad (5.2)$$

The mean value function of $N(t)$ is defined as

$$M(t) = E[N(T)] = \sum_{k=1}^{\infty} kP_k(t) \quad (5.3)$$

Taking the derivative of Eq. (5.3) and substituting Eq. (5.2) into it,

$$\begin{aligned} M'(t) &= q\phi(t)\sum_k (k-1)P_{k-1}(t) - p\phi(t)\sum_k (k+1)P_{k+1}(t) \\ &= -(p-q)\phi(t)M(t) \end{aligned}$$

This differential equation of $M(t)$ with initial condition, $M(0)=N$, gives [7]

$$M(t) = N \cdot \exp \left[-(p-q) \int_0^t \phi(s) ds \right].$$

The above mean value function can be incorporated into the software life-cycle cost model to determine the optimal release time and the optimal reliability level. Total reliability cost, consisting of reliability cost during testing and reliability cost during maintenance, can be formulated on a "per-fault" basis [7,10,19,22]. The reliability cost during testing is a function of the number of faults removed during testing and the length of testing time. The reliability cost during maintenance is also a function of the number of faults removed during operation and the length of the operational time. Then, total reliability cost can be expressed as follows.

$$\begin{aligned} TC(t) &= C_1 [M(0) - M(t)] + (C_2 + C_3) [M(t) - M(T)] + C_4 t + C_5 (T - t) \\ &= (C_2 + C_3 - C_1) M(t) + (C_4 - C_5) t + C_1 M(0) - (C_2 + C_3) M(T) \\ &\quad + C_5 T. \end{aligned} \tag{5.4}$$

The variable cost with respect to t is

$$\begin{aligned} VC(t) &= C_6 M(t) + C_7 t \\ &= C_6 N \cdot \exp \left[-(p-q) \int_0^t \phi(s) ds \right] + C_7 t \end{aligned}$$

where

- C_1 cost of correcting a fault during testing
- C_2 cost of correcting a fault during operation
- C_3 penalty costs per fault during operation
- C_4 cost of testing per unit time

C_5	cost of maintenance per unit time
C_6	$C_2 + C_3 - C_1$
C_7	$C_4 - C_5$
TC	total reliability cost
VC	variable reliability cost with respect to t
t^*	optimal release time
T	useful life of the software.

The minimum of the variable cost can be found by setting the derivative to zero.

$$VC'(t) = -C_6 N \phi(t) \exp \left[-(p-q) \int_0^t \phi(s) ds \right] + C_7$$

$$\phi(t) \exp \left[-(p-q) \int_0^t \phi(s) ds \right] = \frac{C_7}{C_6 N}$$

and

$$\ln \phi(t) - (p-q) \int_0^t \phi(s) ds = \ln \frac{C_7}{C_6 N} \quad 0 \leq t \leq T. \quad (5.5)$$

Given a specific failure rate function, the optimal release time can be determined from Eq. (5.5).

For a constant failure rate model [8,9], $\phi(t) = \phi$, the optimal release time can be shown to be

$$t^* = \ln \left(\frac{C_6 \phi N}{C_7} \right)^{\frac{1}{(p-q)\phi}}$$

For an exponentially decreasing failure rate, $\phi(t) = \phi e^{-\beta t}$,

$$\ln(\phi e^{-\beta t}) - (p-q) \int_0^t \phi e^{-\beta s} ds = \ln \frac{C_7}{C_6 N}$$

and

$$\frac{(p-q)\phi}{\beta} e^{-\beta t} - \beta t - \ln \frac{C_7}{C_6 \phi N} - \frac{(p-q)\phi}{\beta} = 0 \quad 0 \leq t \leq T. \quad (5.6)$$

This is a single-variable root-finding problem and can be solved by Newton's method. It can be shown that the second derivative is positive for both the constant failure rate in Eq. (5.5) and the exponentially decreasing failure rate in Eq. (5.6). The solutions obtained by setting the first derivative to zero are indeed a minimum.

As indicated by Musa et al. [17], failure identification personnel, failure correction personnel, and computer time are required in testing. These limiting resources should be considered in determining the cost coefficients of C_1 , C_2 , C_4 , and C_5 . In determining C_3 , the failure can be classified into levels of severity. The number of faults and cost per fault are estimated for each severity level. Then, C_3 can be estimated based on expectation.

To illustrate the exponentially decreasing failure rate model, let $N=200$, $C_1=5$, $C_2=20$, $C_3=50$, $C_4=200$, $C_5=20$, $p-q=0.95$, $T=200$, $\phi = 0.2/\text{week}$, and $\beta = 0.01$. From Eq. (5.6),

$$19 \exp(-t/100) - 0.01t - 16.5 = 0 \quad 0 \leq t \leq 200$$

The solution is $t=13.3$. Therefore, the life-cycle cost is minimized when testing time is 13.3 week.

CONCLUSION

The cost-estimation model, the resource-allocation model, and the program evolution model all deal with reliability - empirically, indirectly, and subjectively. However, these macro models point out different aspects of software reliability cost issues and pave the way for future development of reliability-related life-cycle cost models. Software-reliability models, based on rigorous reliability theory, can be used to estimate reliability cost more precisely. This study examined life-cycle cost modeling with emphasis on reviewing reliability cost in the software life cycle. Once the software-reliability costs are taken care of, the software life-cycle cost can readily be obtained.

REFERENCES

1. Basili, V. R. and J. Beane. "Can the Parr curve help with manpower distribution and resource estimation problem?" Journal of Systems and Software, 2, No. 1, 1981, 59-69.
2. Basili, V. R. and K. Freburger. "Programming measurement and estimation in the software engineering laboratory." Journal of Systems and Software, 2, No. 1, 1981, 47-57.
3. Belady, L. A. and M. M. Lehman. "A model of large program development." IBM Systems Journal, No. 3, 1976, 225-252.
4. Boehm, B. W. Software Engineering Economics. Prentice-Hall, Englewood Cliffs, NJ, 1981.
5. Boehm, B. W. and R. W. Wolverton. "Software cost modeling: some lessons learned." Journal of Systems and Software, 1, No. 2, 1980, 195-201.
6. Feller, W. An Inew York, 1973. Probability and Its Applications, Vol. I. John Wiley and Sons, N.Y., 1973.
7. Forman, E. H. and N. D. Singpurwalla. "Optimal time intervals for testing hypothesis in computer software errors." IEEE Trans. on Reliability, R-28, No. 3, 1979, 250-253.
8. Goel, A. L. and K. Okumoto. "Time-dependent error-detection rate model for software reliability and performance measures." IEEE Trans. Reliability, R-28, No. 3, 1979, 206-211.
9. Jelinski, Z. and P. B. Moranda. "Software reliability research." in Statistical Computer Performance Evaluation. Academic Press, New York, 1972, 465-484.
10. Koch, H. S. and P. Kubat. "Optimal release time of computer software." IEEE Trans. Software Engineering, SE-9, No. 3, 1983, 323-327.
11. Kremer, W. "Birth-death and bug counting." IEEE Trans. Reliability, R-32, No. 1, 1983, 37-47.
12. Kuo, W. "Software reliability estimation: a realization of competing risk." Microelectronics and Reliability, 23, No. 2, 1983, 249-260.

13. Lehman, M. M. "Programs, life cycles, and laws of software evolution." Proceedings of the IEEE, 68, No. 9, 1980, 1060-1076.
14. Leman, M. M. and L. A. Belady. "Programming system dynamics or the meta-dynamics of systems in maintenance and growth." IBM Research Report, RC3546, 1971.
15. McCall, J. "Measurement technology for software life cycle support." Proceedings of the COMPCON, 1985, 313-320.
16. Mohanty, S. N. "Software cost estimation: present and future." Software - Practice and Experience, 11, 1981, 103-121.
17. Musa, J. D., A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application. to be published by McGraw-Hill, New York, 1987.
18. Norden, P. V. "Useful tools for project management." in Operations Research and Development. John Wiley & Sons, New York, 1963.
19. Okumoto, K. and A. L. Goel. "Optimal release time for software systems." Proceedings of the COMPSAC, 1979, 500-503.
20. Parr, F. N. "An alternative to the Rayleigh curve model of software development effort." IEEE Trans. Software Engineering, SE-6, No. 5, 1980, 291-296.
21. Putnam, L. H. "A general empirical solution to macro software sizing and estimation problem." IEEE Trans. Software Engineering, SE-4, No. 7, 1978, 345-361.
22. Shanthikumar, J. G. and S. Tufekci. "Application of a software reliability model to decide software release time." Microelectronics and Reliability, 23, No. 1, 1983, 41-59.
23. Shooman, M. L. Software Engineering - Design, Reliability, and Management. McGraw Hill, New York, 1983.
24. Walston, C. E. and C. P. Felix. "A method of programming measurement and estimation." IBM Systems Journal, 16, No. 1, 1977, 54-73.
25. Wolverton, R. W. "The cost of developing large scale software." IEEE Computers, C-23, No. 6, 1974, 615-636.
26. Woodside, C. M. "A mathematical model for the evolution of software." Journal of Systems and Software, 1, No. 3, 1980, 337-345.

SECTION VI. RELIABILITY OPTIMIZATION WITH SOFTWARE COMPONENT

SOFTWARE RELIABILITY-COST FUNCTION

Previous Sections have discussed the issues of mixed-integer reliability techniques, software reliability models, and software reliability costs. This Section applies these materials to integrate software components into the system reliability optimization problem. Assume that the reliability of a system with hardware components and software components is to be optimized subject to some constraints. Formulating this problem into a mixed-integer reliability optimization problem, the component reliability level and the number of redundancies of both hardware and software components are to be determined.

To integrate software components into this optimization problem, two issues have to be investigated. First, a software reliability function and a software reliability-cost function have to be chosen so that they can be incorporated into the constraint function to represent the amount of resource required to reach a certain reliability level. Second, the reliability function of software redundancy with common-cause failure has to be determined so that it can be incorporated into the objective function of the optimization problem.

The software reliability-cost function represents the resources required to improve the reliability of the software. For the bug-counting model, software reliability is a function of the number of initial faults and debugging time. Thus, the cost of improving a software from one reliability level to another can be related to the number of faults removed during the debugging period and the debugging time.

Based on the Jelinski-Moranda model, the expected number of faults removed after debugging time t is

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

and the program failure rate after debugging time t is

$$\lambda(t) = N\phi e^{-\phi t}.$$

Representing debugging time in terms of failure rate,

$$\ln \lambda = \ln(N\phi) - \phi t$$

$$t = \frac{1}{\phi} [\ln(N\phi) - \ln \lambda].$$

Let the objective failure rate be λ^* . The debugging time t^* to reach λ^* can be represented as

$$t^* = \frac{1}{\phi} [\ln N\phi - \ln \lambda^*]$$

Also, the expected number of faults removed, μ^* , to reach λ^* can be represented as

$$\begin{aligned} \mu^*(t^*) &= N(1 - e^{-\phi t^*}) \\ &= N[1 - e^{-(\ln N\phi - \ln \lambda^*)}]. \end{aligned}$$

Let the current time be t and the current failure rate be λ , the extra debugging time and the extra faults removed to reach λ^* are

$$\Delta t = t^* - t = -\frac{1}{\phi}(\ln \lambda - \ln \lambda^*)$$

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

$$= N \cdot e^{-\ln N \phi} [e^{\ln \lambda} - e^{\ln \lambda^*}] \quad \lambda \geq \lambda^*$$

As indicated by Musa et al. [6], failure-identification personnel, failure-correction personnel, and computer time are the three cost factors involved in debugging. By associating the costs of failure-identification personnel and computer time to Δt , and the cost of failure-correction personnel to $\Delta \mu$, a software reliability-cost function can be formulated as follows.

$$RC(\lambda, \lambda^*) = (C_1 + C_3)\Delta t + C_2\Delta \mu$$

where

$RC(\lambda, \lambda^*)$ cost of reliability improvement from λ to λ^*

C_1 cost per unit time of the failure-identification personnel

C_2 cost per failure of the failure-correction personnel

C_3 cost per unit time of the computer.

In some cases, the reliability objective is based on the reliability level of a given operational time. For instance, the reliability objective is 0.98 for 100 operational hours. To formulate the reliability-cost function of this type, reliability can be represented as a function of debugging time plus operational time. Based on the Jelinski-Moranda model,

$$r(t+s) = e^{-\lambda(t)s} = \exp[-N\phi s e^{-\phi t}]$$

where

t	debugging time
s	operational time
$\lambda(t)$	program failure rate after t units of debugging time
$r(\cdot)$	reliability of a software component.

To represent t in terms of $r(t+s)$,

$$\ln r(t+s) = -N\phi s e^{-\phi t}$$

$$-\phi t = \ln \left[\frac{-\ln r(t+s)}{N\phi s} \right]$$

$$t = \frac{1}{\phi} \ln \left[\frac{N\phi s}{-\ln r(t+s)} \right].$$

Similarly, $\mu(t)$ can be represented in terms of $r(t+s)$. Hence,

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

$$= N \left[1 + \frac{\ln r (t+s)}{N\phi s} \right].$$

The debugging cost of improving the reliability from r to r^* can be expressed as

$$RC(r, r^*) = (C_1 + C_3)\Delta t + C_2\Delta\mu$$

where

$$\begin{aligned} \Delta t &= t^* - t \\ &= \frac{1}{\phi} [\ln s^* - \ln s + \ln(-\ln r) - \ln(-\ln r^*)] && \text{if } s \neq s^* \\ &= \frac{1}{\phi} [\ln(-\ln r) - \ln(-\ln r^*)] && \text{if } s = s^* \end{aligned}$$

$$\begin{aligned} \Delta\mu &= \mu(t^*) - \mu(t) \\ &= \frac{1}{\phi s} [\ln r^* - \ln r]. \end{aligned}$$

SOFTWARE REDUNDANCY

Besides debugging, adding redundancy is another way of improving the reliability of a software system. In software, redundancies are programs developed by different groups of people or different companies based on the same specifications. These programs are designed to perform the same function. In order to make the failures of the redundant copies to be as independent as possible, different computer languages, development tools, development methodologies, and testing strategies may be applied to different redundant programs.

Nevertheless, it has been shown that software redundancies are not totally independent [1,5]. Some input data will fail more than one redundancy because of the common errors made by different development teams. For example, errors in specifications, design, acceptance testing, or input data may cause multiple copies of software to fail. This partial independency of software redundancies can be represented by a common-cause model. Some specific common-cause models have been proposed, especially in the area of nuclear safety, to consider nature disasters or power shut-down [2,4,7]. The common-cause model for software redundancy is developed as follows.

Two-Component Model

A system with two partially independent software components in parallel is shown in Fig. 6.1. Due to the common-cause failure, this system can be transformed into a series system with two independent

components in parallel and a common-cause component as shown in Fig. 6.2. This two-component common-cause failure model has been addressed by Dhillon [2] for hardware systems.

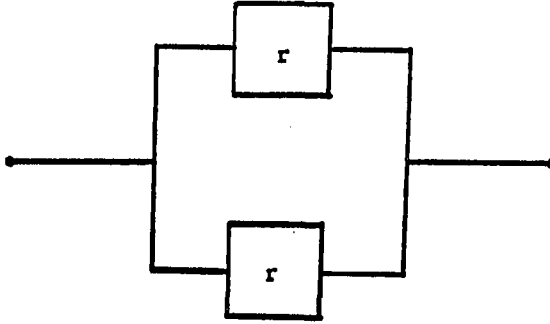


FIGURE 6.1. Two-component software redundancy

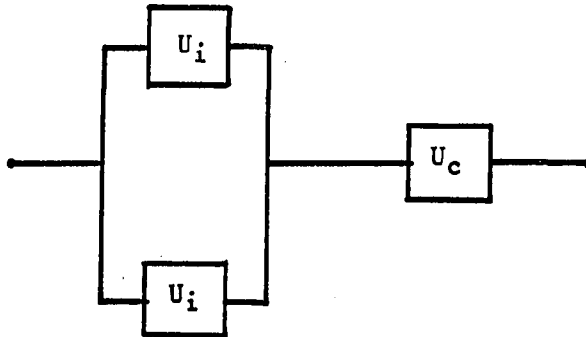


FIGURE 6.2. Transformed two-component software redundancy

The reliabilities of the independent component and the common-cause component can be derived as follows.

Notation:

λ failure rate of each software component

λ_i	failure rate of the independent component
λ_c	failure rate of the common-cause component
θ	common-cause ratio
r	reliability of each software component
U_i	reliability of the independent component
U_c	reliability of the common-cause component
R_s	system reliability

Let

$$\lambda = \lambda_i + \lambda_c$$

$$\theta = \lambda_c / \lambda.$$

Then

$$\lambda_c = \theta \lambda$$

$$\lambda_i = (1-\theta) \lambda$$

and

$$r(t) = e^{-\lambda t}$$

$$U_i(t) = e^{-\lambda_i t} = [r(t)]^{(1-\theta)}$$

$$U_c(t) = e^{-\lambda_c t} = [r(t)]^\theta.$$

The reliability of this two-component common-cause system can be expressed as

$$R_s = \left[1 - (1 - U_i)^2 \right] \cdot U_c$$

$$\left[1 - (1 - r^{1-\theta})^2 \right] \cdot r^\theta$$

Two-Component Markov Model With Common-Cause

A two-component Markov model with common-cause failure is shown in Fig. 6.3.

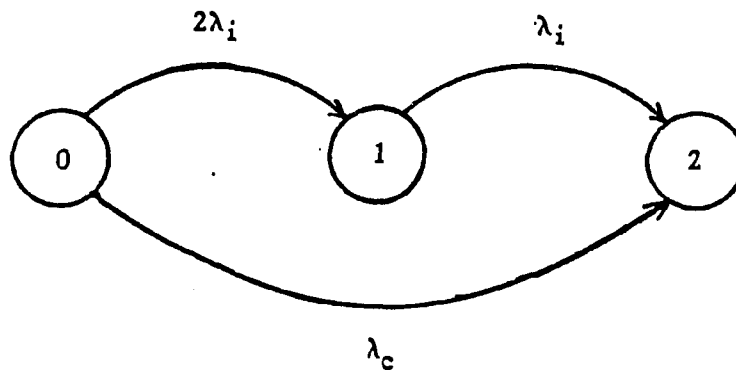


FIGURE 6.3. Two-component Markov model with common-cause failure

Let the state number of this Markov process be the number of components failed. The differential equations of this Markov process is

$$P_0'(t) = -(2\lambda_i + \lambda_c)P_0(t)$$

$$P_1'(t) = 2\lambda_i P_0(t) - \lambda_i P_1(t)$$

$$P_2'(t) = \lambda_c P_0(t) + \lambda_i P_1(t)$$

$$P_0(t) + P_1(t) + P_2(t) = 1$$

with initial condition $P_0(0)=1$.

Taking the Laplace transform,

$$sP_0(s) - 1 = -(2\lambda_i + \lambda_c)P_0(s)$$

$$P_0(s) = 1/(s + 2\lambda_i + \lambda_c)$$

and

$$sP_1(s) = 2\lambda_i P_0(s) - \lambda_i P_1(s)$$

$$P_1(s) = 2\lambda_i P_0(s)/(s + \lambda_i)$$

$$= A/(s + \lambda_i) + A/(s + 2\lambda_i + 2\lambda_c)$$

where

$$A = 2\lambda_i/(\lambda_i + \lambda_c).$$

Taking the inverse Laplace transform, the state probabilities are

$$P_0(t) = e^{-(2\lambda_i + \lambda_c)t}$$

and

$$P_1(t) = A \cdot [e^{-\lambda_i t} - e^{-(2\lambda_i + \lambda_c)t}].$$

The system reliability is

$$R_s(t) = P_0(t) + P_1(t)$$

$$= e^{-(2\lambda_i + \lambda_c)t} + A \cdot [e^{-\lambda_i t} - e^{-(2\lambda_i + \lambda_c)t}]$$

Three-Component Model

A system with three partially independent software components in parallel is shown in Fig. 6.4. Since some input data will cause one, two, or three components to fail, this system can be transformed into Fig. 6.5.

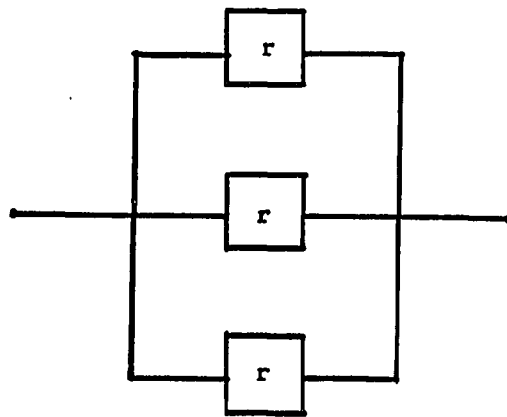


FIGURE 6.4. Three-component software redundancy

This transformation is based on the assumption that the failure rate of each software component can be broken down into an independent failure rate, a two-component common-cause failure rate, and a three-component common-cause failure rate. The system reliability can be derived as follows.

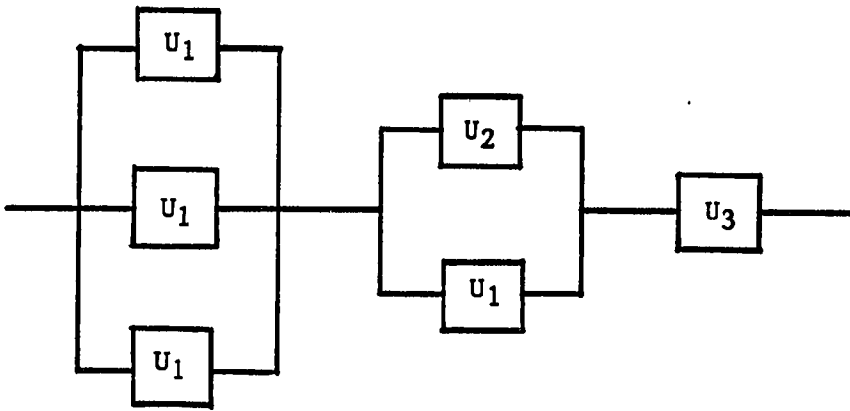


FIGURE 6.5. Transformed three-component software redundancy

$$\lambda = \lambda_1 + \lambda_2 + \lambda_3$$

$$\theta_k = \lambda_k / \lambda$$

$$r(t) = e^{-\lambda t}$$

$$U_k(t) = e^{-\lambda_k t} = r^{\theta_k} \quad k = 1, 2, 3.$$

where

λ_k k -component common-cause failure rate

U_k k -component common-cause stage reliability.

The system reliability is

$$\begin{aligned} R_s &= [1 - (1 - U_1)^3] [1 - (1 - U_1)(1 - U_2)] \cdot U_3 \\ &= [1 - (1 - r^{\theta_1})^3] [(1 - r^{\theta_2})(1 - r^{\theta_1})] \cdot r^{\theta_3}. \end{aligned}$$

Three-Component Markov Model With Common-Cause

Based on the same argument, a three-component Markov model with common-cause failure is shown in Fig. 6.6.

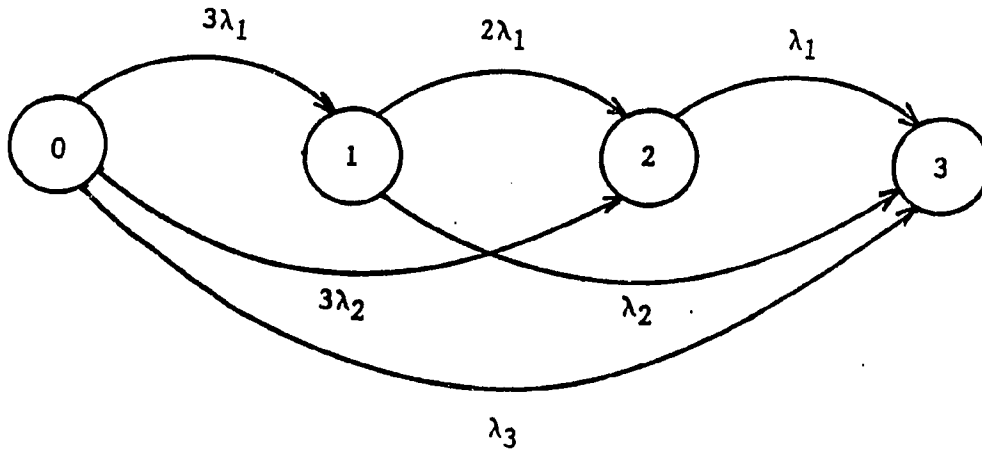


FIGURE 6.6. Three-component Markov model with common-cause failures

The differential equations and initial condition are as follows.

$$P_0'(t) = -(3\lambda_1 + 3\lambda_2 + \lambda_3)P_0(t)$$

$$P_1'(t) = 3\lambda_1 P_0(t) - (2\lambda_1 + \lambda_2)P_1(t)$$

$$P_2'(t) = 3\lambda_2 P_0(t) + 2\lambda_1 P_1(t) - \lambda_1 P_2(t)$$

$$P_3'(t) = \lambda_3 P_0(t) + \lambda_2 P_1(t) + \lambda_1 P_2(t)$$

$$P_0(t) + P_1(t) + P_2(t) + P_3(t) = 1$$

and initial condition $P_0(t)=1$.

Taking the Laplace transform,

$$sP_0(s) - 1 = -(3\lambda_1 + 3\lambda_2 + \lambda_3)P_0(s)$$

$$P_0(s) = 1/(s + 3\lambda_1 + 3\lambda_2 + \lambda_3)$$

and

$$sP_1(s) = 3\lambda_1 P_0(s) - (2\lambda_1 + \lambda_2)P_1(s)$$

$$P_1(s) = 3\lambda_1 P_0(s) / (s + 2\lambda_1 + \lambda_2)$$

$$= A/(s + 2\lambda_1 + \lambda_2) - A/(s + 3\lambda_1 + 3\lambda_2 + \lambda_3)$$

where

$$A = 3\lambda_1 / (\lambda_1 + 2\lambda_2 + \lambda_3).$$

And

$$sP_2(s) = 3\lambda_2 P_0(s) + 2\lambda_1 P_1(s) - \lambda_1 P_2(s)$$

$$P_2(s) = 3\lambda_2 P_0(s) / (s + \lambda_1) + 2\lambda_1 P_1(s) / (s + \lambda_1)$$

$$= B/(s + \lambda_1) - B/(s + 3\lambda_1 + 3\lambda_2 + \lambda_3) + C/(s + \lambda_1) - D/(s + 2\lambda_1 + \lambda_2) +$$

$$E/(s + 3\lambda_1 + 3\lambda_2 + \lambda_3)$$

where

$$B = 3\lambda_2 / (2\lambda_1 + 3\lambda_2 + \lambda_3)$$

$$C = 6\lambda_1^2 / [(\lambda_1 + \lambda_2)(2\lambda_1 + 3\lambda_2 + \lambda_3)]$$

$$D = 6\lambda_1^2 / [(\lambda_1 + \lambda_2)(\lambda_1 + 2\lambda_2 + \lambda_3)]$$

$$E = 6\lambda_1^2 / [(2\lambda_1 + 3\lambda_2 + \lambda_3)(\lambda_1 + 2\lambda_2 + \lambda_3)].$$

Taking the inverse Laplace transform, the state probabilities are

$$P_0(t) = \exp[-(3\lambda_1 + 3\lambda_2 + \lambda_3)t]$$

$$P_1(t) = A \cdot \{\exp[-(2\lambda_1 + \lambda_2)t] - \exp[-(3\lambda_1 + 3\lambda_2 + \lambda_3)t]\}$$

$$\begin{aligned} P_2(t) &= (B+C)\exp[-\lambda_1 t] - D \cdot \exp[-(2\lambda_1 + \lambda_2)t] - \\ &= (B-E)\exp[-(3\lambda_1 + 3\lambda_2 + \lambda_3)t]. \end{aligned}$$

The system reliability is

$$R_s(t) = P_0(t) + P_1(t) + P_2(t).$$

N-Component Model

Based on the same argument, an N-component system with common-cause can be transformed from Fig. 6.7 to Fig. 6.8.

The system reliability can be derived by defining

$$\lambda = \sum_{k=1}^N \lambda_k$$

$$\theta_k = \lambda_k / \lambda$$

$$U_k = r^{\theta_k} \quad k = 1, \dots, N.$$

The system reliability is

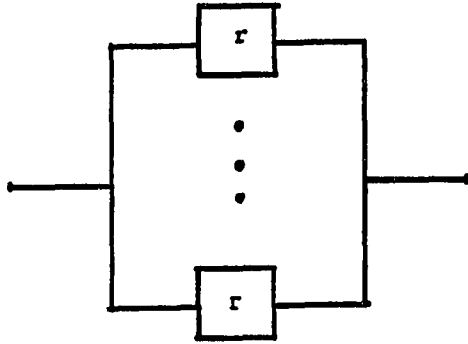


FIGURE 6.7. N-component software redundancy

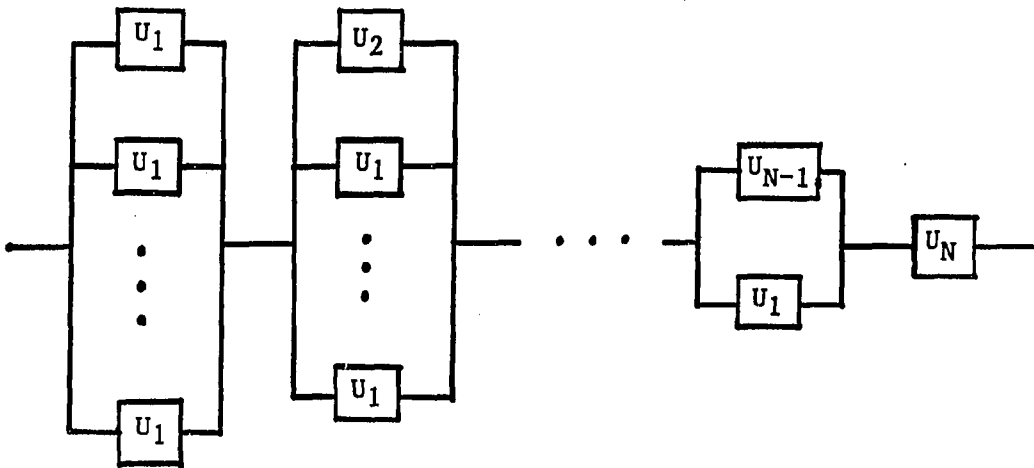


FIGURE 6.8. Transformed N-component software redundancy

$$R_s = \left[1 - (1-U_1)^N \right] \left[1 - (1-U_1)^{N-2}(1-U_2) \right] \cdots \left[1 - (1-U_1)(1-U_{N-1}) \right] U_N$$

$$= \prod_{k=1}^N \left[1 - (1-U_1)^{N-k}(1-U_k) \right].$$

Further assume that

$$\lambda_1 = a\lambda, \quad 0 \leq a \leq 1, \text{ for all } N$$

and

$$\lambda_2 = \beta\lambda_3 = \beta^2\lambda_4 = \cdots = \beta^{N-2}\lambda_N, \quad \beta \geq 1.$$

From the above assumptions, it can be shown that

$$\sum_{i=2}^N \lambda_i = (\beta^{N-1} - 1)\lambda_N$$

$$= \lambda - \lambda_1 = (1 - a)\lambda$$

$$\lambda_N = \frac{(1-a)\lambda}{\beta^{N-1} - 1}$$

$$\lambda_k = \beta^{N-k}\lambda_N$$

and

$$\theta_k = \frac{\lambda_k}{\lambda} = \frac{(1-a) \cdot \beta^{N-k}}{\beta^{N-1} - 1}$$

The system reliability can then be written as

$$R_s = \prod_{k=1}^N \left[1 - (1-U_1)^{N-k} (1-U_k) \right]$$

$$= \prod_{k=1}^N \left[1 - (1-r^{\alpha})^{N-k} (1-r^{\theta k}) \right].$$

N-Component Markov Model With Common-Cause

For a system of more than three software redundancies in parallel, it would be very difficult to estimate common-cause failure rate of two components, three components, etc. A simplified N-component Markov model is shown in Fig. 6.9. In this model, the common-cause failures cause all the redundancies to fail. This common-cause failure rate may represent the failure rate of system software whose failure will cause all the application software to fail.

The differential equations of this Markov process is

$$P_N'(t) = -(N\lambda + \lambda_c)P_N(t)$$

$$P_k'(t) = (k+1)\lambda P_{k+1}(t) - (k\lambda + \lambda_c)P_k(t) \quad k=N-1, \dots, 1$$

$$P_0'(t) = \lambda_c [P_1(t) + \dots + P_N(t)] + \lambda P_1(t)$$

$$\sum_{k=0}^N P_k(t) = 1$$

$$P_N(0) = 1.$$

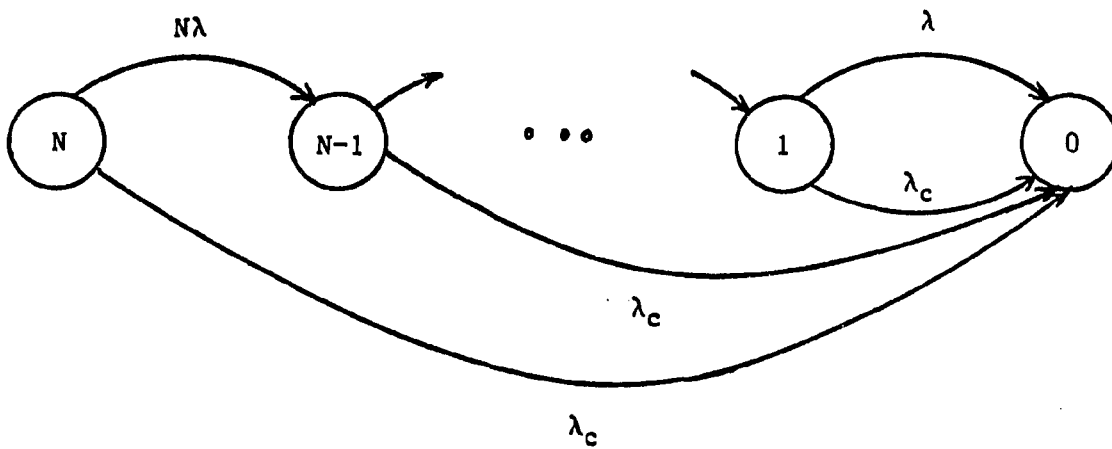


FIGURE 6.9. N-component Markov model with common-cause failure

Taking the Laplace transform and the inverse Laplace transform, the state probabilities can be derived as follows.

$$sP_N(s) - 1 = -(N\lambda + \lambda_c)P_N(s)$$

$$P_N(s) = 1/(s + N\lambda + \lambda_c),$$

then

$$P_N(t) = e^{-(N\lambda + \lambda_c)t}.$$

Also

$$N\lambda P_N(s) - [(N-1)\lambda + \lambda_c]P_{N-1}(s) = sP_{N-1}(s)$$

$$P_{N-1}(s) = N\lambda P_N(s)$$

$$= \frac{N}{s+(N-1)\lambda+\lambda_c} - \frac{N}{s+N\lambda+\lambda_c}$$

then

$$P_{N-1}(t) = Ne^{-[(N-1)\lambda+\lambda_c]t} - Ne^{-(N\lambda+\lambda_c)t}$$

Also

$$(N-1)\lambda P_{N-1}(s) - [(N-2)\lambda + \lambda_c]P_{N-2}(s) = sP_{N-2}(s)$$

$$\begin{aligned} P_{N-2}(s) &= \frac{(N-1)\lambda}{s+(N-2)\lambda+\lambda_c} \times \frac{1}{s+(N-1)\lambda+\lambda_c} \times \frac{N\lambda}{s+N\lambda+\lambda_c} \\ &= N(N-1) \left\{ \frac{1}{[s+(N-2)\lambda+\lambda_c]} (1)(2) + \frac{1}{[s+(N-1)\lambda+\lambda_c]} (-1)(1) + \right. \\ &\quad \left. \frac{1}{[s+N\lambda+\lambda_c]} (-1)(-2) \right\} \end{aligned}$$

and

$$\begin{aligned} P_1(s) &= N! \left\{ \frac{1}{[s+\lambda+\lambda_c]} (1)(2) \cdots (N-1) + \right. \\ &\quad \left. \frac{1}{[s+2\lambda+\lambda_c]} (-1)(1) \cdots (N-2) + \cdots + \right. \\ &\quad \left. \frac{1}{[s+N\lambda+\lambda_c]} (-1)(-2) \cdots [-(N-1)] \right\}. \end{aligned}$$

In general,

$$P_k(s) = \sum_{j=k}^N \left\{ \frac{N!}{k!} \left[\prod_{\substack{p=k \\ p \neq j}}^N (p-j) \right]^{-1} \frac{1}{s+j\lambda+\lambda_c} \right\}$$

and

$$P_k(t) = \sum_{j=k}^N \left\{ \frac{N!}{k!} \left[\prod_{\substack{p=k \\ p \neq j}}^N (p-j) \right]^{-1} e^{-(j\lambda + \lambda_c)t} \right\}.$$

The system reliability is

$$R(t) = \sum_{k=1}^{N-1} P_k(t).$$

The above derivation of state probabilities are exact forms but complex. An approximated form of system reliability is derived as follows.

$$\begin{aligned} P_0'(t) &= \lambda_c [P_1(t) + \dots + P_N(t)] + \lambda P_1(t) \\ &= \lambda_c [1 - P_0(t)] + \lambda P_1(t). \end{aligned}$$

If

$$\lambda P_1(t) \ll \lambda_c [1 - P_0(t)],$$

neglecting $\lambda P_1(t)$,

$$P_0'(t) = \lambda_c [1 - P_0(t)]$$

$$sP_0(s) = \lambda_c/s - \lambda_c P_0(s)$$

$$P_0(s) = 1/s - 1/(s + \lambda_c)$$

and

$$P_0(t) = 1 - e^{-\lambda_c t},$$

The approximated system reliability is

$$R(t) \approx 1 - P_0(t) = e^{-\lambda_c t}.$$

FORMULATION OF THE HARDWARE-SOFTWARE RELIABILITY OPTIMIZATION

To optimize the reliability of a hardware-software system, the reliability-redundancy allocation approach discussed in Sections II and III is applied. A general formulation of this problem is expressed as follows.

$$\text{Max } R_s(\bar{X}, \bar{R})$$

subject to

$$\sum_{j=1}^N g_{ij}(r_j, x_j) \leq b_i \quad \text{for all } i$$

When software components are involved, the above problem can be transformed into the following form.

$$\text{Max } R_s(R_1, \dots, R_N)$$

subject to

$$\sum_{j \in H} f_{ij}(r_j) \cdot h_{ij}(x_j) +$$

$$\sum_{j \in S} f_{ij}(r_j^0, r_j) \cdot h_{ij}(x_j) \leq b_i \quad \text{for all } i$$

where

- $R_j(r_j, x_j)$ reliability of stage j
 $g_{ij}(r_j, x_j)$ $f_{ij}(r_j) \cdot h_{ij}(x_j)$ or $f_{ij}(r_j^0, r_j) \cdot h_{ij}(x_j)$
 $f_{ij}(r_j)$ hardware reliability-cost function of resource i

	at stage j
$f_{ij}(r_j^0, r_j)$	software reliability-cost function of resource i at stage j
$h_{ij}(x_j)$	redundancy-cost function of resource i at stage j
H	set of hardware stages
S	set of software stages.

The objective function of the above formulation is represented in terms of the stage reliabilities. For hardware stage, the stage reliability is

$$R_j(r_j, x_j) = 1 - (1-r_j)^{x_j}.$$

For software stage, the stage reliability is

$$R_j = \prod_{k=1}^{x_j} [1 - (1 - r_j)^{x_j^{-k}} (1 - r_{jk})]$$

$$= \prod_{k=1}^{x_j} [1 - (r_j^a)^{x_j^{-k}} (1 - r_j^{\theta_k})].$$

where r_{jk} is U_k of the jth stage.

The constraint function is represented as the product of a reliability-cost function and a redundancy-cost function. For hardware components, an example of reliability-cost function used in Sections II and III is

$$r_j(t) = \exp[-\lambda_j t]$$

$$f(r_j) = a_j \left(\frac{-t}{\ln r_j} \right)^{\beta_j}.$$

For software components, the reliability-cost function is

$$f(r_j^0, r_j) = (C_1 + C_3)\Delta t + C_2\Delta\mu$$

where

$$\begin{aligned} \Delta t &= t^* - t \\ &= \frac{1}{\phi} [\ln(-\ln r_j^0) - \ln(-\ln r_j)] \end{aligned}$$

and

$$\begin{aligned} \Delta\mu &= \mu(t^*) - \mu(t) \\ &= \frac{1}{\phi s} [\ln r_j - \ln r_j^0]. \end{aligned}$$

The redundancy-cost function, $h_{ij}(x_j)$, depends upon the type of constraint involved. A constant function, increasing function, or decreasing function can be used as needed.

A NUMERICAL EXAMPLE

To express a N-stage series system, the three constraint functions used in Sections II and III are adapted for hardware stages. For software stages, further assume that

$$f_{1j}(r_j) = f_{3j}(r_j) = 1$$

$$h_{1j}(x_j) = p_0 + p_j x_j$$

$$h_{3j}(x_j) = w_j x_j \exp(x_j/4)$$

$$h_{2j}(x_j) = x_j$$

The hardware-software reliability optimization problem can be expressed as

$$\begin{aligned} \text{Max } R_S(\bar{X}, \bar{R}) &= \prod_{j=1}^N R_j(x_j, r_j) \\ &= \prod_{j \in H} [1 - (1-r_j)^{x_j}] \cdot \prod_{j \in S} \prod_{k=0}^{x_j} [1 - (1-r_j^a)^{x_j-k} (1-r_j^{\theta_k})] \end{aligned}$$

subject to

$$\sum_{j \in H} p_j x_j^2 + \sum_{j \in S} (p_0 + p_j x_j) \leq P$$

$$\sum_{j \in H} a_j \left(-t / \ln r_j \right)^{\beta_j} (x_j + \exp(x_j/4)) +$$

$$\sum_{j \in S} \{ x_j (C_1 + C_3) [\ln(-\ln r_j^0) - \ln(-\ln r_j)] / \phi +$$

$$x_j C_2 [\ln r_j - \ln r_j^0] / \phi s \leq C$$

$$\sum_{j=1}^N w_j x_j \exp(x_j/4) \leq W$$

A numerical example of a 5-stage series system was solved. Stages 1 through 4 are hardware stages, while stage 5 is software stage. With the data given in Table 6.1, the problem was solved by the eight combination methods discussed in Section II using two initial base points, 0.5 and 0.7. The best solution among the 16 answers, as shown in Table 6.2, was obtained by the combination of H-J search method and the G-A-G redundancy method.

TABLE 6.1. Data for numerical example

j	a_j	P_j	W_j	P	C	W
1	2.33×10^{-5}	1	7			
2	1.45×10^{-5}	2	8			
3	5.41×10^{-6}	3	8	110	175	200
4	8.05×10^{-5}	4	6			
5	-	4	9			
$\beta_j = 1.5,$		$j = 1, 2, 3, 4, 5$		$t = 1000$		
$C_1 = 0.015$		$C_2 = 50$	$C_3 = 0.01$	$p_0 = 3$		
$\phi = 0.0012$		$r_j^0 = 0.9$	$t = 100$ (CPU time for software)			
$\alpha = 0.8$		$\beta = 2$				

TABLE 6.2. Result of the numerical example

\bar{R}	(0.8672, 0.94, 0.94, 0.82, 0.90)
\bar{X}	(3, 2, 2, 3, 3)
R_j	(0.9976, 0.9964, 0.9964, 0.9942, 0.9789)
R_s	0.9640

REFERENCES

1. Echhardt, D. E. Jr. and L. D. Lee. "A theoretical basis for the analysis of multiversion software subject to coincident errors." IEEE Trans. Software Engineering, SE-11, No. 12, 1985, 1511-1517.
2. Dhillon, B. S. Reliability Engineering in System Design and Operation. Van Nostrand, New York, 1983.
3. Dhillon, B. S. and J. Natesan. "Moments of N-unit redundant systems with time dependent failure rate." Microelectronics and Reliability, 23, No. 1, 1983, 61-69.
4. Dhillon, B. S. "On common-cause failures - bibliography." Microelectronics and Reliability, 18, 1979, 533-534.
5. Knight, J. C. and N. G. Leveson. "An experimental evaluation of the assumption of independence in multi-version programming." IEEE Trans. software Engineering, SE-12, No. 1, 1986, 96-109.
6. Musa, J. D., A. Iannino, and K. Okumoto. Software Reliability: Measurement, Prediction, Application, to be published by McGraw-Hill, New York, 1987.
7. Hecht, H. and H. Dussault. "Correlated failures in fault-tolerant computers." IEEE Trans. Reliability, R-36, No. 2, 1987, 171-175.

SECTION VII. CONCLUSIONS

CONCLUSIONS AND SUGGESTIONS FOR FUTURE STUDY

After two decades of development, reliability optimization has become a branch of reliability engineering. This is due to the increasing need of a highly reliable system and the fact of limited resources. This research formulates the reliability optimization into a mixed-integer programming problem which determines both the number of redundancies to be used and the component reliability levels. This formulation unifies the traditional approach of dealing with only redundancy or reliability. Although this extension is obvious, only three papers have been published on this specific topic since 1973. This is understandable because of the difficulty of the problem and the suspicion of the realism of the problem. This dissertation provides part of the answer to the above two questions by proposing two techniques for solving the mixed-integer reliability optimization problem and discussing a hardware-software system which matches this formulation very well.

In order to integrate software into this reliability optimization problem, software reliability models, software reliability-cost function, software redundancy, and reliability costs in software life-cycle models are investigated. These studies pave the way for the future study in software reliability and systems reliability with software components. Suggestions for future studies are discussed below.

1. New methods of solving the mixed-integer reliability optimization problem should be investigated. The combination method discussed in Section II provides no information about global optimality because of the heuristic approach used. Although very general, it is not efficient because of the iterative trials of the sequential search method. More efficient and effective search methods can be studied to take advantage of the features of the reliability problem.

The Lagrange multiplier and branch-and-bound method is more accurate, but suffers from numerical instability in solving the simultaneous nonlinear equations. In the branch-and-bound stage, the branching variables are fixed, once an integer solution is obtained, rather than carried over to the subsequent problems as constraints. This proposed method helps the problem from becoming bigger and bigger which in turn would increase the difficulty of solving the nonlinear simultaneous equations, but a better solution may be missed by this heuristic. More studies can be done to investigate the significance of improvement in the final solution and the extra effort taken to carry over the branching variables.

2. The integration of software and hardware components discussed in Section VI includes only the time-domain

software reliability model. Future studies shall be extended to the input-domain software reliability model. Expressing hardware reliability using input-domain concept can also be attempted.

3. A system including a number of hardware components and software components, or a software system consisting a number of modules can be regarded as a network system. Techniques developed for network reliability can be applied to software system as well.
4. Software reliability models have been criticized for their difficulty of being understood and implemented. The difficulty arises from the reliability theory behind these models. In implementation, the testing strategies must conform with the model assumptions, which frequently is in conflict with common practice. To make software reliability models easier to use, more study should be done to accommodate software reliability models to the testing strategies. Another direction is to extend the applicability of the model to encompass the entire software life cycle.
5. In Section VI, software reliability-cost function is assumed to be a linear function of the debugging time and the number of faults removed. This function should be validated if real data are available. The validation of software reliability models also deserve more investigation.

6. Most software reliability models discussed in Section IV and reliability-cost function discussed in Sections V and VI are centered around the concept of bug-counting. In some cases, failures cannot be traced back to a fault (incorrect logic, incorrect statement, missing statement, etc.) in the program. For instance, slow response and numerical error may require the whole module to be rewritten using a new algorithm. In this case, even though the number of failures can be counted, counting the number of faults in the program would be misleading. Software reliability models for this type of situation should be studied.

ACKNOWLEDGEMENTS

I am particularly grateful to my major professor, Dr. Way Kuo, whose continuous support and encouragement through this work have made this dissertation possible. I consider myself fortunate to have worked with such a respectable educator. His persistence in high quality research, devotion to education, kindness, and wisdom are admired and respected.

Each of the members of the committee made valuable contribution to this work and my six years of education at Iowa State University. Dr. Roger W. Berger, chairman of my master's thesis work, introduced me to the areas of Information Systems, Software, and Microcomputer which stimulated my interest in Software Reliability. Dr. Herbert T. David and William Q. Meeker Jr.'s course in Reliability is an important theoretical guidance to this work. Dr. John Even's Queueing Theory and Dr. Mervyn G. Marasinghe's Statistical Computation have been very helpful in formulating and solving problems in this dissertation.

A special thanks shall be given to my wife, Hsueh-Foo, for her love, sacrifice, and support. Thanks are also extended to my sons, John and Albert, and my parents for their understanding and support during this research.