1989

Software reliability optimization by redundancy and software quality management

Dong Hae Chi
Iowa State University

Software reliability optimization by redundancy and software quality management

Chi, Dong Hae, Ph.D.

Iowa State University, 1989

Software reliability optimization by redundancy and software quality management

by

Dong Hae Chi

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major: Industrial Engineering

Approved:

In Charge of Major Work

For the Major Department

For the Graduate College

Iowa State University
Ames, Iowa
1989

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 GENERAL INTRODUCTION

1.1 Abstract

This study investigates both the trade-offs among system reliability improvement, resource consumption, and other relevant constraints, and the application of statistical control methods to monitor variations. A process for reliability-related quality programming is developed to fill existing gaps in software design and development so that a quality programming plan can be achieved. A software reliability-to-cost relation is developed both from a software reliability-related cost model and software redundancy models with common-cause failures. The software reliability optimization problem will be formulated into a mixed-integer programming problem and solved by a branch-and-bound technique.

A procedure will be developed to identify, define, develop, and demonstrate a quality performance measure to improve system operation that is based on statistical control methods. Despite the most painful effort to control product quality, variation in product quality is unavoidable. Through the use of process control techniques, such as statistical control chart, unusual variations in the software development process can be controlled and reduced.

1.2 Research Problem

Software technology has been criticized for its high cost, low reliability, and frequent delays. Forty percent of software development costs are spent in testing to remove errors [9] and assure high quality, but in fact, high cost and delays are still cited as the results of low reliability. By focusing on the overall system, we can improve low system reliability (1) by debugging the program or (2) by adding redundant components. Module testing, integration testing, and field testing represent the first approach, while N-version programming, recovery block, redundant data structure, and redundant data storage are examples of the second approach.

The techniques of using more reliable components and adding redundancies to improve system reliability have been widely used in hardware systems. Nevertheless, software differs from hardware in terms of failure causes and reliability modeling measures. Therefore, the conventional techniques for modeling hardware systems cannot be directly applied to software performance modeling. Because many systems include a significant proportion of software and because over sixty percent of the system life-cycle cost has been spent on software-related factors, there is an urgent need to evaluate the performance of integrated software modules to meet optimal design specifications. This is, however, a sophisticated task because

- the system has many restrictions, such as cost, manpower, management, scheduling, processing time, computer memory, facilities
- no methodology addresses and monitors software quality and development
- no dynamic optimization procedure exists to locate solutions for a complicated mixed-integer-type programming problem

- no systematic and generic protocol can be used to evaluate and feed back performance of quality programming.

1.3 Objectives of The Research

The objective of this study is to perform optimally a complete software life-cycle analysis incorporating the principles of optimization and statistical quality control. The research consists of the following two topics.

1. Optimal Allocation of Software Reliability and Redundancy

   To integrate software components into an optimization problem, The following issues must be investigated.

   - provide reliability-related quality programming process
   - predict system performance
   - develop the software reliability-related cost function
   - develop the software redundancy model with common cause failures
   - formulate the software reliability optimization
   - derive other reliability-related resources function
   - optimize reliability-redundancy allocation

2. Software Quality Management

   Through the use of process control techniques, the variation in the software development process can be controlled and reduced. To set up *a procedure* to identify, define, monitor, and control software quality, the following must be investigated.

- plan the statistical software quality control procedure with each specified step related to a development activity
- investigate input domain testing process
- use of statistical quality control techniques
- specify the software quality variation outcomes

2 AN OVERVIEW OF QUALITY PROGRAMMING

Quality programming is a means to perform optimally a complete software life-cycle analysis incorporating the principles of optimization and statistical quality control. A diagram of the reliability-related quality programming process is depicted in Figure 2.1. In the following, those development phases of Figure 2.1 that are not covered in current software practice will be discussed in order to perform optimally a complete software life-cycle analysis that incorporates the principles of optimization.

2.1 Modeling

Modeling is the first and most important step in quality programming development process. In modeling phase, an accurate picture of the problem must be developed to gain as broad a perspective of the problem as possible at the outset. All aspects of input, output, and processing must be studied carefully to prevent the original problem from being destroyed by misleading opinions, considerable irrelevant information.

It is the study of all the factors necessary to understand the problem, to generate a quality solution, and to allow the use of statistical quality control. A model for software development is like a model performed in the manufacturing industries. The modeling factors discussed by [12] are:

- Modeling of inputs
  - types of inputs
  - characteristics of each type of input
  - rules for constructing inputs
  - sources of inputs
- Modeling of outputs
  - output description
  - output prototype design
  - output strategies
  - output quality planning
- Modeling of software
  - process description
  - rules of using inputs
  - methods of producing outputs
  - data flows in a process being automated
  - process control
  - software characteristics
  - methods of developing the software system

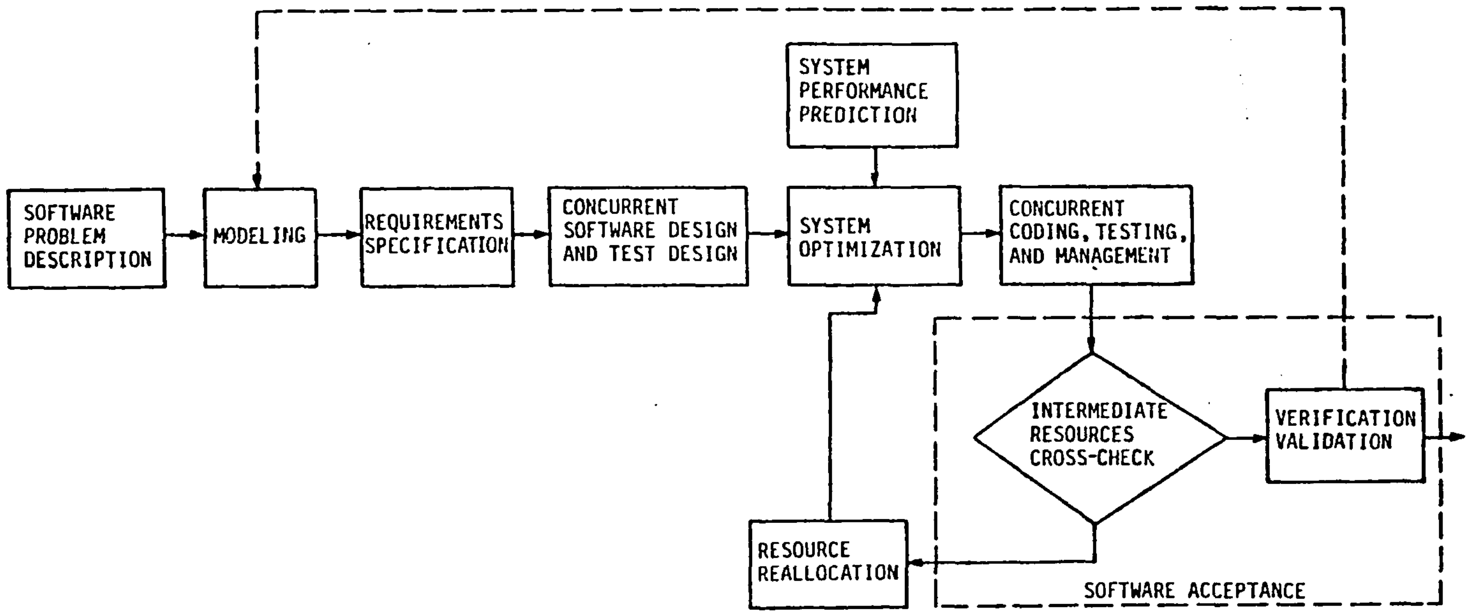

SOFTWARE PROBLEM DESCRIPTION
MODELING
REQUIREMENTS SPECIFICATION
CONCURRENT SOFTWARE DESIGN AND TEST DESIGN
SYSTEM PERFORMANCE PREDICTION
SYSTEM OPTIMIZATION
CONCURRENT CODING, TESTING, AND MANAGEMENT
INTERMEDIATE RESOURCES CROSS-CHECK
VERIFICATION VALIDATION
RESOURCE REALLOCATION
SOFTWARE ACCEPTANCE

Figure 2.1: A process for reliability-related quality programming

2.2 Requirement Specification

The second step in a quality programming development process is the requirement specification. In this phase, the problem should be analyzed by a step-by-step procedure and documented in detail to cover all necessary requirements and to obtain detailed qualitative and quantitative characteristics of these inputs and outputs.

The result of the requirement specification phase of quality programming development must be a formal document that completely describes the solution, using both words and diagrams. This document can be used to communicate to the programmer, software designer, test designer, system optimizer, failure-identification personnel, failure-correction personnel, user, and other concerned parties.

The volume of the document varies dramatically from software to software, depending on system complexity, size, and contractual requirements. The requirement specification activity includes software requirements, test requirements, and documentation requirements [12].

2.2.1 Software requirement

In the modeling phase it was sufficient to understand and identify the input, processing, and output quantities. The next software requirement phase should specify the detailed input, processing, and output requirements for design of the software. In conventional practice, the requirements for the following equalities has been poorly or insufficiently specified.

- Input description is the nature or extent of data

- Definition of product unit is the user's detailed output requirement
- Product unit defectiveness is the criterion of acceptability.

The above requirement specifications should be stated carefully by the statistical quality control.

2.2.2 Test requirement

specification of test methods The test methods are regular test, weighted test, boundary test, invalid test, and special test method. A combination of the methods is required to conduct the tests.

statistical inference requirements The user should require that proper data be collected in order to perform the necessary statistical tests.

statistical sampling methods The user should specify the most appropriate statistical sampling methods consistent with the product unit definitions developed as part of the modeling activity. A sampling process for estimating the defective rate of the product unit and another sampling process for accepting the software product unit should be used.

software acceptance criteria How good the product unit population must be and how thorough the system testing must be to satisfy the developer and the user that the software is acceptable and has been sufficiently tested.

The above factors should be carefully addressed and specified for both system test requirement and module test requirement. By doing this, both user and developer would have statistical evidence that quality is built into the software.

2.2.3 Documentation requirement

The type of documents should be identified and specified in detail. The product description, in textual or blueprint form, written instruction (process description) are fundamental tools to help ensure the understandability and quality of software.

2.3 System Performance Prediction

Use of data from the past history of similar environments can help predict the results of future experiments. The data, called the *indices of performance*, should be provided in order to conduct system performance prediction. The management realizes that all production personnel are part of the system and so are their problems. Therefore, an effort to collect the *indices of performance* should be done in advance to improve future product. On the basis of the past *indices of performance*, the system failure intensity or system failure rate under a specified condition can be predicted.

2.4 System Pre-optimization

The number of redundancies of each subsystem needs to be determined before the design phase begins; this is because all redundancies are supposed to developed independently from the design phase. By the use of data estimated from the system performance prediction phase, the system optimization problem can be formulated and solved. The procedure of pre-optimization is the same as that of main system optimization which is conducted in the middle of coding, testing, and management phase. A solution obtained at the system pre-optimization phase gives management a general idea of system design.

2.5 Concurrent Software Design and Test Design

The concurrent development for both software design and test design is advantageous because it allows cross-checking of the designs as early as possible and it can reduce the development time considerably.

2.5.1 Software design

The designer must keep in mind the software engineering goals of modifiability, understandability, reliability, and efficiency as he or she proceeds with the software design. The software engineering principles of abstract data typing, information hiding, modularization, localization, uniformity, completeness, conformability, and statistical quality control must be observed carefully in developing the design [12].

In this study, a general guideline for producing a quality software design, including numerous design tools and techniques, is shortly discussed. Because design is a very personalized and highly interactive process we shall leave the choice of these tools and techniques to the reader.

2.5.1.1 Modern software design methods

top-down design The characteristics are:

- At each level, the details of the design at lower levels are hidden. Only the necessary data and control are defined.
- Make a module small enough that it is within a programmer's intellectual span of control (about 50 lines).

- The design error will not be discovered until the end of the design process.

structured programming The characteristic is the use of a single-entry and a single-exit control structure to provide a well-defined, clear, and simple approach to program design. Since it eliminates GO TO statements completely, the program structure is often vastly complicated and sometimes makes the running time longer. The type of structured programming are SEQUENCE, IF THEN ELSE, DO WHILE, and so on.

modular design A module means a modest-sized subprogram which performs independently on specific function. A top-down design results in a modular design.

2.5.1.2 Design representations techniques There are almost 18 different techniques so that a group of techniques is commonly used for designing a software system.

flow charts There are two types of flow charts. First, the high-level flow chart is used to represent the flow of the logic. The high-level flow chart contains only control structures. Second, the detailed flow charts is used for the detail of logic. Each symbol of the detailed flow chart represents a single line of code.

pseudo-code (metacode) Pseudo-code which consists of a shorthand notation for control structure is a detailed subsection of high-level flowchart. Therefore, pseudo-code technique is more flexible and clear than flow chart technique.

HIPO diagrams Hierarchy plus Input-Process-Output diagram consists of one H block diagram and a set of the overview IPO and detailed IPO diagram.

Warnier-Orr Diagram This diagram utilizes nested sets of braces, some pseudo-code, logic symbols.

A recommended design representation technique [61] is:

- An H diagram is drawn and major subprograms are identified.
- High-level flow charts are drawn for control structures and each major sub-program.
- Pseudo-code is written for each flowchart.
- The program (code) is written in the source language.

2.5.2 Test design

When the software modeling and requirements specification documents are sufficiently prepared, the necessary procedure for test design is to review and refine those documents. This procedure is applicable both to the entire software system and to the modules of the system.

2.6 Concurrent Coding, Testing, and Management

Each system module should be tested with the statistical quality control tool as soon as it is coded, and therefore the system can be built on a "secure-quality-module" basis. By using the SIAD (Symbolic Input Attribute Decomposition) tree we can represent the input domain in a convenient form and can easily trace back the location of faults. Moreover, we may use the various statistical control charts to find system variation. Once the variation of system is detected, the assignable causes (especially common cause) of variation need to be identified and eliminated by the use of cause and effect diagram.

2.7 System Optimization

In developing a fault-tolerant software, the software engineers have to consider the trade-offs among reliability improvement, resource consumption, and other relevant constraints. An optimal design is needed to maximize the system reliability under the restricted resources. The system optimization problem is formulated subject to various restricted resources. The value of decision variables (component's reliabilities and the number of redundancies of each subsystem) can be determined when the system optimization problem is solved. The predetermined value of decision variables may be varied later as the development phase moves forward.

2.8 Software Acceptance

In the middle of concurrent coding and test phase, the system and component reliabilities are examined. Since more information about the developing software, such as failure intensity or failure rate, are readily available at this time, more accurate system and component reliabilities can be reevaluated with updated data.

If the system and module reliabilities do not meet the reliabilities required, reallocate the resources and solve the system optimization problem again with the updated data. A set of solutions along with determined decision variables will be obtained. The management chooses a solution among the new multi-optimal solutions obtained. The decision to be made is whether to improve module's reliabilities or to increase the number of redundancies of some modules through manageable ways. This iteration continues until the current reliabilities meet the requirement.

2.9 Resource Reallocation

It is obvious that the component's failure rates of different stages (subsystems) are different. When an optimal solution is chosen, each component has its own projected reliability. Since the failure rates of each component are different, the time required to reach the projected component reliability is also different. Residual resources should be reallocated on the basis of those times required. Assigning accurate amount of resources to each stage at the beginning can eventually save development cost, time, and efforts.

3 SYSTEM PERFORMANCE PREDICTION

As mentioned in the previous chapter, the main objective of the system optimization for quality programming is to determine the number of redundancies and reliability of each subsystem under the given various resources available. The management is supposed to choose a solution from among the new multi-optimal solutions obtained at the system optimization stage. To make the management's decision reasonable and efficient, the number of redundancies of each subsystem needs to be determined before the design phase begins; this is because all redundancies are supposed to be developed independently from design phase. The initial decision on the number of redundancies of each subsystem doesn't need to be very accurate, but close enough (1) to increase the number of redundancies later without reallocating the major man power for new redundancies, or (2) to decrease the number of redundancies without wasting major effort.

The purpose of this chapter is to describe techniques that can be used to predict the system performance (e.g., failure rate) at the end of specification phase. As computer scientists try to analyze the software problem and the quality of the product, one of their first steps in the solution is to measure the software's complexity. Many attempts to quantify the complexity of software have been made [6,5,8,11].

Belady, in his survey on complexity, listed over 60 techniques which have appeared in the literature [8]. Section 3.2 will discuss some important techniques and focus in depth on Halstead's equation. Section 3.3 will discuss how can the complexity be converted to the number of errors in the program and how does the estimated initial number of errors relate to the system performance prediction. Finally, the indices of performance which makes the estimation of the system performance closer to the true software system performance will be discussed in Section 3.4.

3.1 Notation

a	slope of the line in Fig. 3.1
c	constant
E	effort measure
f_r	relative frequency of occurrence for type r
H	information content
N	program length (total operators plus operands) in Halstead length equation
n	total tokens
n_r	frequency of occurrence of rank r
r	rank
t	number of distinct types of operators plus that of operands
η_1	number of distinct operator types appearing in an algorithm
η_2	number of distinct operand types appearing in an algorithm

3.2 Review of Complexity

Types of complexity are (1) size or bulk, which can be measured by the number of instructions; (2) difficulty of text, which can be measured by the number of different type of operators and operands; (3) structural complexity, which can be measured by the graph properties of control structure; and (4) intellectual complexity, which can be measured by the algorithmic difficulty.

3.2.1 Zipf's law of natural language

Before the Halstead's equation is discussed it is better to check the background of program length estimation, so called, the Zipf's law of natural language. Laws of this nature were first studied by Zipf in connection with natural languages. He studied the relationship between frequency of occurrence n_r and rank r for words from English, Chinese, and the Latin of Plautus. The relationship between n_r and r is depicted in Fig 3.1. Derivation of length equation (n) is as follows:

$$\begin{aligned} \log f_r &= \log c - a \log r \\ \log f_r \cdot r^a &= \log c \\ f_r \cdot r^a &= c\ (constant). \end{aligned}$$

If a = 1, then

$$\begin{aligned} f_r \cdot r &= c \\ \frac{n_r}{n} \cdot r &= c \\ n_r &= \frac{cn}{r}. \end{aligned} \tag{3.1}$$

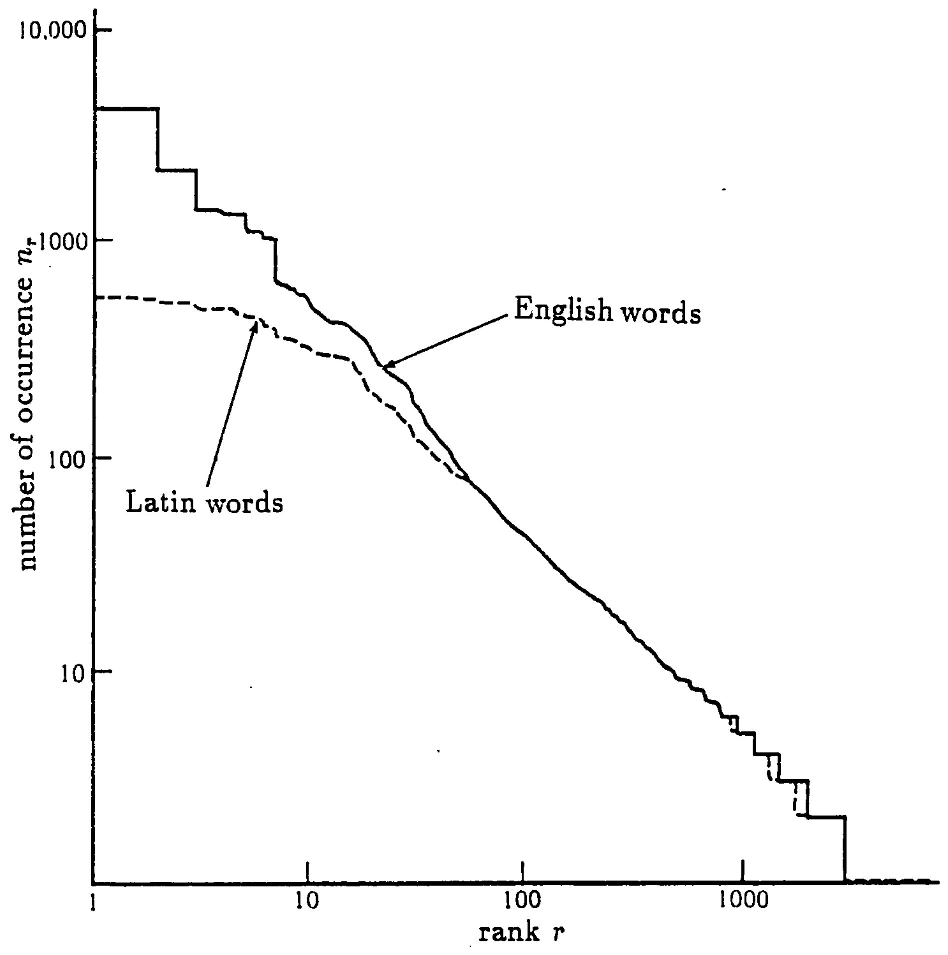


Figure 3.1: Occurrences frequency vs. rank

If we take the summation of both sides of Eq. 3.1 we get

$$\sum_{r=1}^{t} n_r = cn \sum_{r=1}^{t} \frac{1}{r}. \tag{3.2}$$

The summation of the series 1/r is given as follows:

$$\sum_{r=1}^{t} \frac{1}{r} = 0.5772 + \ln t + \frac{1}{2t} - \frac{1}{12t(t+1)} + \dots \tag{3.3}$$

Substitution of Eq. 3.3 (retaining only two terms for modest-sized t) into Eq. 3.2 yields an expression for the constant c in terms of t.

$$c = \frac{1}{0.5772 + \ln t} \tag{3.4}$$

In most cases the rarest type will occur only once, in other words, $n_{r_{max}} = 1$. Since $n_r = 1$ and $r = t$, we can get another equation for constant c from Eq. 3.1.

$$c = \frac{t}{n} \tag{3.5}$$

By substituting Eq. 3.5 into Eq. 3.4 we can get a length equation in terms of t.

$$n = t(0.5772 + \ln t) \tag{3.6}$$

This equation tells that if we know the number of distinct type t, we can estimate the number of total tokens n.

(Example) Let's estimate the length of an article which has 200 different word types.

$$n = 200(0.5772 + \ln 100) = 1037 \qquad \text{words long.}$$

(Zipf's second law) In the tail of Zipf's law, there are generally several identical n_r values which make plateau of t_k types, each with the same n_r. Therefore, the

Zipf's first law can be modified as follows:

$$n = \frac{6}{\pi^2}(0.5772 + \ln t)t \tag{3.7}$$

Shooman and Laemmel [63] have shown that Zipf's law length equation has about 25% overall agreement in the estimation of program length for several software examples. Although it is not a good idea to use Zipf's law for the prediction of system reliability, this law is simple to use and gives a general concept for estimating program length early in the design phase.

3.2.2 Halstead length equation

Halstead [28] in his work found that for a nontrivial class of algorithms there is a quantitative relationship between operators and operands and their usage. He assumed that a program is a sequence of symbols, made up of alternating operator and operand symbols. In other words, the program can be generated by a stochastic process.

The procedure of Halstead length equation generation will be introduced in final report.

Halstead length equation is

$$N = \eta_1 \log_2^{\eta_1} + \eta_2 \log_2^{\eta_2} \tag{3.8}$$

Halstead's measurements are somewhat closer than those of Zipf. The overall agreement is about 14%. More data on significant sized real-world programs should be used to investigate the accuracy of the Halstead length equation. However, this equation can play an even more important role if it can be used for estimating program performance early in the design process.

3.2.3 Other complexity models

Other complexity models are Halstead's effort function (E), Shannon's information theory, and the graphic complexity model, and the like.

- $E = \frac{2H^2}{H^\star}$
- $H = \log_2^i$
- cyclomatic complexity
- knot complexity
- polynomial complexity

These deterministic models empirically measure the qualitative attributes of a software and are used in the early phase of the software life cycle to predict the number of errors in a program.

3.3 Complexity vs. Number of Errors

Complexity measurement estimates and predicts the number of errors in the program. Four different hypotheses necessary to convert complexity measure to number of errors are:

- Length hypothesis: the number of bugs per statement (e.g., machine language statement is equal to one operator plus one operand).
- Information hypothesis: the number of bugs per information content (H).
- Effort measure: the number of bugs per effort (E).

- Akiyama's hypothesis: the number of bugs per the number of decisions plus the number of subroutine calls.

The proportionality constants should be carefully calculated.

3.4 The Indices of Performance

For more accurate estimation of system performance, other facts, aside from the program length, should be considered. The indices of performance related to the program complexity itself are

- program length
- language level
- interrelationship among instruction
- others

The indices of performance related to the developers are:

- skill (personal working experience)
- efforts (team communication, etc.)
- consistency
- others

An effort to collect the *indices of performance* should be made in advance to improve the future product. At the end of the specification phase, the system failure intensity or system failure rate under a specified condition can be approximately predicted on the basis of the past *indices of performance.* A set of data obtained from the system performance prediction phase can be used to optimize the system.

An optimal solution obtained from the system pre-optimization phase could be different from the one obtained from the system optimization phase. However, the management can have a general idea of system design before the design phase begins. A set of equations which make the estimation of the system performance closer to the true system performance of underdeveloping software need to be investigated further.

4 OPTIMAL ALLOCATION OF SOFTWARE RELIABILITY AND REDUNDANCY

To integrate software components into an optimization problem, two issues should be investigated. First, a software reliability-related cost function has to be chosen so that components can be incorporated into the constraint function to represent the amount of resources 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 following notation will be used in this chapter.

4.1 Notation

a,b	unknown parameters of nonhomogeneous Poisson model
b_i	amount of resource i available
C_1	cost per unit calendar time associated with failure detection
C_2	cost per unit calendar time associated with failure elimination
$f_{ij}(r_j, r_j^*)$	software reliability cost function of resource i at stage j
$f_{ij}(r_j)$	hardware reliability cost function of resource i at stage j
H,S	set of hardware and software stages, respectively
$h_{ij}(x_j)$	redundancy cost function of resource i at stage j

k	redundant component cost coefficient
N	initial number of bugs in program
P_k	probability of being in state k in Markov model
$R_j(r_j, x_j)$	reliability of stage j
RC(λ,$\lambda^\star$)	cost of reliability improvement from λ to $\lambda^\star$
r	denotes reliability in general
r_j,$r_j^\star$	reliability and projected reliability of stage j, respectively
s	operational time, $s > 0$
t	debugging time
t_r	resource usage parameter per CPU hr (person hr/CPU hr)
x_j	number of components at stage j
α,β	failure rate ratio
Λ_i	the i^{th} Lagrange multipliers
λ,$\lambda^\star$	current and projected failure rate, respectively
λ_1	failure rate of the independent component
	($\lambda_1 = \lambda_A = \lambda_B$, in the two component case)
λ_i	failure rate of the common-cause of i components
λ(t)	program failure rate after t units of debugging time
μ(t),m(t)	expected number of faults removed after t units of debugging time
μ_r	resource usage parameter per failure (person hr/failure)
Φ	unknown parameter of JM model
—	denotes a vector

4.2 Review of Software Reliability Model

The definition of software reliability chosen is the one offered by Boehm [9].

> Software possesses reliability to the extent that it can be expected to perform its intended functions satisfactory.

The objectives of this survey of the software reliability model may be summarized by the following:

- Determine what software structural and development characteristics are available for analysis of software reliability.
- Define improved methods for collecting reliability data.
- Based on error histories seen in the data, define sets of error categories.
- Perform a survey of existing software reliability models.

4.2.1 Software reliability vs. hardware reliability

Because the basic modeling techniques of software reliability are adapted from reliability theory developed for hardware systems, a comparison of software reliability and hardware reliability help in the use of these theories and in the study of hardware and software systems.

4.2.2 Classification of software reliability model

Many ways of classifying software reliability models have been proposed. Software reliability models can be classified into the deterministic model and the probabilistic model. Performance measures of the deterministic model are obtained by analyzing the program texture and do not involve any random event. These deterministic models empirically measure the qualitative attributes of a software and are used in the early phase of the software life cycle to predict the number of errors in a program or are 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.

4.3 Software Reliability-Cost Function Development

The software reliability-related 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, as well as to the debugging time.

As indicated by Musa et al. [50], failure-identification personnel, failure-correction personnel, and computer time are the three key cost factors involved in debugging. By associating the resources of failure-identification personnel and computer time with Δt, and the resources of failure-identification personnel, failure-correction personnel, and computer time with $\Delta\mu$, we can formulate a software reliability-related cost function as follows:

$$RC(\lambda, \lambda^{\star}) = C_1 t_r \Delta t + C_2 \mu_r \Delta\mu \tag{4.1}$$

The formulations of the extra debugging time (Δt) and that of the extra faults removed ($\Delta\mu$) to reach $\lambda^{\star}$ from λ depend on the choice of the software reliability model. In this study, two important software reliability models have been employed to formulate the Δt and $\Delta\mu$.

First of all, the Jelinski-Moranda (JM) model [36] is used because this is one of the earliest and probably the most commonly used model for assessing software reliability. Next, because of its simplicity and applicability over a wide range of testing situations, the Goel-Okumoto nonhomogeneous Poisson process (NHPP) model [25] is used.

Time between failures models like JM model make following assumptions which are unrealistic.

1. The instantaneous failure rate of software is proportional to the number of errors remaining in it, each of which is equally likely to cause the next failure.

2. The time separations between failures are statistically independent and distributed exponentially with different failure rates.

However, fault count models, typically, Goel-Okumoto NHPP model, assumes that the failure process is a nonhomogeneous Poisson process. This model replaces assumptions 1 and 2 above with those corresponding to the structure of a Poisson process. The interfailure times are no longer independent, and the instantaneous failure rate between failures varies with time. Some software development teams have successfully used this model to predict the number of remaining faults.

4.3.1 Jelinski-Moranda model

The software hazard function, or the failure rate during t_i, the time between the (i-1)st and ith failures, is given by

$$Z(t_i) = \lambda_i = \Phi[N - (i - 1)] \tag{4.2}$$

Since this hazard function is constant, the number of faults discovered can be easily expressed in terms of the failure rate:

$$\begin{aligned} (i-1) &= N - \frac{\lambda_i}{\Phi} \\ i &= 1 + N - \frac{\lambda_i}{\Phi} \end{aligned} \tag{4.3}$$

Let λ_j be the projected failure rate, then

$$j = 1 + N - \frac{\lambda_j}{\Phi} \tag{4.4}$$

The extra number of faults needed to be removed to reach the projected failure rate from the current failure rate is

$$\begin{aligned} \Delta\mu &= j - i \\ &= \frac{1}{\Phi}(\lambda_i - \lambda_j) \qquad (4.5) \\ &= 1 + N - i - \frac{\lambda_j}{\Phi} \qquad (4.6) \end{aligned}$$

Either Eq. 4.5 or Eq. 4.6 can be used to get the number of faults removed to reach λ_j.

Also, the expected extra debugging time required can be expressed in terms of the failure rate. The total debugging time observed up to (i-1)st fault discovery is

$$\sum_{k=1}^{i-1} t_k = \sum_{k=1}^{i-1} \frac{1}{\lambda_k}$$

because the MTTF is the reciprocal proportion of failure rate. The total debugging time required to reach the projected failure rate λ_j is

$$\sum_{k=1}^{j-1} t_k = \sum_{k=1}^{j-1} \frac{1}{\lambda_k}$$

Therefore, the extra debugging time required to reach λ_j becomes

$$\begin{aligned}\Delta t &= \sum_{k=1}^{j-i} t_k - \sum_{k=1}^{i-1} t_k \\ &= \sum_{k=1}^{j-i} \frac{1}{\lambda_k} - \sum_{k=1}^{i-1} \frac{1}{\lambda_k} \\ &= \sum_{k=i}^{j-1} \frac{1}{\lambda_k} \qquad (4.7)\end{aligned}$$

Eq. 4.7 can be rewritten in terms of λ_j, so that Δt of a given λ_j can be directly evaluated. Since $j = 1 + N - \frac{\lambda_j}{\Phi}$,

$$\begin{aligned}\Delta t &= \sum_{k=i}^{j-1} \frac{1}{\Phi[N-(k-1)]} \\ &= \sum_{k=i}^{N-\frac{\lambda_j}{\Phi}} \frac{1}{\Phi[N-(k-1)]} \qquad (4.8)\end{aligned}$$

In some cases, the reliability objective is based on the reliability level of a given operation time. To formulate the reliability-related cost function of this type,

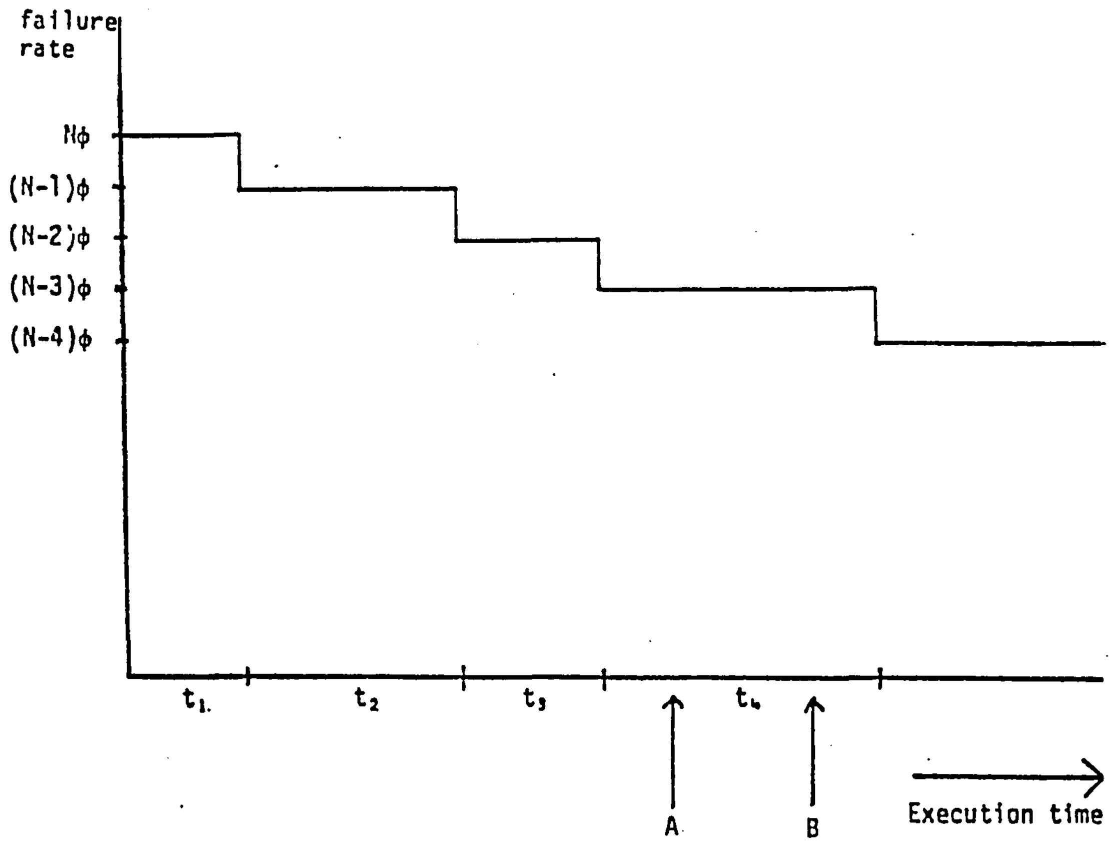


Figure 4.1: A typical plot of $Z(t_i)$ for the JM model

reliability can be represented as a function of debugging time plus operation time. The system reliability $r(s)$ after the ith failure occurs is

$$r_i(s) = e^{-\lambda_i \cdot s} \tag{4.9}$$

To represent the number of faults discovered in terms of the system reliability,

$$\begin{aligned} \ln r_i(s) &= -\lambda_i s \\ \lambda_i &= -\frac{\ln r_i(s)}{s} \end{aligned} \tag{4.10}$$

Substitute Eq. 4.10 into Eq. 4.3; then,

$$i = 1 + N + \frac{\ln r_i(s)}{\Phi S} \tag{4.11}$$

Let $r_j(s)$ be the system reliability desired; then,

$$j = 1 + N + \frac{\ln r_j(s)}{\Phi S} \tag{4.12}$$

Therefore, the extra number of faults removed to reach the system reliability required is

$$\Delta\mu = j - i = \frac{1}{\Phi S}[\ln r_j(s) - \ln r_i(s)] \tag{4.13}$$

Eq. 4.13 can be rewritten in terms of the parameters available and the system reliability required.

$$\Delta\mu = 1 + N - i + \frac{\ln r_j(s)}{\Phi S} \tag{4.14}$$

Also, the extra debugging time required to reach $r_j(s)$ after ith fault discovery is

$$\Delta t = \sum_{k=1}^{j-1} t_k - \sum_{k=1}^{i-1} t_k.$$

From Eq. 4.12, let M be j-1, then

$$M = j - 1 = N + \frac{\ln r_j(s)}{\Phi S} \tag{4.15}$$

Therefore,

$$\Delta t = \sum_{k=i}^{M} \frac{1}{\Phi[N - (k-1)]} \tag{4.16}$$

4.3.2 Goel-Okumoto nonhomogeneous Poisson process model

In this model Goel and Okumoto [25] assumed that a software system is subject to failures at random times caused by faults present in the system. The following form of the model was proposed

$$P[N(t) = y] = \frac{(m(t))^t}{y} e^{-m(t)}, y = 0, 1, 2, \ldots \tag{4.17}$$

When the Goel-Okumoto NHPP model is used, the expected number of faults removed after debugging time t is

$$m(t) = a[1 - e^{-bt}] \tag{4.18}$$

and the program failure rate at intermediate debugging time t is

$$\lambda(t) = m'(t) = abe^{-bt}. \tag{4.19}$$

Therefore, the debugging time t and the debugging time $t^\star$, to reach the projected failure rate $\lambda^\star$, can be represented in terms of failure rate. Since $\ln \lambda = \ln(ab) - bt$,

$$t = \frac{1}{b}[\ln(ab) - \ln \lambda]. \tag{4.20}$$

and

$$t^\star = \frac{1}{b}[\ln(ab) - \ln\lambda^\star]. \tag{4.21}$$

Additionally, the expected number of faults removed, $m(t^\star)$, to reach $\lambda^\star$ can be represented as

$$\begin{aligned} m(t^\star) &= a(1 - e^{-bt^\star}) \\ &= a[1 - e^{-(\ln(ab)-\ln\lambda^\star)}] \\ &= a(1 - \frac{\lambda^\star}{ab}) \\ &= a - \frac{\lambda^\star}{b}. \end{aligned} \tag{4.22}$$

Let the objective failure rate be $\lambda^\star$, the current time be t, and the current failure rate be λ, the extra debugging time required and the extra faults removed to reach $\lambda^\star$ from λ are

$$\begin{aligned} \Delta t &= t^\star - t \\ &= \frac{1}{b}(\ln\lambda - \ln\lambda^\star) \end{aligned} \tag{4.23}$$

$$\begin{aligned} \Delta m &= a(e^{-bt} - e^{-bt^\star}) \\ &= ae^{-\ln ab}\left[e^{\ln\lambda} - e^{\ln\lambda^\star}\right] \\ &= \frac{1}{b}(\lambda - \lambda^\star) \qquad\qquad \lambda \geq \lambda^\star. \end{aligned} \tag{4.24}$$

Let S_k be the time between failures (k-1)th and kth, and t_k be the time to k failures, then it can be shown that the conditional reliability function of S_k, given $t_{k-1} = t$, is

$$r_{S_k|t_{k-1}}(s|t) = exp\left[-a\left(e^{-bt} - e^{-b(t+s)}\right)\right]. \tag{4.25}$$

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

$$\begin{aligned}
e^{-bt} - e^{-b(t+s)} &= -\frac{\ln r(s|t)}{a} \\
e^{-bt} - e^{-bt} \cdot e^{-bs} &= -\frac{\ln r(s|t)}{a} \\
e^{-bt}\left(1 - e^{-bs}\right) &= -\frac{\ln r(s|t)}{a} \\
e^{-bt} &= -\frac{\ln r(s|t)}{a(1-e^{-bs})} \\
-bt &= \ln\left[\frac{-\ln r(s|t)}{a(1-e^{-bs})}\right] \\
t &= -\frac{1}{b}\ln\left[-\frac{\ln r(s|t)}{a(1-e^{-bs})}\right].
\end{aligned}$$

Similarly, m(t) can be represented in terms of $r(s|t)$. Hence,

$$\begin{aligned}
m(t) &= a(1-e^{-bt}) \\
&= a\left[1 + \frac{\ln r(s|t)}{a(1-e^{-bs})}\right] \\
&= a + \frac{\ln r(s|t)}{\left(1-e^{-bs}\right)}.
\end{aligned}$$

Therefore,

$$\begin{aligned}
\Delta t &= t^\star - t \\
&= \frac{1}{b}\left[\ln\left(-\frac{\ln r(s|t)}{a(1-e^{-bs})}\right) - \ln\left(-\frac{\ln r(s|t^\star)}{a(1-e^{-bs})}\right)\right] && (4.26) \\
\Delta m &= m(t^\star) - m(t) \\
&= \frac{\ln r(s|t^\star)}{(1-e^{-bs})} - \frac{\ln r(s|t)}{(1-e^{-bs})} \\
&= \frac{1}{(1-e^{-bs})}\left[\ln r(s|t^\star) - \ln r(s|t)\right]. && (4.27)
\end{aligned}$$

4.4 Software Redundancy Model

In software development, 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 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 [18,38]. Some input data will fail more than one redundancy because of the common errors made by different development teams. This partial independence 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. The common-cause model for software redundancy is developed as follows.

4.4.1 Two-component Markov model

Because of common-cause failure, a system with two partially independent software components in parallel can be transformed into a series system with two independent components in parallel and a common-cause component as shown in Figure 4.2. The Markov model of this system with common-cause failure is shown in Figure 4.3 where the failure rates of each independent component are assumed to be the same.

Let the state number of this Markov process be the number of components failed. Then, the differential equations of this Markov process are

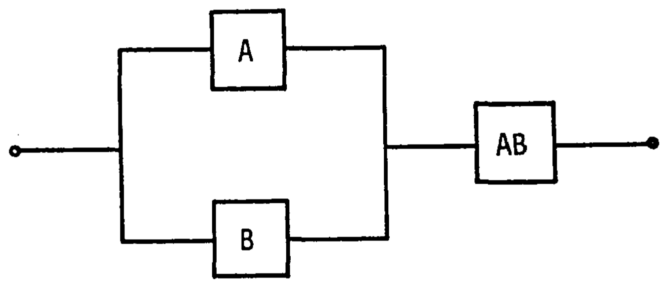


Figure 4.2: Transformed two-component software redundancy

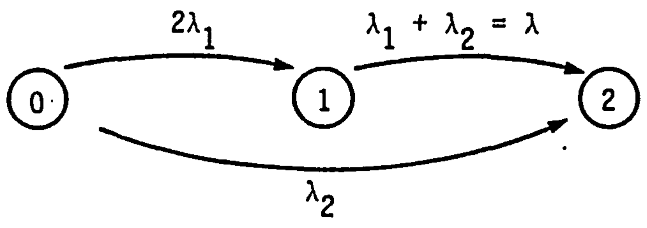


Figure 4.3: Two-component Markov model with common-cause failure

$$P_0'(t) = -(2\lambda_1 + \lambda_2)P_0(t)$$
$$P_1'(t) = 2\lambda_1 P_0(t) - \lambda P_1(t)$$
$$P_2'(t) = \lambda_2 P_0(t) + \lambda_1 P_1(t)$$
$$P_0(t) + P_1(t) + P_2(t) = 1$$

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

When the Laplace transform is taken,

$$\begin{aligned} sP_0(s) - 1 &= -(2\lambda_1 + \lambda_2)P_0(s) \\ P_0(s) &= 1/(s + 2\lambda_1 + \lambda_2) \end{aligned}$$

and

$$\begin{aligned} sP_1(s) &= 2\lambda_1 P_0(s) - (\lambda_1 + \lambda_2)P_1(s) \\ P_1(s) &= \frac{2\lambda_1 P_0(s)}{s + \lambda_1 + \lambda_2} \\ &= \frac{A_1}{s + \lambda_1 + \lambda_2} + \frac{A_2}{s + 2\lambda_1 + \lambda_2} \end{aligned}$$

where

$$\begin{aligned} A_1 &= 2\lambda_1/\lambda_1 \quad = 2 \\ A_2 &= 2\lambda_1/-\lambda_2 = -2. \end{aligned}$$

Taking the inverse Laplace transform, the state probabilities are

$$\begin{aligned} P_0(t) &= e^{-(2\lambda_1+\lambda_2)t} \\ P_1(t) &= 2e^{-(\lambda_1+\lambda_2)t} - 2e^{-(2\lambda_1+\lambda_2)t} \end{aligned}$$

The system reliability is

$$\begin{aligned} R_s(t) &= P_0(t) + P_1(t) \\ &= 2e^{-(\lambda_1+\lambda_2)t} - e^{-(2\lambda_1+\lambda_2)t}. \end{aligned} \tag{4.28}$$

By the use of matrix method, the same system reliability can be derived. The differential equations based on the Markov model can be expressed in terms of matrix.

$$\vec{P}'(t) = \mathbf{A}\vec{P}(t)$$

The transient matrix **A** for a two-component redundant system is

$$\mathbf{A} = \begin{bmatrix} -(2\lambda_1+\lambda_2) & 0 & 0 \\ 2\lambda_1 & -(\lambda_1+\lambda_2) & 0 \\ \lambda_2 & (\lambda_1+\lambda_2) & 0 \end{bmatrix}$$

with eigenvalues $E_0 = -(2\lambda_1 + \lambda_2)$, $E_1 = -(\lambda_1 + \lambda_2)$, and $E_2 = 0$.

For every complex $n \times n$ matrix **A** there exists a nonsingular matrix **P** such that the matrix

$$\mathbf{J} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$$

is in the canonical form

$$\mathbf{J} = \begin{bmatrix} J_0 & & & \\ & J_1 & 0 & \\ & 0 & \ddots & \\ & & & J_n \end{bmatrix},$$

where **J** is called Jordan canonical form and it is a diagonal matrix with diagonal element of matrix **A**, i.e.,

$$\mathbf{J} = \begin{bmatrix} E_0 & & & \\ & E_1 & & \\ & & \ddots & \\ & & & E_n \end{bmatrix}.$$

A set of corresponding eigenvectors for a two-component redundant system is

$$\mathbf{P} = (\vec{P_0}\vec{P_1}\vec{P_2})$$

The eigenvectors lead to three sets of linear equations associated with a set of three equations in three unknowns.

$$(\mathbf{E}_i\mathbf{I} - \mathbf{A})\vec{P_i} = 0 \qquad\qquad i = 0, 1, 2$$

where $\mathbf{E}_i$ is an eigenvalue and $\mathbf{I}$ is the identity matrix.

The values of the first vector $\vec{P_0}$ are the solution of the following simultaneous equations.

$$(\mathbf{E}_0\mathbf{I} - \mathbf{A})(\vec{P_0})$$

$$= \begin{bmatrix} 0 & 0 & 0 \\ -2\lambda_1 & -\lambda_1 & 0 \\ -\lambda_2 & -(\lambda_1 + \lambda_2) & -(2\lambda_1 + \lambda_2) \end{bmatrix} \begin{bmatrix} P_{00} \\ P_{01} \\ P_{02} \end{bmatrix} = 0$$

Therefore,

$$\vec{P_0} = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

For $\vec{P}_1$,

$$(\mathbf{E}_1\mathbf{I} - \mathbf{A})(\vec{P}_1)$$

$$= \begin{bmatrix} -\lambda_1 & 0 & 0 \\ -2\lambda_1 & 0 & 0 \\ -\lambda_2 & -(\lambda_1 + \lambda_2) & -(\lambda_1 + \lambda_2) \end{bmatrix} \begin{bmatrix} P_{10} \\ P_{11} \\ P_{12} \end{bmatrix} = 0$$

Therefore,

$$\vec{P}_1 = \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}$$

For $\vec{P}_2$,

$$(\mathbf{E}_2\mathbf{I} - \mathbf{A})(\vec{P}_2)$$

$$= \begin{bmatrix} (2\lambda_1 + \lambda_2) & 0 & 0 \\ -2\lambda_1 & +(\lambda_1 + \lambda_2) & 0 \\ -\lambda_2 & -(\lambda_1 + \lambda_2) & 0 \end{bmatrix} \begin{bmatrix} P_{20} \\ P_{21} \\ P_{22} \end{bmatrix} = 0$$

Therefore,

$$\vec{P}_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Hence, the matrix **P** becomes

$$\mathbf{P} = (\vec{P}_0\vec{P}_1\vec{P}_2) = \begin{bmatrix} 1 & 0 & 0 \\ -2 & -1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

The inverse matrix $\mathbf{P}^{-1}$ is the same as $\mathbf{P}$ and the matrix $\mathbf{J}$ becomes

$$\mathbf{J} = \begin{bmatrix} -(2\lambda_1 + \lambda_2) & 0 & 0 \\ 0 & -(\lambda_1 + \lambda_2) & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Using the initial conditions,

$$\vec{P}(0) = \begin{bmatrix} P_0(0) \\ P_1(0) \\ P_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \xi$$

and

$$\begin{bmatrix} P_0(t) \\ P_1(t) \\ P_2(t) \end{bmatrix} = \mathbf{P}e^{\mathbf{J}t}\mathbf{P}^{-1}\xi = \begin{bmatrix} 1 & 0 & 0 \\ -2 & -1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} e^{E_0 t} & 0 & 0 \\ 0 & e^{E_1 t} & 0 \\ 0 & 0 & e^{E_2 t} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ -2 & -1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Therefore,

$$P_2(t) = e^{-(2\lambda_1+\lambda_2)t} - 2e^{-(\lambda_1+\lambda_2)t} + 1.$$

The system reliability is

$$R_s(t) = 1 - P_2(t) = 2e^{-(\lambda_1+\lambda_2)t} - e^{-(2\lambda_1+\lambda_2)t}.$$

4.4.2 Three-component Markov model

A system with three partially independent software components (A, B, and C) in parallel is shown in Figure 4.4. Since some input will cause one, two, or three components to fail, the failure rate of each software component (e.g., component A) can be broken down into an independent failure rate (λ_A), two two-component

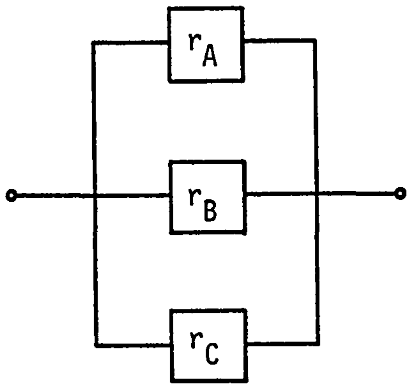


Figure 4.4: Three-component software redundancy

common-cause failure rates (λ_{AB} and λ_{AC}), and a three-component common-cause failure rate (λ_{ABC}). A Venn diagram (Figure 4.5) can be introduced to provide a better picture of these failure rate divisions.

Here, the system is operating as long as any entire circle among three circles is good. By assuming $\lambda_1 = \lambda_A = \lambda_B = \lambda_C$, $\lambda_2 = \lambda_{AB} = \lambda_{BC} = \lambda_{AC}$, and $\lambda_3 = \lambda_{ABC}$, a three-component Markov model with common-cause failure is shown in Figure 4.6. The differential equations and initial condition are as follows.

$$\begin{aligned}
P_0'(\mathrm{t}) &= -(3\lambda_1 + 3\lambda_2 + \lambda_3)P_0(t) \\
P_1'(\mathrm{t}) &= 3\lambda_1 P_0(t) - (2\lambda_1 + 3\lambda_2 + \lambda_3)P_1(t) \\
P_2'(\mathrm{t}) &= 3\lambda_2 P_0(t) + 2(\lambda_1 + \lambda_2)P_1(t) - (\lambda_1 + 2\lambda_2 + \lambda_3)P_2(t) \\
P_3'(\mathrm{t}) &= \lambda_3 P_0(t) + (\lambda_2 + \lambda_3)P_1(t) + (\lambda_1 + 2\lambda_2 + \lambda_3)P_2(t)
\end{aligned}$$

$$P_0(t) + P_1(t) + P_2(t) + P_3(t) = 0$$

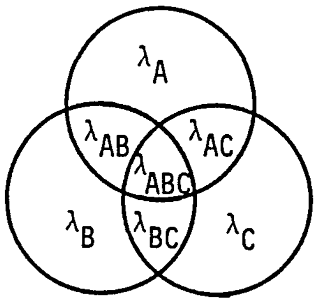


Figure 4.5: A Venn diagram of failure rate

and initial conditions $P_0(0) = 1, P_1(0) = P_2(0) = P_3(0) = 0$.

Taking the Laplace transform,

$$\begin{aligned} sP_0(s) - 1 &= -(3\lambda_1 + 3\lambda_2 + \lambda_3)P_0(s) \\ P_0(s) &= 1/(s + 3\lambda_1 + 3\lambda_2 + \lambda_3) \end{aligned}$$

and

$$\begin{aligned} sP_1(s) &= 3\lambda_1 P_0(s) - (2\lambda_1 + 3\lambda_2 + \lambda_3)P_1(s) \\ P_1(s) &= \frac{3\lambda_1 P_0(s)}{s + 2\lambda_1 + 3\lambda_2 + \lambda_3} \\ &= \frac{A_1}{s + 2\lambda_1 + 3\lambda_2 + \lambda_3} + \frac{A_2}{s + 3\lambda_1 + 3\lambda_2 + \lambda_3} \end{aligned}$$

where A_1=3, A_2=-3. And

$$sP_2(s) = (2\lambda_1 + 2\lambda_2)P_1(s) + 3\lambda_2 P_0(s) - (\lambda_1 + 2\lambda_2 + \lambda_3)P_2(s)$$

$$\begin{aligned}
P_2(s) &= \frac{(2\lambda_1 + 2\lambda_2)P_1(s) + 3\lambda_2 P_0(s)}{(s + \lambda_1 2\lambda_2 + \lambda_3)} \\
&= \frac{B_1}{s + \lambda_1 + 2\lambda_2 + \lambda_3} + \frac{B_2}{s + 2\lambda_1 + 3\lambda_2 + \lambda_3} + \frac{B_3}{s + 3\lambda_1 + 3\lambda_2 + \lambda_3}
\end{aligned}$$

where

$$\begin{aligned}
B_1 &= \frac{3(2\lambda_1^2 + \lambda_2^2 + 3\lambda_1\lambda_2)}{(\lambda_1 + \lambda_2)(2\lambda_1 + \lambda_2)} = 3 \\
B_2 &= \frac{6\lambda_1^2 + 6\lambda_1\lambda_2}{-(\lambda_1 + \lambda_2)(\lambda_1)} = -6 \\
B_3 &= \frac{6\lambda_1^2 + 3\lambda_1\lambda_2}{-(2\lambda_1 - \lambda_2)(-\lambda_1)} = 3.
\end{aligned}$$

Taking the inverse Laplace transform, the state probabilities are

$$\begin{aligned}
P_0(t) &= e^{-(3\lambda_1+3\lambda_2+\lambda_3)t} \\
P_1(t) &= 3e^{-(2\lambda_1+3\lambda_2+\lambda_3)t} - 3e^{-(3\lambda_1+3\lambda_2+\lambda_3)t} \\
P_2(t) &= 3e^{-(\lambda_1+2\lambda_2+\lambda_3)t} - 6e^{-(2\lambda_1+3\lambda_2+\lambda_3)t} + 3e^{-(3\lambda_1+3\lambda_2+\lambda_3)t}.
\end{aligned}$$

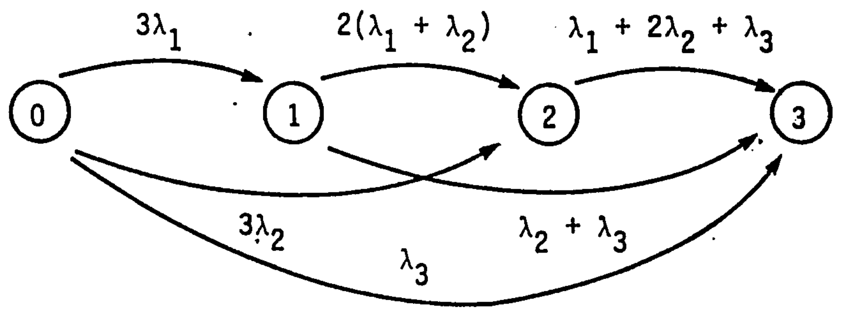
The system reliability is

$$\begin{aligned}
R_s(t) &= P_0(t) + P_1(t) + P_2(t) \\
&= 3e^{-(\lambda_1+2\lambda_2+\lambda_3)t} - 3e^{-(2\lambda_1+3\lambda_2+\lambda_3)t} && (4.29) \\
&= +e^{-(3\lambda_1+3\lambda_2+\lambda_3)t}. && (4.30)
\end{aligned}$$

4.4.3 Four-component Markov model

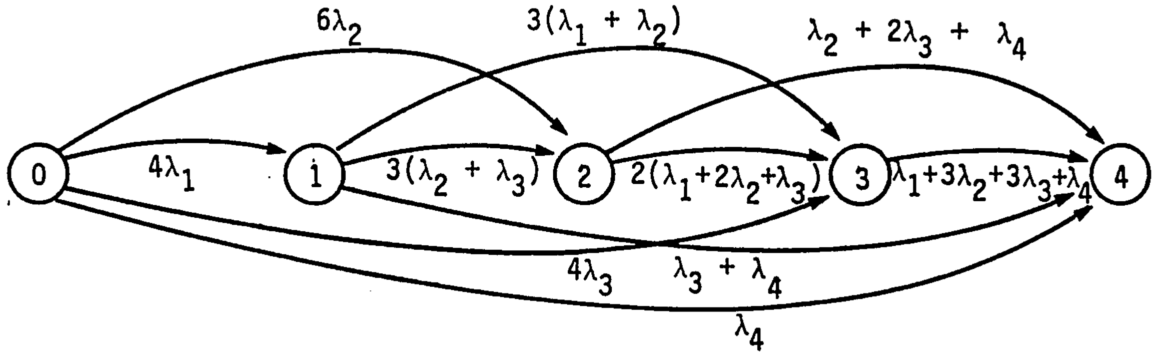
Based on the same argument, the four-component Markov model is shown in Figure 4.7 and the differential equations are as follows:

$$P_0'(t) = -(4\lambda_1 + 6\lambda_2 + 4\lambda_3 + \lambda_4)P_0(t)$$



$3\lambda_1$
$2(\lambda_1 + \lambda_2)$
$\lambda_1 + 2\lambda_2 + \lambda_3$
0
1
2
3
$3\lambda_2$
$\lambda_2 + \lambda_3$
λ_3

Figure 4.6: Three-component Markov model with common-cause failures



$6\lambda_2$
$3(\lambda_1 + \lambda_2)$
$\lambda_2 + 2\lambda_3 + \lambda_4$
0
$4\lambda_1$
1
$3(\lambda_2 + \lambda_3)$
2
$2(\lambda_1+2\lambda_2+\lambda_3)$
3
$\lambda_1+3\lambda_2+3\lambda_3+\lambda_4$
4
$4\lambda_3$
$\lambda_3 + \lambda_4$
λ_4

Figure 4.7: Four-component Markov model with common-cause failures

$$
\begin{aligned}
P_1'(t) &= 4\lambda_1 P_0(t) - (3\lambda_1 + 6\lambda_2 + 4\lambda_3 + \lambda_4)P_1(t) \\
P_2'(t) &= 6\lambda_2 P_0(t) + 3(\lambda_1 + \lambda_2)P_1(t) - (2\lambda_1 + 5\lambda_2 + 4\lambda_3 + \lambda_4)P_2(t) \\
P_3'(t) &= 4\lambda_3 P_0(t) + 3(\lambda_2 + \lambda_3)P_1(t) + (2\lambda_1 + 4\lambda_2 + 2\lambda_3)P_2(t) \\
&\quad -(\lambda_1 + 3\lambda_2 + 3\lambda_3 + \lambda_4)P_3(t) \\
P_4'(t) &= \lambda_4 P_0(t) + (\lambda_3\lambda_4)P_1(t) + (\lambda_2 + 2\lambda_3 + \lambda_4)P_2(t) \\
&\quad +(\lambda_1 + 3\lambda_2 + 3\lambda_3 + \lambda_4)P_3(t)
\end{aligned}
$$

with initial conditions $P_0(0)=1$, $P_1(0) = P_2(0) = P_3(0) = P_4(0) = 0$.

Taking the Laplace transform and then the inverse Laplace transform, the system reliability is

$$
\begin{aligned}
R_s &= 4e^{-(\lambda_1+3\lambda_2+3\lambda_3+\lambda_4)t} - 6e^{-(2\lambda_1+5\lambda_2+4\lambda_3+\lambda_4)t} \\
&\quad +4e^{-(3\lambda_1+6\lambda_2+4\lambda_3+\lambda_4)t} - e^{-(4\lambda_1+6\lambda_2+4\lambda_3+\lambda_4)t}. \qquad (4.31)
\end{aligned}
$$

4.4.4 N-component Markov model with common-cause

The model can be extended to a generic N-component model. Without making a significant discrepancy in system reliability, a simplified N-component Markov model can be considered and shown in Figure 4.8. In this simplified model, the only common-cause failures considered are the common-cause failures that cause all the redundancies to fail. This common-cause failure rate may represent the failure rate of system software. The system reliability of this simplified model can be derived from the preliminary analysis as follows:

The differential equations of this Markov process are

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

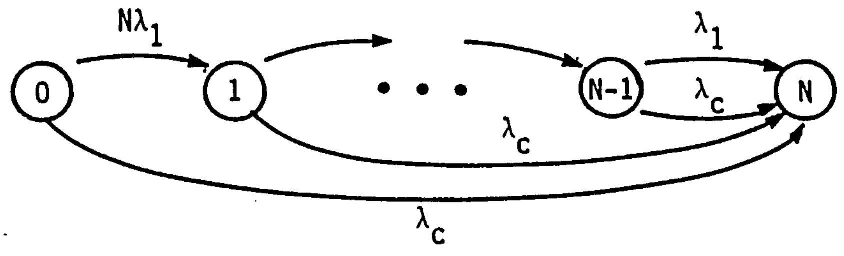


Figure 4.8: N-component Markov model with common-cause failure

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

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

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

$P_N(0) = 1$

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

$$\begin{aligned} sP_N(s) - 1 &= -(N\lambda + \lambda_c)P_N(s) \\ P_N(s) &= 1/(s + N\lambda + \lambda_c); \end{aligned}$$

then,

$$P_N(t) = e^{-(N\lambda+\lambda_c)t}. \qquad (4.32)$$

In addition,

$$N\lambda P_N(s) - [(N-1)\lambda + \lambda_c]P_{N-1}(s) = sP_{N-1}(s) \qquad (4.33)$$

$$\begin{aligned} P_{N-1}(s) &= \frac{N\lambda P_N(s)}{s + (N-1)\lambda + \lambda_c} \\ &= \frac{N}{s + (N-1)\lambda + \lambda_c} - \frac{N}{s + N\lambda + \lambda_c} \end{aligned}$$

then

$$P_{N-1}(t) = Ne^{-[(N-1)\lambda+\lambda_c]t} - Ne^{-(N\lambda+\lambda_c)t}. \qquad (4.34)$$

In general, the state probabilities and the system reliability can be derived as follows.

$$P_k(t) = \sum_{j=k}^{N} \left(\frac{N!}{k!}[\prod_{p=k,p\neq j}^{N} (p-j)]^{-1} e^{-(j\lambda+\lambda_c)t} \right) \qquad (4.35)$$

$$R_s(t) = \sum_{k=1}^{N-1} P_k(t). \tag{4.36}$$

The above solutions of state probabilities are in exact but complex forms. An approximated form of Equation 4.36 along with the accuracy of the solution under the simplification assumption in a specified environment, is derived as follows.

$$P_0'(t) = \lambda_c[1 - P_0(t)] + \lambda P_1(t). \tag{4.37}$$

If

$$\lambda P_1(t) \ll \lambda_c[1 - P_0(t)],$$

neglecting $\lambda P_1(t)$,

$$\begin{aligned} P_0'(t) &= \lambda_c[1 - P_0(t)] \\ sP_0(s) &= \lambda_c/s - \lambda_c P_0(s) \\ P_0(s) &= 1/s - 1/(s + \lambda_c) \end{aligned}$$

and

$$P_0(t) = 1 - e^{-\lambda_c t}.$$

The approximated system reliability is

$$R_s(t) \simeq 1 - P_0(t) = e^{-\lambda_c t}. \tag{4.38}$$

Because of expensive development cost, in reality, it would be very undesirable to have a system of more than four software redundancies in parallel. When other common causes exist among a small subset of N modules, we have to revise the above simplified model. A refinement for the full-version, unsimplified, N-component common cause model is evaluated in next subsection.

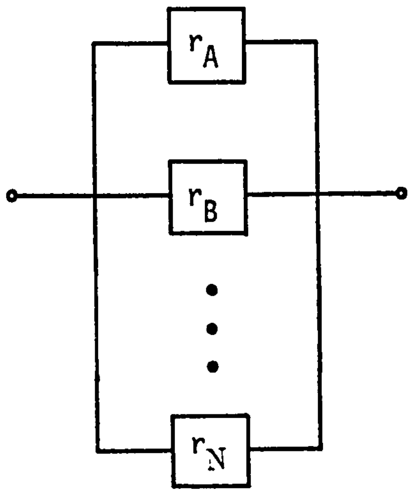


Figure 4.9: Configuration of N-component redundant system

4.4.5 Generic N-component Markov model

A system with N partially independent software components in parallel is shown in Figure 4.9. The unsimplified Markov model of this system with common-cause failure is shown in Figure 4.10 where the failure rates of each independent component are assumed to be the same. Let the state number of this Markov process be the number of components failed. Then, the differential equations of this Markov process can be expressed in terms of matrix.

$$\vec{P}'(t) = \mathbf{A}\vec{P}(t)$$

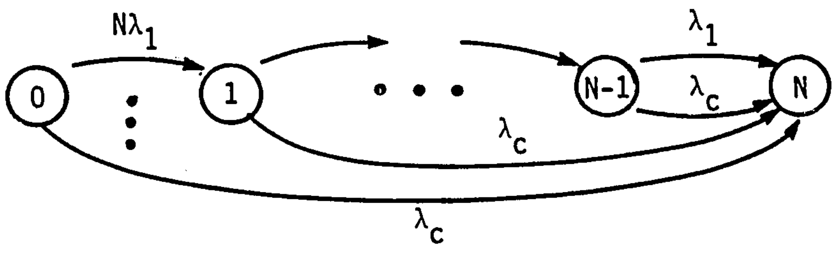


Figure 4.10: Generic N-component Markov model

The transient matrix **A** is

$$\mathbf{A} = \begin{bmatrix} a_{00} & 0 & 0 & \cdots & 0 & 0 \\ a_{01} & a_{11} & 0 & \cdots & 0 & 0 \\ a_{02} & a_{12} & a_{22} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ a_{0(n-1)} & a_{1(n-1)} & a_{2(n-1)} & \cdots & a_{(n-1)(n-1)} & a_{n(n-1)} \\ a_{0n} & a_{1n} & a_{2n} & \cdots & a_{(n-1)n} & a_{nn} \end{bmatrix}$$
$$= (a_{ij})$$

4.4.5.1 Construction of matrix A An element a_{ij} of matrix **A** represents the rate that the system moves from state i to state j. Let z be the system's jump size (z = j - i).

<u>Jump size (z) = 1</u>

First of all, the element a_{01} represents the rate that the system moves from state zero to state one. The system can move from state zero to state one only when a component out of n good components is failed due to an independent cause failure. Since all independent cause failures are mutually exclusive, the number of possible outcomes of this event is the same as that of choosing one component from n good components. In other words, the number of possible outcomes is that of choosing one out of n good components and none out of zero failed component.

$$n\lambda_1 = \binom{n}{1}\binom{0}{0}\lambda_1$$

Therefore, the element a_{01} becomes

$$\binom{n}{1}\sum_{k=0}^{0}\binom{0}{k}\lambda_1 \tag{4.39}$$

Second, the element a_{12} in matrix $\mathbf{A}$ represents the rate that system moves from state one to state two. The system can move from state one to state two when a component out of (n-1) good components is failed due to the introduction of one of these two cases to the system.

- λ_1 case — One of (n-1) remained independent cause failures. The number of possible outcomes of this case consists of choosing one out of (n-1) good components and none out of one failed component.

$$(n-1)\lambda_1 = \binom{n-1}{1}\binom{1}{0}\lambda_1$$

- λ_2 case — One of (n-1) two-component common-cause failures. Here, the number of possible outcomes for two-component common-cause failure including the failed component is that of choosing one component out of (n-1) good components and one out of one failed component.

$$(n-1)\lambda_2 = \begin{pmatrix} n-1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \lambda_2$$

The element a_{12} is derived by adding the above two cases together.

$$(n-1)(\lambda_1 + \lambda_2) = \begin{pmatrix} n-1 \\ 1 \end{pmatrix} \sum_{k=0}^{1} \begin{pmatrix} 1 \\ k \end{pmatrix} \lambda_{k+1} \tag{4.40}$$

Third, the element a_{23} means the system moves from state two to state three. Since the system is in state two, there are two failed components and (n-2) good components. If one more component out of (n-2) good components fails, then, the system moves to state three. Three different types of failure rate (λ_1, λ_2, and λ_3) are involved in this case.

- λ_1 case — A failure out of (n-2) remained independent failure. The number of possible outcomes consists of choosing one out of (n-2) good components and none out of two failed components.

$$(n-2)\lambda_1 = \begin{pmatrix} n-2 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \end{pmatrix} \lambda_1$$

- λ_2 case — A failure out of two (n-2) two-component common-cause failures. This is because there are (n-2) possible combinations of two-component common-cause failure including one bad component. Since there are two bad components in the system, the total number of combination is 2(n-2). This number consists of choosing one component out of (n-2) good components and another out of two failed components.

$$2(n-2)\lambda_2 = \begin{pmatrix} n-2 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} \lambda_2$$

- λ_3 case — A failure out of (n-2) three-component common-cause failures. There are (n-2) possible outcomes of three-component common-cause failure that include two bad components. This is because one component from (n-2) good components and other two from two failed components.

$$(n-2)\lambda_3 = \begin{pmatrix} n-2 \\ 1 \end{pmatrix} \begin{pmatrix} 2 \\ 2 \end{pmatrix} \lambda_3$$

Therefore, the element a_{23} becomes

$$(n-2)(\lambda_1 + 2\lambda_2 + \lambda_3) = \begin{pmatrix} n-2 \\ 1 \end{pmatrix} \sum_{k=0}^{2} \begin{pmatrix} 2 \\ k \end{pmatrix} \lambda_{k+2} \tag{4.41}$$

At last, the element $a_{(n-1)n}$ means the system moves from state (n-1) to state n. Since the system is in state (n-1), there are (n-1) failed components and one good component left. N different types of failure rates (λ_1, λ_2, ..., λ_n) are involved in this element.

- λ_1 case — A failure out of one good component. There is only one good component left. The number of possible outcomes consists of choosing one component out of one good component and none out of (n-1) failed components.

$$1\lambda_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} n-1 \\ 0 \end{pmatrix} \lambda_1$$

- λ_2 case — A failure out of (n-1) two-component common-cause failure. Each two-component failure includes one remained good component and one of (n-1) failed components. The number of possible outcomes consists of choosing one component from one good component and one component from (n-1) failed components.

$$(n-1)\lambda_2 = \binom{1}{1}\binom{n-1}{1}\lambda_2$$

- λ_3 case — A failure out of (n-1)(n-2)/2 three-component common-cause failure. There are (n-1)(n-2)/2 three-component common-cause failures. Each of them includes one remained good component and any two out of (n-1) failed components. The number of combination becomes simply choosing one component out of one good component and two components out of (n-1) failed components.

$$\frac{(n-1)(n-2)}{2}\lambda_3 = \binom{1}{1}\binom{n-1}{2}\lambda_3$$

⋮

- λ_{n-1} case — A component can also fail when the system is introduced an (n-1)-component common-cause failure that includes a last good component. Each (n-1)-component failure has a good component and (n-2) failed components. The number of outcomes of (n-1)-component common-cause failure is the same as the number of choosing one component from one good component and (n-2) failed components from (n-1) failed components.

$$\binom{1}{1}\binom{n-1}{n-2}\lambda_{n-1}$$

- λ_n case — Since the system is in state (n-1), the only possible outcome of n-component common-cause failure is the one that includes all (n-1) failed components and a left good component. The number of this outcome consists of choosing one out of one good component and (n-1) out of (n-1) failed components.

$$1\lambda_n = \binom{1}{1}\binom{n-1}{n-1}\lambda_n$$

The element $a_{(n-1)n}$ is derived by adding all cases together.

$$\binom{n-1}{0}\lambda_1+\binom{n-1}{1}\lambda_2+\binom{n-1}{2}\lambda_3+\ldots+\binom{n-1}{n-2}\lambda_{n-1}+\binom{n-1}{n-1}\lambda_n$$

$$= \binom{1}{1}\sum_{k=0}^{n-1}\binom{n-1}{k}\lambda_{k+1} \qquad (4.42)$$

In general, when the system jumps only one step (z = j - i = 1), the element a_{ij} of matrix **A** can be represented as follows:

$$a_{ij} = \binom{n-i}{z}\sum_{k=0}^{j-z}\binom{j-z}{k}\lambda_{k+z}, \qquad z = 1 \qquad (4.43)$$

<u>Jump size (z) = 2</u>

The element a_{02}, first, represents the rate that the system moves from state zero to state two. This can happen when a two-component common-cause failure fails two components in the system. There are n(n-1)/2 possible two-component common-cause failure combinations because any two components out of n good components are the candidates of this combination. The number of possible outcomes becomes the number of choosing two components out of n good components and none out of zero failed component.

$$\frac{n(n-1)}{2}\lambda_2 = \begin{pmatrix} n \\ 2 \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix}\lambda_2$$

Therefore, the element a_{02} becomes

$$\begin{pmatrix} n \\ 2 \end{pmatrix}\lambda_2 = \begin{pmatrix} n \\ 2 \end{pmatrix}\sum_{k=0}^{0}\begin{pmatrix} 0 \\ k \end{pmatrix}\lambda_{k+2} \tag{4.44}$$

Second, for the element a_{13}, the system moves from state one to state three when two components out of (n-1) good components are failed. Since a component has been failed and (n-1) good components are available, there are two cases to be considered.

- λ_2 case — Any two-component common-cause failure that do not consist the component already failed is a good candidate. The number of this outcomes consists of choosing two good components from (n-1) good components and none from one failed component.

$$(n-1)(n-2)/2\lambda_2 = \begin{pmatrix} n-1 \\ 2 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}\lambda_2$$

- λ_3 case — Any three-component common-cause failure that includes the component already failed is a good candidate. The number of this outcomes consists of choosing two good components from (n-1) good components and one component from one failed component.

$$(n-1)(n-2)/2\lambda_3 = \binom{1}{1}\binom{n-1}{2}\lambda_3$$

Therefore, the element a_{13} becomes the sum of the above two cases.

$$\binom{n-1}{2}\lambda_2 + \binom{n-1}{2}\lambda_3 = \binom{n-1}{2}\sum_{k=0}^{1}\binom{1}{k}\lambda_{2+k} \tag{4.45}$$

Third, the element a_{24} represents the rate that the system moves from state two to state three. This could happen if any of these three kinds of failure introduced to the system.

- λ_2 case — A failure out of (n-2)(n-3)/2 two-component common-cause failures. Since two components are already failed, the possible outcomes for two-component common-cause failure are any combination of two components from (n-2) good components and none from two failed components.

$$\frac{(n-2)(n-3)}{2}\lambda_2 = \binom{n-2}{2}\binom{2}{0}\lambda_2$$

- λ_3 case — A failure out of (n-2)(n-3) three-component common-cause failures. The possible candidates for three-component common-cause failure are any combination of two components from (n-2) good components and one of failed component. Since there are two components failed, the total number of candidates for three-component common-cause failure is two (n-2)(n-3)/2.

$$(n-2)(n-3)\lambda_3 = \binom{n-2}{2}\binom{2}{1}\lambda_3$$

- λ_4 case — A failure out of (n-2)(n-3)/2 four-component common-cause failures. The possible candidates for four-component common-cause failure are any combination of two components from (n-2) good components and two from two failed components.

$$\frac{(n-2)(n-3)}{2}\lambda_4 = \binom{n-2}{2}\binom{2}{2}\lambda_4$$

The element a_{24} is derived by adding three cases together.

$$\binom{n-2}{2}\lambda_2 + \binom{n-2}{2}2\lambda_3 + \binom{n-2}{2}\lambda_4 = \binom{n-2}{2}\sum_{k=0}^{2}\binom{2}{k}\lambda_{k+2} \tag{4.46}$$

In the same token, the elements a_{35}, a_{46}, ..., $a_{(n-3)(n-1)}$ can be derived. Finally, for the element $a_{(n-2)n}$, the types of failure rate involved in this element are from λ_2 to λ_n. Since there are already (n-2) failed components in the system, the system moves to state n if the remaining two good components fail. Therefore, every common-cause failure candidates should include these two good components.

- λ_2 case — The number of possible outcomes consists of choosing two components from two good components and none from (n-2) failed components.

$$\binom{2}{2}\binom{n-2}{0}\lambda_2$$

- λ_3 case — The number of possible outcomes consists of choosing two components from two good components and one from (n-2) failed components.

$$\binom{2}{2}\binom{n-2}{1}\lambda_2$$

$\vdots$

- λ_{n-1} case — The number of possible outcomes consists of choosing two components from two good components and (n-3) from (n-2) failed components.

$$\binom{2}{2}\binom{n-2}{n-3}\lambda_2$$

- λ_n case — The number of possible outcomes consists of choosing two components from two good components and (n-2) from (n-2) failed components.

$$\binom{2}{2}\binom{n-2}{n-2}\lambda_2$$

The sum of these cases becomes

$$\binom{n-2}{0}\lambda_2+\binom{n-2}{1}\lambda_3+\ldots+\binom{n-2}{n-3}\lambda_{n-1}+\binom{n-2}{n-2}\lambda_n$$

$$=\binom{n-(n-2)}{2}\sum_{k}^{n-2}\binom{n-2}{k}\lambda_{k+2} \qquad (4.47)$$

In general, when the system jumps two step (z = 2), the element a_{ij} of matrix **A** can be expressed as follows:

$$a_{ij} = \binom{n-i}{z} \sum_{k=0}^{j-z} \binom{j-z}{k} \lambda_{k+z}, \qquad z = 2 \qquad (4.48)$$

Jump size (z) = 3

The same procedure is applied to derive each element of matrix **A** when jump size becomes three. For the element a_{03}, the number of outcomes for three-component common-cause failure (λ_3) consists of choosing three components from n good components and none from zero failed component.

$$\binom{n}{3}\binom{0}{0}\lambda_3$$

For the element a_{14}, the number of outcomes for three-component common-cause failure (λ_3) and four-component common-cause failure (λ_4) consists of choosing three components from (n-1) good components. Here, outcome of four-component common-cause failure includes a failed component.

$$\binom{n-1}{3}\binom{1}{0}\lambda_3, \qquad \binom{n-1}{3}\binom{1}{1}\lambda_4$$

For the element a_{25}, there are three types of common-cause failure involved. The number of outcomes for three-component common-cause failure (λ_3) consists of choosing three components from (n-2) good components and none from two failed components. The number of outcomes for four-component common-cause failure (λ_4) consists of choosing three components from (n-2) good components and one component from two failed components. The number of outcomes for five-component common-cause failure (λ_5) consists of choosing three components from (n-2) good components and choosing two components from two failed components.

$$\binom{n-2}{3}\binom{2}{0}\lambda_3, \quad \binom{n-2}{3}\binom{2}{1}\lambda_4, \quad \binom{n-2}{3}\binom{2}{2}\lambda_5$$

In general, when the system jumps three steps (z = 3), the element a_{ij} of matrix **A** can be expressed as follows:

$$a_{ij} = \binom{n-i}{z}\sum_{k=0}^{j-z}\binom{j-z}{k}\lambda_{k+z}, \qquad z = 3 \qquad (4.49)$$

<u>Jump size (z) = n-1</u>

There are two elements belong to this jump size. For the element $a_{0(n-1)}$, the number of outcomes for (n-1)-component common-cause failure (λ_{n-1}) consists of choosing (n-1) components from n good components and none from zero failed component.

$$n\lambda_{n-1} = \binom{n}{n-1}\binom{0}{0}\lambda_{n-1}$$

For the element a_{1n}, the number of outcomes for (n-1)-component common-cause failure (λ_{n-1}) consists of choosing (n-1) components from (n-1) good components. The number of outcomes for n-component common-cause failure (λ_n) consists of choosing (n-1) components from (n-1) good components and one component from one failed component.

$$\binom{n-1}{n-1}(\lambda_{n-1} + \lambda_n) = \binom{n-1}{n-1}\sum_{k=0}^{1}\binom{1}{k}\lambda_{k+n-1}$$

In general, when the system jumps (n-1) steps, the element a_{ij} of matrix **A** can be expressed as follows:

$$a_{ij} = \begin{pmatrix} n-i \\ z \end{pmatrix} \sum_{k=0}^{j-z} \begin{pmatrix} j-z \\ k \end{pmatrix} \lambda_{k+z}, \qquad z = n-1 \tag{4.50}$$

<u>Jump size (z) = n</u>

For the element a_{0n}, the system moves from state zero to state n only when n-component common-cause failure is introduced. The number of outcomes for n-component common-cause failure (λ_n) consists of choosing n components from n good components.

$$1\lambda_n = \begin{pmatrix} n \\ n \end{pmatrix} \sum_{k=0}^{0} \begin{pmatrix} 0 \\ k \end{pmatrix} \lambda_{k+n} \tag{4.51}$$

<u>Jump size (z) = 0</u>

All diagonal elements of matrix **A** belong to this jump size. According to the property of transient matrix, the sum of each element in the same column should be zero. Therefore, the rate of these element is the negative of the sum of corresponding column vector elements.

$$a_{ii} = -\sum_{k=1}^{n-i} a_{i(k+i)}, \qquad i = 0, 1, \ldots, n \tag{4.52}$$

<u>Jump size = negative</u>

Since the system is nonrepairable, there is no backward flow in the Markov process. All elements in the upper side of matrix **A** are zero.

Hence, the general equations for the elements of matrix **A** are:

$$a_{ij} = \begin{cases} \begin{pmatrix} n-i \\ z \end{pmatrix} \Sigma_{k=0}^{j-z} \begin{pmatrix} j-z \\ k \end{pmatrix} \lambda_{k+z} & \text{if } i < j \\ -\Sigma_{k=1}^{n-i} \left[\begin{pmatrix} n-i \\ z \end{pmatrix} \Sigma_{k=0}^{j-z} \begin{pmatrix} j-z \\ k \end{pmatrix} \lambda_{k+z} \right] & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{4.53}$$

4.4.5.2 Numerical solution for system reliability In this study the system reliability of n components will be found by the use of numerical analysis. In order to find the numerical solution of the system reliability the numerical value of each component in matrix **A** should be calculated. In the middle of testing phase, the component failure rate can be estimated. Because the redundant components are not yet developed, the common-cause failure rates are unknown. Then, a careful estimation of parameters α, β based on historical data gives estimated common-cause failure rates. The relationship between component failure rate and common-cause failure rate are as follows:

$$\lambda_1 = \alpha\lambda, \qquad 0 \leq \alpha \leq 1$$

$$\lambda_2 = \beta\lambda_3 = \ldots = \beta^{n-2}\lambda_n$$

and

$$(1-\alpha)\lambda = \lambda_2 + \lambda_3 + \ldots + \lambda_n.$$

Eigenvalues are the diagonal elements of matrix **A**.

$$E_0 = a_{00}, \; E_1 = a_{11}, \; \ldots, \; E_n = a_{nn} = 0$$

For every complex $n \times n$ matrix $\mathbf{A}$ there exists a nonsingular matrix $\mathbf{P}$ such that the matrix

$$\mathbf{J} = \mathbf{P}^{-1}\mathbf{A}\mathbf{P}$$

is in the canonical form

$$\mathbf{J} = \begin{bmatrix} J_0 & & & \\ & J_1 & 0 & \\ & 0 & \ddots & \\ & & & J_n \end{bmatrix},$$

where $\mathbf{J}$ is called Jordan canonical form and it is a diagonal matrix with diagonal element of matrix $\mathbf{A}$, i.e.,

$$\mathbf{J} = \begin{bmatrix} E_0 & & & \\ & E_1 & & \\ & & \ddots & \\ & & & E_n \end{bmatrix}.$$

A set of corresponding eigenvectors is

$$\mathbf{P} = (\vec{P_0}\vec{P_1}\ldots\vec{P_n}) \qquad (4.54)$$

The eigenvectors lead to (n+1) sets of linear equations associated with a set of (n+1) equations in (n+1) unknowns.

$$(\mathbf{E}_i\mathbf{I} - \mathbf{A})\vec{P_i} = 0 \qquad\qquad i = 0, 1, \ldots, n$$

where $\mathbf{E}_i$ is an eigenvalue and $\mathbf{I}$ is the identity matrix.

The values of each $\vec{P_i}$ are the solution of Equation 4.54. Each vector $\vec{P_i}$ has (n+1) elements, $\vec{P_i} = [P_{i0}, P_{i1}, \ldots, P_{in}]^T$, and leads (n+1) simultaneous equations.

Since all elements of upper side of matrix **A** equals to zero, all elements before i^{th} element of vector $\vec{P_i}$ become to zero.

$$P_{ik} = 0, \qquad\qquad if \;\; k < i$$

Here, P_{ik} is the k^{th} element of vector $\vec{P_i}$.

The first element that is not zero is the i^{th} element of vector $\vec{P_i}$. All elements P_{ik} $(k > i)$ of vector $\vec{P_i}$ can be represented a function of i^{th} element (P_{ii}). Once, the value of P_{ii} is arbitrary chosen, the value of rest P_{ik} $(k > i)$ can be determined. By the inspection of P_{ii} values from simple cases, it is found that $P_{ii} = (-1)^{n-i}$ gives a simple matrix whose inverse matrix is the same. Therefore, let P_{ii} be $(-1)^{n-i}$, then, the value of rest elements of vector $\vec{P_i}$ can be calculated one by one.

After the completion of **P** matrix, the inverse $\mathbf{P}^{-1}$ matrix needs to be found. With the initial conditions,

$$\xi = [P_0(0), P_1(0), \ldots, P_n(0)]^T = [1, 0, \ldots, 0]^T$$

The probability of staying in each state can be calculated based on following equation.

$$\begin{bmatrix} P_0(t) \\ P_1(t) \\ \vdots \\ P_n(t) \end{bmatrix} = \mathbf{P}e^{\mathbf{J}t}\mathbf{P}^{-1}\xi$$

Finally, the system reliability is

$$R_s(t) = 1 - P_n(t).$$

4.5 Formulation of The Software Reliability Optimization

To optimize the reliability of a software system, the reliability redundancy allocation approach is applied. A general formulation of this problem is

$$Max\ R_S(\overline{X}, \overline{R})$$

subject to

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

4.5.1 A pure software system

A software is always accompanied with hardwares. However, when the reliability of hardware component in the system is known, the system reliability can be optimized by including only software components. When only software components are involved in the optimization problem, the above problem can be transformed into the following form:

$$Max\ R_S(R_1, ..., R_N)$$

subject to

$$\sum_{j \in S} f_{ij}(r_j, r_j^{\star}) \cdot h_{ij}(x_j) \leq b_i \qquad \text{for all i}$$

The objective function of the above formulation is represented in terms of the stage reliabilities that are, in turn, functions of both independent module reliabilities and of the number of redundant modules. The constraint function of the above formulation is reliability-related cost function. The reliability-related cost function is

$$f(r_j, r_j^\star) = C_1 t_r \Delta t + C_2 \mu_r \Delta\mu$$

where, the formations of the extra debugging time (Δt) and that of the extra faults removed (Δμ) to reach $r^\star$ from r depend on the choice of the software reliability model.

When the JM model is used, the formulations of the extra debugging time (Δt) and that of the extra faults removed (Δμ) becomes;

$$\Delta t = \sum_{k=i}^{M} \frac{1}{\Phi[N-(k-1)]}$$

$$\Delta\mu = 1 + N - i + \frac{\ln r^\star(s)}{\Phi S}$$

where

$$M = N + \frac{\ln r^\star(s)}{\Phi S}$$

When the NHPP model is used, the formulations of the extra debugging time (Δt) and that of the extra faults removed (Δμ) becomes;

$$\Delta t = t^\star - t = \frac{1}{\Phi}[\ln(-\ln r_j) - \ln(-\ln r_j^\star)]$$

and

$$\Delta\mu = \mu(t^\star) - \mu(t) = \frac{1}{\Phi s}[\ln r_j^\star - \ln r_j]$$

.

The redundancy-cost function, $h_{ij}(x_j)$, depends upon the type of constraints involved. A constant function, an increasing function, or a decreasing function can be used as needed and should be described in a generic form but in a form that reflects the software development life-cycle.

4.5.2 A hardware and software mixed system

When both hardware and software components are not trivial, the reliability of both hardware and software components should be optimized with optimal number of redundancy. In a pure software system, each stage represents an independent functional module or subsystem. However, this model can be extended to optimize the reliability of a hardware and software system by adding the constraint function of hardware part.

The objective function can be

$$Max \quad R_s(R_1, ..., R_N).$$

The new constraints become

$$\Sigma_{j \in H} f_{ij}(r_j) \cdot h_{ij}(X_j) \ + \ \Sigma_{j \in S} f_{ij}(r_j, r_j^{\star}) \cdot h_{ij}(x_j) \leq b_i$$

The objective function of the above formulation is represented in terms of the stage reliabilities. Each stage can be a pure software component, a pure hardware component, or a hardware and software mixed component. The constraint function is represented as the product of a reliability-related cost function and a redundancy-cost function. For hardware components, the reliability-related cost function is

$$r_j = exp[-\lambda_j S]$$
$$f(r_j) = v_j \left[\frac{-S}{\ln r_j} \right]^{u_j}.$$

For software components, the reliability-related cost function is

$$f(r_j, r_j^{\star}) = C_1 t_r \Delta t + C_2 \mu_r \Delta \mu$$

where, the formations of the extra debugging time (Δt) and that of the extra faults removed ($\Delta\mu$) to reach $r^\star$ from r again depend on the choice of the software reliability model.

When the JM model is used, the formulations of the extra debugging time (Δt) and that of the extra faults removed ($\Delta\mu$) becomes;

$$\Delta t = \sum_{k=i}^{M} \frac{1}{\Phi[N-(k-1)]}$$

$$\Delta\mu = 1 + N - i + \frac{\ln r^\star(s)}{\Phi S}$$

where

$$M = N + \frac{\ln r^\star(s)}{\Phi S}$$

When the NHPP model is used, the formulations of the extra debugging time (Δt) and that of the extra faults removed ($\Delta\mu$) becomes;

$$\Delta t = t^\star - t = \frac{1}{\Phi}[\ln(-\ln r_j) - \ln(-\ln r_j^\star)]$$

and

$$\Delta\mu = \mu(t^\star) - \mu(t) = \frac{1}{\Phi s}[\ln r_j^\star - \ln r_j]$$

The redundancy-cost function, $h_{ij}(x_j)$, depends upon the type of constraints involved. A constant function, an increasing function, or a decreasing function can be used as needed and should be described in a generic form but in a form that reflects the software development life-cycle.

4.5.3 The type of resources

The types of resources are:

- manpower
- computer usage
- project duration required (development time)
- reliability
- memory size
- others

The reliability cost functions related with those resources can be formulated as follows:

1. Manpower and computer time (total cost)

$$f_{ij} = RC(r, r^{\star}) = C_1 t_r \Delta t + C_2 \mu_r \Delta \mu \leq b_1$$

2. Project duration required

$$f_{ij} = t_r \Delta t \leq b_4$$

   where $t_r\Delta$t is the failure-identification time. It is assumed that the failure-correction time is small enough to be ignored.

3. Memory space

$$f_{ij} = M \leq b_5$$

4.5.4 Other problem formulation

When the development cost is the major concern and the performance requirement is the system minimum reliability, a general formulation of this problem is

$$\min \sum_{j \in s} f_{ij}(r_j, r_j^\star) \cdot h_{ij}(x_j) \tag{4.55}$$

where $r_j^\star$ is the projected component reliability. The main constraint could be $R_S(\overline{X}, \overline{R}) \geq R_{S,req}$. The decision variables that need to be calculated are $\overline{X}$ and $\overline{R}$.

4.5.5 Redundancy cost function

$$h_{ij}(x_j) = k \cdot x_j \tag{4.56}$$

The cost of increasing a redundant component is usually less than 1.5 times of the original unit development cost; this is because redundancy components shared

- the cost of specification,
- some of the design cost,
- most of testing cost, and
- most of the documentation cost

together with the original component.

4.6 Optimal Reliability and Redundancy Allocation Algorithm

In most reliability optimization problems, the decision variables are the number of redundancies that are integers (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. 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 [46]. As the problem gets complicated, however, the rounding off and trial-and-error procedure become inefficient and inaccurate. In addition, this approach provides no theoretical reasoning and has difficulties in extending the integer programming problem to the mixed-integer programming problem. Such an extension is frequently needed for reliability optimization. Furthermore, computation on the trial-and-error basis cannot be efficiently automated.

A method combining the Lagrange multiplier technique with the branch-and-bound technique is proposed by Kuo et al. [40]. 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 allocation problem. When dealing with the latter problem, only branching and bounding the integer variable is necessary.

4.6.1 Lagrange multiplier and Kuhn-Tucker conditions

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

$$Max\ L(\overline{X},\overline{R},\overline{\Lambda}) = R_S(\overline{X},\overline{R}) - \sum_{i=1}^{M} \Lambda_i[g_i(\overline{X},\overline{R}) - b_i] \tag{4.57}$$

$$\Lambda_i's \geq 0.$$

The necessary conditions for a maximum to exist form a system of simultaneous equations. The solutions to these simultaneous equations are extreme points in the constraints of the problem. The nonlinear simultaneous equations can be solved by any mathematical algorithm, such as Newton's method, which expresses the multi-variable root-finding problem. Subroutines for solving nonlinear simultaneous equations are available in many mathematical libraries. Examples are ZSCNT and ZSPOW of IMSL [33], and ZONE of PORT mathematical library [53]. These subroutines are accurate, convenient, and efficient. However, they may not converge, and no feasible solution exist.

4.6.2 The branch-and-bound technique in integer programming

The branch-and-bound technique of integer programming for reliability optimization is stated in the paper by Garfinkel and Nemhauser [22]. In step 2 among those steps described in that paper, there are many criteria for selecting the variable for branching [27]. This study selects the variable x_j that minimizes $min(x_i^\star, 1-x_i^\star)$ over index i.

4.6.3 Randomized Hooke and Jeeves method

The Hooke and Jeeves method requires only objective function evaluations and does not use partial derivatives. The Hooke and Jeeves method uses the iterative technique. This method is easy to apply for use on digital computers, since the technique repeats its typical iterative moves: exploratory and pattern. The algorithm can quickly detect and follow a steep valley of a multi-variable function because the information accumulated in previous iterations may be used to find the most profitable search directions. For this reason, the method is a well-known direct search method for unconstrained minimization problems.

However, the Hooke and Jeeves method has some difficulties when it is applied to constrained problems. We may expect the method fails to improve the objective function at the boundary, sharp corners, shallow regions, or ridges.

Although the Hooke and Jeeves method can get the optimal solution in unconstrained minimization problems, slow convergence close to the optimum may be expected. As mentioned before, the Hooke and Jeeves method has some difficulties due to its moving in only one direction at a time when constrained minimization problems are considered. Some methods which modify the Hooke and Jeeves method are proposed. One way to solve these difficulties is to consider random move in n-dimensions instead of one direction at a time when the search is frozen in a certain region.

The simplest concept of random searching inside the n-dimensional hypercube is applied to exploratory move of the Hooke and Jeeves method. Since this method is unbiased in choosing the next moving point, it may be useful to find the location of a global minimum when the objective function has multiple minima. In addition, it may solve the local difficulties of the deterministic methods (Hooke and Jeeves method).

4.6.4 Combination of the randomized H-J method and the branch-and-bound technique

Branching and bounding only the redundancy variables are necessary and sufficient. The above steps can be directly applied to the mixed-integer programming problem. For a 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 randomized Hooke and Jeeves method, their new optimal values are obtained. The branch-and-bound process is stopped whenever all the integer variables find integer values. Multiple near optimal solutions may be achieved to provide management with several options and flexibility.

4.7 Examples

4.7.1 A pure software system

To illustrate the procedure of optimal allocation of software reliability and redundancy, a two-stage series software system without a hardware component is employed. A brief description of procedure follows. At the end of the specification phase, the parameters can be estimated by the use of any of the complexity models. A solution derived from this optimization suggests a general direction for system design to management.

In the early state of the test phase, more accurate data can be collected as times between failures. The form of data should be based on the reliability model chosen in the design phase. The estimation of model parameters can be obtained on the basis of the real data collected. Next, an optimization problem can be set up, depending on the goal of the decision-making process.

A formulation of the optimization problem considered is

$$max \quad R_s = f(\bar{\lambda}, \bar{x}) \tag{4.58}$$

subject to

$$\{C_1 t_r \Delta t_j + C_2 \mu_r \Delta \mu_j\} \cdot kx_j \leq b_1 \tag{4.59}$$

The parameters, cost coefficient and other data needed to solve this optimization problem are given in Table 4.1. Further assume that

$$\lambda_1 = \alpha\lambda \ , \qquad 0 \leq \alpha \leq 1 \ ,$$

$$\lambda_2 = \beta\lambda_3 = \ldots = \beta^{N-2}\lambda_N,$$

and

$$(1-\alpha)\lambda = \lambda_2 + \lambda_3 + \ldots + \lambda_N.$$

With the data given in Table 4.1, the problem was solved by the randomized H-J method and the branch-and-bound method [40]. The optimal solution, shown in Table 4.2, was obtained. After 20 simulation runs with different starting points and random number seeds, the optimal solution of system reliability ranges from $R_s = 0.805$ to $R_s = 0.825$ with the total cost ranging from \$73,100 to \$74,800. The optimal solution also indicates that stage 1 needs two components to optimize the system. The results of this optimization should serve as important input for the decision-making process.

Table 4.1: Data for a numerical example

	stage 1	stage 2
Φ	0.00685	0.00164
N	32.2	42
α,β	0.95, 0.20	
t_r,μ_r	1, 0.1	
C_1,C_2	42, 40	
b_1	75,000	
k	k=0.4 for redundancies	
S	20	

The final step is to allocate or reallocate the residual resources on the basis of resources required for each stage. Since the failure rate of each component is different, the time required to reach the projected failure rate or the reliability of each component is different. By using Equations 4.6 and 4.7 or Equations 4.13 and 4.16, the resources required to reach the projected component reliability can be calibrated. In this example, stage one needs 26,427 units and stage two needs 48,207 units of resources. Management should realize that assigning an accurate amount of resources (e.g., number of personnel) to each stage at the beginning can eventually save development cost, time, and efforts.

4.7.2 A hardware and software mixed system

The system in this example has two components in series. The first component (stage A) can not have any redundant component and the cost of developing hardware component is trivial. The reliability and development cost of hardware

Table 4.2: Optimal solution

	stage 1	stage 2
X_i	2	1
λ_i	0.00685	0.00984
cost	\$26,427	\$48,207
total cost	\$74,634	
R_S	0.81	

component of stage A is known. The second component (stage B) can have redundant components. Each hardware component in stage B has a corresponding software. The reliability and developing cost of hardware component of stage B are also known. Softwares of both stages A and B have not yet been developed. The failures of stage A and stage B are independent. However, there are common-cause failure among software redundancies of stage B.

For the purposes of this study each component in the system is said to have failed if the output from corresponding component is not the same as it designed. The reliabilities and development costs for both hardware components are given in Table 4.3. The model used in this example is NHPP model. The procedure of optimizing the system is the same as the pure software system example discussed in the previous section.

In the early state of the test phase, more accurate data can be collected as times between failures. The estimation of NHPP model parameters can be obtained on the basis of the real data collected and are given in the Table 4.4. Here, the resource usage parameters t_r and μ_r are unit because the time used in this model assumed to be an actual calendar time.

Table 4.3: Component reliabilities and development costs

	900-hr reliability	cost
H/W comp. in stage A	0.985	2380
H/W comp. in stage B	0.980	3400

Next, an optimization problem can be set up, depending on the goal of the decision-making process. A formulation of the optimization problem considered is

$$max \;\; R_s = f(\overline{R}, \overline{x}) \tag{4.60}$$

subject to

$$\{C_1 t_r \Delta t_j + C_2 \mu_r \Delta \mu_j\} \cdot k x_j \leq b_1 \tag{4.61}$$

With the data given in Tables 4.3 and 4.4, the problem was solved by the randomized Hooke and Jeeves method and the branch-and-bound method. The optimal solution, shown in Table 4.5, was obtained. The optimal solution indicates that stage B needs two components to optimize the system and enhances the system reliability up to 0.9814 from 0.9639. The results of this optimization should serve as an important input for the decision-making process.

Table 4.4: Data for a mixed system example

	stage A	stage B
a,b	0.0093, 138.37	0.023, 64
α,β	0.985, 0.120	
t_r,μ_r	1, 1	
C_1, C_2	48, 40	
b_1	100,000	
k	k=0.3 for redundancies	
S	900	

Table 4.5: Optimal solution

	stage 1	stage 2
x_i	1	2
R_i	0.9970	0.9965
cost	61,000	30,000
total cost	100,000	
R_s	0.9814	

5 SOFTWARE QUALITY MANAGEMENT

Software technology has been criticized by dissatisfied users for its poor quality, cost overruns, and late delivery. It has been recognized that the modern software development methodologies, such as structured programming, structured analysis, structured design technique, and others, can hardly solve these problems. The main purpose of this chapter is to suggest protocols to handle these problems of software development by applying statistical quality control to every phase of the testing cycle. Today, management in the software industry knows that despite the most painful efforts to control product quality, variation in product quality is unavoidable. Through the use of process control techniques, such as statistical control charts, unusual variations in the software development process can be controlled and reduced.

Finding faults after the failure occurrence can at least prevent software with low reliability from plaguing users, but this is not what statistical software quality control is all about. The statistical software quality control affects not only the end product but also its process. It involves new practices, dealing with new personnel, learning new forms of communications, and providing a new concept of development. To develop high-quality software efficiently, the statistical software quality control procedure must be planned, with each specified step related to a development activity.

The software testing process is not exhaustive but instead is more representative of a sampling process. If our testing process only observes portions of the entire population and removes only the discovered faults among the latent faults, the software testing process is fundamentally the same as the statistical sampling process of manufacturing.

By use of Cho's SIAD (Symbolic Input Attribute Decomposition) [12] random test input for statistical quality control can be obtained. Cho defines SIAD as a tree element, representing the input domain of a software entity arranged in a linear list with the structure preserved by a set of tree symbols for random sampling. The SIAD tree represents the input domain of a piece of software in a form that facilitates construction of random test input units for producing random product units for quality inspections. Four types of SIAD trees have been developed: regular, weighted, ruled, and network.

It is the intention of this study to provide a guideline or a standard procedure for using statistical quality control, to find the outcomes of variation, and to identify their causes in software development. The proposed process of software quality control may not be very radical but more a formalization of quality control practices on the software development and recording of them in a way that allows developers to know what they are doing and why they are doing it.

The proposed process of software quality control is depicted in Figure 5.1 and steps involved with software quality control are:

- Description of software quality variation outcome

- Data collection design
- Data charting design
- Concurrent data collection and charting
- Variation analysis (Interpretation)
- Cause identification
- Cause elimination

5.1 Review on SIAD Trees and Input Domain Reliability Model

5.1.1 SIAD tree

Up to 50 percent of the requirements for software development never get addressed in a proper manner in the industry. Specifically, in current software development practice, test requirements are missing from the requirements specification. Software engineering (development) requirements, software requirements, test requirements, and documentation requirements are four parts of necessary specification requirements.

After identifying the software engineering goals and principles, the detailed input, output, and processing requirements should be specified. These requirements are equivalent to raw materials, industrial processing, and product design requirement in the manufacturing industries. One of the way to identify the input requirements is using a convenient form called the SIAD (Symbolic Input Attribute Decomposition) tree for software design and test design.

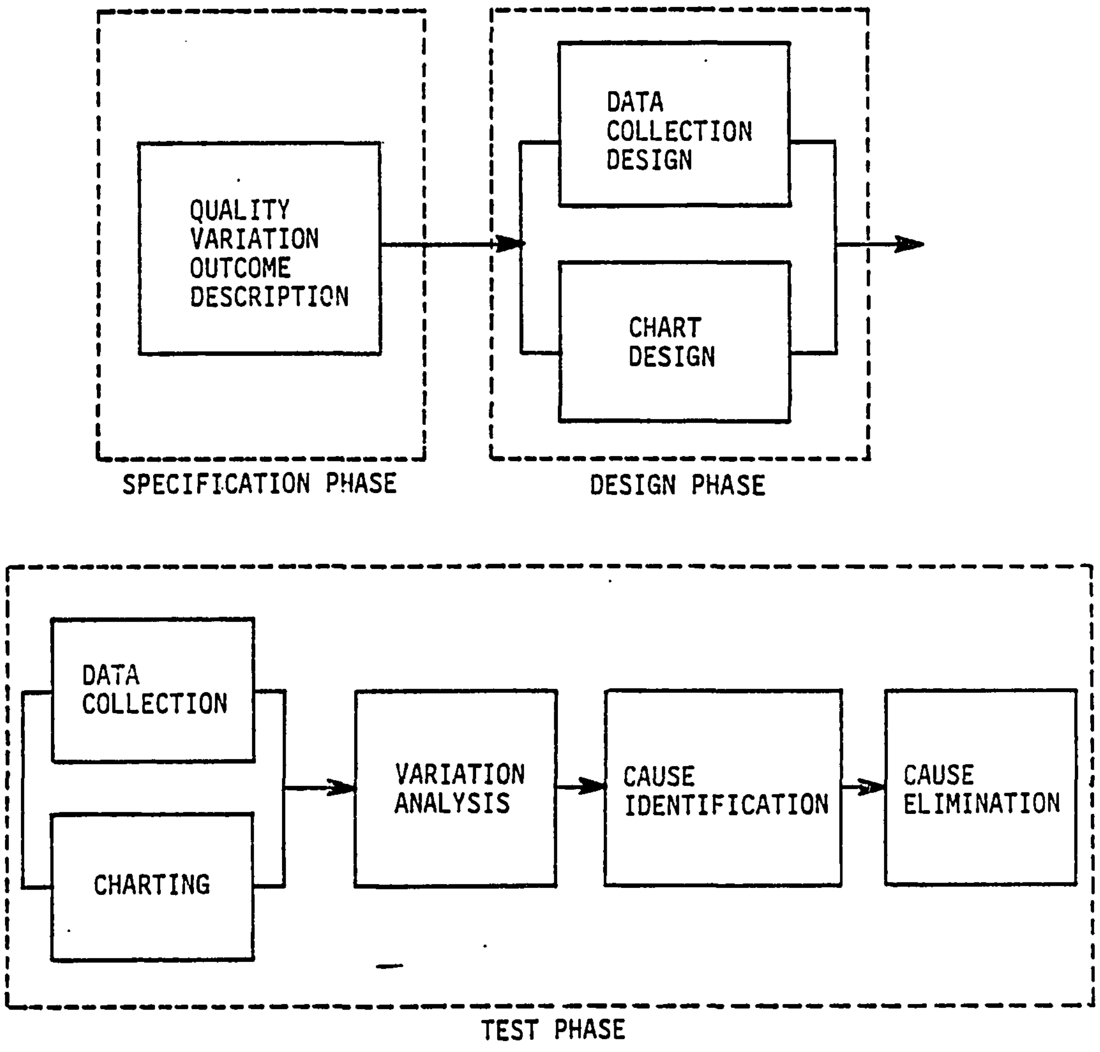


Figure 5.1: Software quality control process

The statistical quality control begins by taking random sampling from input domain which has been specified by the SIAD tree. Four types of SIAD trees have been developed by Cho so far: regular, weighted, ruled, and network.

5.1.1.1 Regular SIAD tree Figure 5.2 shows an example structure. The symbols A, B, ..., U are called the tree elements. A tree symbol in Table 5.1 shows the relationship of an element to other elements. A tree so arranged is a regular SIAD tree, as each element is index column. An SIAD tree is used a tool for describing the input domain of a piece of software and as a basis for construction of test input units using random sampling, which makes it possible to apply the principle of statistical quality control.

A set of random numbers between 1 and N is produced using a random number generator. The element with its index equal to the random number is selected. Say, two elements are to be taken from the tree using the random numbers 3 and 8. The elements D and L are drawn for constructing the test input unit. An element is tested with all other relevant elements by way of tree symbols. Table 5.2 shows the list of relevant elements with each sampled element. This listing gives a meaningful description of each sampled element for guiding test input unit design.

5.1.1.2 Weighted SIAD tree A weighted SIAD tree is identical to a regular SIAD tree, except that each tree element in the weighted SIAD tree is indexed with selected weights or multiple indices. Table 5.3 shows an example of a weighted SIAD tree modified from the regular SIAD tree shown in Table 5.1. With the control of weights, different tree elements can have different probabilities of being sampled for test input unit construction.

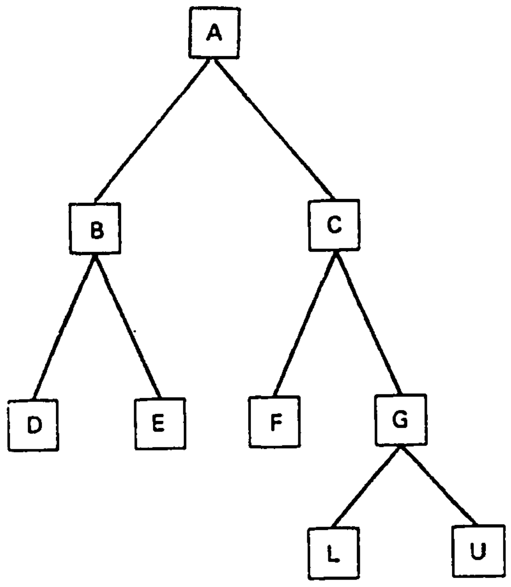


Figure 5.2: A tree structure

Table 5.1: A regular SIAD tree

Index	Tree Symbol	Tree Element
1	X_1	A
2	$X_{1,1}$	B
3	$X_{1,1,1}$	D
4	$X_{1,1,2}$	E
5	$X_{1,2}$	C
6	$X_{1,2,1}$	F
7	$X_{1,2,2}$	G (numerical)
8	$X_{1,2,2,1}$	L (lower bound)
9	$X_{1,2,2,2}$	U (upper bound)

Table 5.2: Test element

Index	Tree Symbol	Tree Element
1	X_1	A
2	$X_{1,1}$	B
3	$X_{1,1,1}$	D
1	X_1	A
5	$X_{1,2}$	C
7	$X_{1,2,2}$	G
8	$X_{1,2,2,1}$	L

Table 5.3: A weighted SIAD tree

Weight	Index	Tree Symbol	Tree Element
2	1-2	X_1	A
4	3-6	$X_{1,1}$	B
8	7-14	$X_{1,1,1}$	D
6	15-20	$X_{1,1,2}$	E
10	21-30	$X_{1,2}$	C
2	31-32	$X_{1,2,1}$	F
5	33-37	$X_{1,2,2}$	G (numerical)
12	38-49	$X_{1,2,2,1}$	L (lower bound)
2	50-52	$X_{1,2,2,2}$	U (upper bound)

5.1.1.3 Ruled SIAD tree A ruled SIAD tree is similar to a regular SIAD tree, except that rules for using the inputs are incorporated into the tree. The rule index and subindex columns are added to regular SIAD tree. These rule index columns specify the general restriction of each tree element.

5.1.1.4 Network SIAD tree The network SIAD tree can be used for applications in which the regular, weighted, and ruled SIAD trees cannot conveniently represent the software input domain. Such applications include operating system, communication networks, and compilers. Using the network SIAD tree, test input units can be constructed systematically for testing the syntax entity by generating a random number of elements.

5.1.2 Input domain reliability model

5.1.2.1 Nelson model Nelson [51] has derived a statistical basis for software reliability assessment based on error correlation with program structure. Data sets are used to execute the program structure. Each specific input data set proceeds through a sequence of segments, with a branch to a new segment taking place at exit of each segment. The sequence of segments in the execution of the program is called a logic path of the program.

For each input data set, the program specifies a computational process by means of which a computer program can computer the function value which the computable function assigns to that input data set. Assuming that inputs are selected independently according to some probability distribution function, the function becomes

$$R(i) = [R(i)]^i = (R)^i \tag{5.1}$$

where $R \equiv R(1)$. The reliability R can be defined as follows:

$$R = 1 - \lim_{n \to \infty} \frac{n_f}{n} \tag{5.2}$$

where n is the number of runs and n_f is number of failures in n runs. This is the operational definition of software reliability of one run.

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.

$$P_k = \begin{pmatrix} M \\ k \end{pmatrix} [1 - R(1)]^k [R(1)]^{M-k}. \tag{5.3}$$

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.

If a sequence of k runs are not selected randomly from the operational profile, R(1) may be different for each run. The maximum likelihood estimate of R(1) can be obtained by running some test cases. It can be expressed as

$$R(1) = 1 - \frac{F_t}{N_t} \tag{5.4}$$

where F_t is the number of test cases that cause failure and N_t is the 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 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

$$R(1) = \sum_i P_i(1 - f_i) \tag{5.5}$$

where P_i is the probability that an input is from partition i and f_i is the 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.

5.1.2.2 Input domain based Stochastic model

The input domain based Stochastic model proposed by Ramamoorthy and Bastani [54] starts from the assumption of reliability growth models. Inputs are selected randomly and independently from the input domain according to the operational distribution. This is a very strong assumption and will not hold in general. The relaxed assumption for general growth model is that input are selected randomly and independently from the input domain according to some probability distribution (which can be change with time).

This means that the effective error size varies with time even though the program is not changed. This permits a modeling of the testing process. 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.

Let

j	number of failures experienced

k	number of runs since the j^{th} failure
T_j(k)	testing process for the k^{th} run
V_j(k)	size of residual errors for the k^{th} run
λ_j	error size under operational inputs
f(T_j(k))	severity of testing process

Then

$$R_j(k|\lambda_j) = \prod_{i=1}^{k} [1 - f(T_j(i))\lambda_j] \tag{5.6}$$

In the above equation, let

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

The testing process is assumed to be identical to the operational environment. Then,

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

Intuitively, errors which are caught later have a smaller size than those which are caught earlier. However, this is true only in a probabilistic sense. This can be modelled by requiring that

$$\Delta_j \leq \Delta_{j-1}.$$

Therefore,

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

5.2 Software Quality Control Process

5.2.1 Description of software quality variation outcomes

The goal of software quality control is to control and, eventually, reduce the unusual variation in the software development process. The first step in the software quality control process is to describe the software quality variation outcomes. The software quality variation outcomes are direct indications of abnormalities observed in statistical control charts. These outcomes of variation should be distinguished from the causes of variation. These outcomes are the possible consequences of causes rather than causes themselves. It is very important to enumerate and specify all outcomes of variation before the data collection design phase. Without a specific purpose in mind or a complete understanding of how the data are to be used, data collection and data analysis are not meaningful. A complete understanding of the outcomes of variation allows more effective use of statistical quality control techniques and makes the elimination of variation easier.

A list of possible outcomes of variation of software development is subjective. The following all outcomes of variation may not be applied to all software, nor be complete in the sense of representing all software projects. A historical record of quality control analysis on quality variations and a careful analysis on the quality control charts of software under development help managers to investigate more outcomes of variation. Suggestions on the study of outcomes of variation follow:

- More-than-error-prone module: Each module has a different structure, a different algorithm, a unique function to perform, and it is developed by a different group of people. Therefore, each software module has a different size and distribution of errors. As module testing goes on, some modules never get better; testing creates as many new faults as it debugs. It is desirable to identify software modules that are behaving in significantly different ways.

- More-than-error-prone personnel: Since each module has a different function to perform and each software developer has a different educational background, a certain type of module may not be suitable for a certain developer. It is necessary to know who tends to create more errors than others in which type of module. Manager may want to assign that personnel to other type of module.

- Near out of control system: When 1) any serious logic error occurs in the program structure, 2) the software does not fit the system's external specification, or 3) the system's initial objectives are misinterpreted, a few adjustments to the software system will not satisfy the user's requirements. Management's attention is required.

- Slow response: A software system may response more slowly than expected to a certain type or amount of input data. Slow response may require the whole module or system to be rewritten with a new algorithm or a different software language. This phenomenon can happen in any testing stage.

- Unusually low failure rate: Gardiner and Montgomery [21] pointed out that an unusually low failure rate could be a potentially troublesome situation. This is also a sign of system variation. It is the indication of either better quality or application of an ineffective or insufficient testing method. Again, management's attention is required.

- Unusually long bug-elimination time: Examination and study of statistical data on fault elimination time should be conducted. Some faults may be easy to detect but not necessarily easy to collect them. Some faults may behave the other way. It might be useful to find the correlation between the number of bugs eliminated and the time spent to eliminate them. These statistic helps the supervisor not only to find unusual variations in the bug-elimination phase but also to predict the total elimination time required for the given initial number of bugs.
- others

5.2.2 Data collection design

Once the software quality variation outcomes are specified, a quality-related data collection should be planned on the basis of each quality variation outcome because a small or moderate amount of intelligently collected data is worth more than a ton of less intelligently collected data. The purpose, methods, and tools of testing change throughout the various phases of the software development. The range of techniques in testing is also extremely broad starting from a syntax-checking within the compiler to design review where the specifications and requirements are tested. The sequence (stage) of testing progress is also numerous. First of all, each software module is independently tested in module testing stage. Syntax-checking, comprehensive checking, various stress points checking, and extremes of the range of variables checking are the variety of module testing. Next, the program structure and the interfaces among these modules are checked in integration testing stage.

Next, the complete software system of which modules are interconnected is tested under the simulated user-environment in the system testing stage. At last, the acceptance test is conducted to check the requirement of software system. In this stage, the developer wants to demonstrate the absence of error, in other words, to convince the purchaser that how good the software is. Meanwhile, the user wants to see the presence of error (i.e., how bug-free software is under the unusual cases).

The design of data collection should be done based on each quality variation outcome of interest, not based on the stage of testing or the methods of testing because the outcomes of quality variation are the facts that the development team wants to detect. The design of data collection could be unique for each quality variation outcome of interest. On the basis of the quality variance outcome, the following questions should be answered. Some of the questions that need to be investigated are:

- which data should be gathered
- how data should be collected
- who should gather data
- when data should be collected
- how much data should be collected

5.2.2.1 Which data should be collected The type of data collected depends on the goal of software quality control (e.g., the outcomes of variation). The followings are some examples of data types based on its outcomes of variation.

1. More-than-error-prone module

   If one of the goals is to detect more-than-error-prone module, it is rational to collect failure rate (e.g., number of faults detected per 100 program lines) per quality measure for all modules. Here, the quality measure is the number of test cases executed.

When data are collected based on time-domain model, the quality measure becomes the execution time for debugging.

2. More-than-error-prone personnel

   Failure rate (e.g., number of faults discovered per 100 lines) per quality measure (the number of test cases executed) is collected for all development personnel.

3. Near out of control state system

   Failure rate (e.g., number of faults discovered per 1000 lines) per quality measure is also collected starting from the integration testing stage. Here, the quality measure could be number of test cases executed, calendar time, or execution time.

4. Slow response

   Response rate (e.g., response time per 1000 line) per volume of data for all modules or system is collected.

5. Unusually low failure rate

   All data types described above are considered as data for the detection of unusually low failure rate outcome.

6. Unusually long bug-elimination time

   Fault collection time of each fault can be a good candidate for the analysis of variation in bug-elimination process.

5.2.2.2 How data should be collected Data should be collected in a format that makes it immediately useful and easy to analyze. Data should be gathered on carefully designed check sheets so that the collected data doesn't need to be transferred to another form.

5.2.2.3 Who should gather data Data should be collected those individuals most familiar with the process of interest. They should be properly trained in data collection techniques and provided with adequate time and resources. Failure identification personnel and failure correction personnel belong to this category. These personnel should be well trained to avoid any misrepresenting data.

5.2.2.4 When data should be collected

- specification phase
- design phase
- testing phase
  - module testing
  - integration testing
  - system testing
  - acceptance testing

Table 5.4 illustrates the relationship between testing stage and the outcomes of variance. The outcomes of variance should be observed under the corresponding testing phase (the one which has the check point). For example, it is better to find more-than-error-prone module in module testing phase. Finding more-than-error-prone module after module test is not desirable.

5.2.2.5 How much data should be collected What is the optimal sampling size? How much data should be corrected? These questions have been a big issue in statistical quality control. In software development, data collection basically lasts the end of software life. However, the size of data and the amount of data collected depends on the goals and objectives of the study, the degree of precision designed, and available resources. The optimal size and amount of sampling which can achieve the best results in statistical software quality control should be further investigated. Moreover, when an error is discovered and corrected, the software is actually changed. Because of the imperfect debugging, the software may be introduced new errors. In the case like this the issue, here, is whether all previous test case should be repeated or not. Further investigation is also required.

5.2.3 Data charting design

In the previous software quality variation description phase, the list of possible outcomes of variation are not completed. Since these outcomes of variation in software development are subjective, hunting down other outcomes of variation subject to current software is an inevitable step.

Charting data into various statistical quality control charts helps developers not only

1. to get statistical evidence of variation, but also

Table 5.4: Outcomes vs. Phases

	more than error prone module	more than error prone personnel	near out of control state system	slow response	unusual low failure rate	unusual long elimination time
module test	√	√		√	√	√
integration test		√	√	√	√	√
system test			√	√	√	√
acceptance test						√

2. to find more outcomes of variation.

First of all, simple histograms, time plots, and other scatter plots so called preliminary control charts are invaluable tools to hunt down outcomes of variation and to eliminate causes of variation. Moreover, these plots give early signal of variation and check the correlation of variables under study.

1. histogram:

   Making specifications on a histogram is a very effective way of communicating to development personnel and fault-discovery personnel what needs to be done to improve the performance of software development. Eventually, histogram gives a new idea to eliminate the source of variation to improve the system. Ex) Pareto diagram: focuses attention on biggest problems first.

2. time plot:

   Constructing a simple time plots before constructing statistically controlled charts might hint at the reliability of system. The scatter represents more unreliability as more time goes.

3. scatter plot:

   Scatter plot is the simplest way to study correlation between two variables. The type of data collected for each outcome of variation was already discussed. Correlation of two variables (failure rate vs. the number of test batch) should be checked to construct the statistical quality control chart.

   Then, ignoring the number of test cases executed, divide the data into convenient subgroups and plot a standard control chart for failure rate. In other words, the data of failure rate should be re-examined by changing the sample size.

Second, the statistically controlled regression control chart should be constructed and analyzed to find the evidence of unusual variation and control the software process. The failure rate (number of fault discovered per a group of lines) collected as the testing continues can not be a linear function of the number of test units executed over the entire range of testing phases (see Fig. 5.3). In this study for simplicity the correlation of these two variables is assumed to be exponentially distributed. As the testing continues, the number of faults discovered tend to decrease. A collection of failure rate points during software testing phase shows the possibility that something other than the standard Shewhart control chart might be desirable. Therefore, the only reasonable choice among the statistical control charts in this case is the regression control chart.

For instance, after a glance at simple histogram or time plots, a suspicion of the existence of variation were found in some modules. Software development team wants to find stronger evidences on the existence of more-than-error-prone module (if there is any) during module testing. The constructed regression control chart is more carefully examined, especially for those of modules which showed a suspicion of the existence of unusual variation.

Assuming linearity (after the transformation of variables), the line which best fits trended data, such as those in Fig. 5.4, may be found by the statistical technique of least squares. This is nothing more than a device for fitting m and b in the straight line equation, $y = mx + b$. With sets of ordinate values (y's) and abscissa values (x's), m and b may be found from

$$m = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2} \tag{5.7}$$

$$b = \frac{(\sum x)(\sum xy) - (\sum y)(\sum x^2)}{(\sum x)^2 - n(\sum x^2)} \tag{5.8}$$

where n = the number of pairs of x, y values.

Here, x becomes the logarithm of failure rates (i.e., the number of faults discovered / a group of line / the number of test cases executed) and y becomes either the logarithm of time periods or that of batches number of testing units. With the equation of the straight line established, the standard error of estimate (σ) is found to use in calculating control limits.

$$r = \frac{(1/n)\sum[(x - \overline{x})(y - \overline{y})]}{\sigma_x \sigma_y} \tag{5.9}$$

$$\sigma = \sigma_y \sqrt{1 - r^2} \tag{5.10}$$

The only remaining task to complete the control chart in Fig. 5.4 is to decide how to put control limits around a line of regression. The decision is closely related to the cost of system (or module) rejection. No solution necessarily universally correct. However, a general rule could be that module testing phase has more tight control limits than integration and system testing phase do.

Perhaps, 2σ or 3σ control limits with tight warning limits may be suitable for module test. Meanwhile, 3σ or 4σ control limits with loose warning limits are justified for integration and system testing phases. Basically, these judgement can be made based on historical records or the expectation of quality of underdeveloping software system.

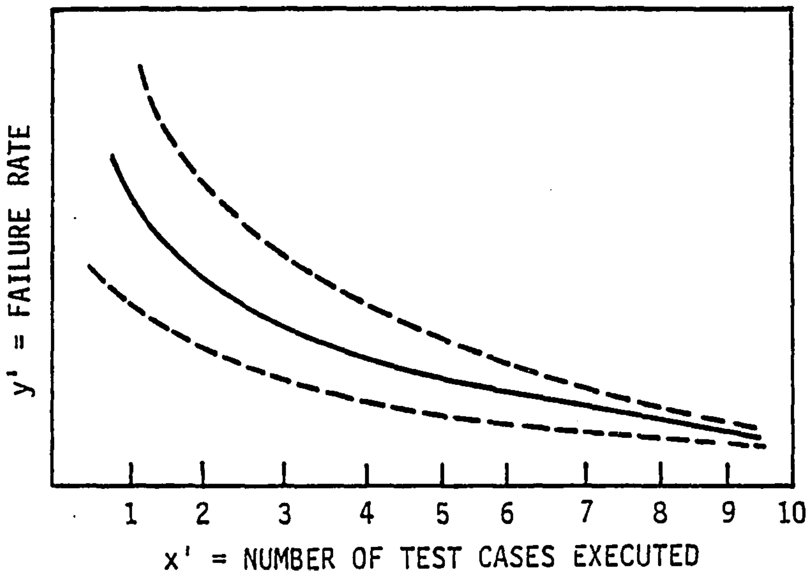


Figure 5.3: Regression line

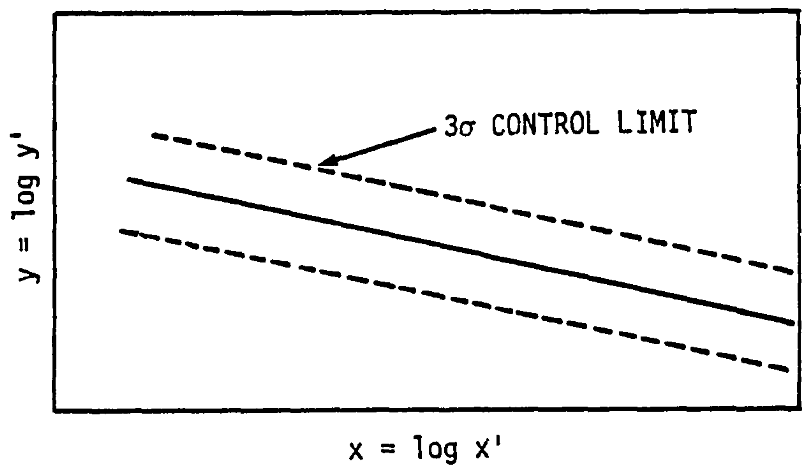


Figure 5.4: Transformed regression line

5.2.4 Concurrent data collection and charting

It should be pointed out that the goal of software quality control is not only to find and locate these software faults, but also to find and reduce the sources of these variation's outcomes. So that, the quality of both underdeveloping software and future software can be improved. Supervisor keeps in mind that variation in a measurement comes from many sources and should working together with fault-correction personnel and fault-discovery personnel to find better way of detecting any quality unusual variation.

5.2.5 Variation analysis

A simple point that strayed beyond preset boundaries (limits) is interpreted as an action signal. In other words, the point beyond boundaries is an evidence that something is wrong.

5.2.6 Cause identification

No matter how carefully specified, designed, and developed software is, the natural variation (sometimes called background noise or chance causes) will present. The natural variation is the result of "nonassignable" causes. The causes that can be identified or assigned is called assignable causes. The assignable causes can create the unnatural variation. A process that is operating with only nonassignable causes of variation present is said to be in statistical control.

The final objective of this chapter is to detect and remove these assignable causes not nonassignable cause. These unnatural variations may be divided into two types:

1. relatively simple: due to a single assignable cause
2. relatively complex: combination of more than two assignable causes

In the former case the cause of variation can usually be found without significant effort. In the latter case major variation should be traced by stratifying or segmenting a data set along the lines of possible sources of variation. The following methods of separating data [30] are used in engineering studies:

- Method A: single break down
- Method B: elimination of variable
- Method C: rearrangements of data
- Method D: designed experiments

Followings are the type of assignable causes of variation and the tools which helps to hunt down the assignable cause of variation.

The assignable cause of variation are:

- Assign new employers (developer or designer, etc.) to an underdeveloping software without sufficient understanding of current software
- insufficient and incorrect specifications
- misunderstand specifications
- insufficient and incorrect testing
- failure to measure the effects of assignable causes and to reduce them

- delays in reporting results of analysis
- improper usage of algorithm
- improper usage of language
- inadequate monitoring of software process
- improper classification of causes (assignable, nonassignable)
- incorrect information about the data collection form
- insufficient instruction on the new data collection form
- others

More suspected assignable causes need to be enumerated to trace down the causes effectively. However, it is not possible to write down all possible assignable causes because the characteristics of these variations are unpredictable, unnatural, inconsistent, and nonhomogeneous. So, when any other evidence or suspicion of the existence of assignable causes were found, separate the data according to suspected sources.

Other tools that might help supervisor to hunt down the assignable causes are:

- fish-bone diagram
- cause and effect diagram

5.2.7 Causes elimination

Before the elimination of the assignable cause that is suspected as the cause of unusual variation, determine whether the real cause have been found. After the elimination of the assignable cause, it is important to check the process returns to stable state.

5.3 Use of Statistical Control Techniques

As Cho [12] claimed in his work, one of the most important design step that is missing during the modeling phase in the current software industry is *input descriptions*. During requirement specification, the types of input data and the rules for using input data should be specified and refined. To do this, we will use the SIAD tree which will help not only to find the location of errors but also to construct the quality control charts. The types of SIAD trees are regular, weighted, ruled and network.

In the software development process, each phase of development should have a quality goal or performance measure. The quality performance measure, first of all, needs to be defined as a vector of quantitative measure based on important variables that can be tracked over program operation time. The vector consists of attributes such as failure intensity along with its confidence limits, failure removal capacity, hardware-related software, and so forth.

Second, the statistically controlled r-chart, run-chart, p-chart and other regression techniques will be constructed and analyzed to monitor and control the software process. Finally, the statistically controlled software system's performance will be demonstrated and predicted in the immediately followed life-cycle phase. This will relate to additional resource (e.g., computation time, failures, and personnel) needed to achieve a specified goal, such as reliability, understandability, efficiency, structure.

5.4 Use of Fault Tree and Event Tree Analyses

Regardless of how refined and correct the product is, the degree of quality of conformance achieved varied from one product unit to the next. The statistical evidence of instability of the software development system should thus be carefully examined. Hence, a standard and generic software development system will be carried out by applying Deming's management philosophy.

It is obvious that the type of action required to reduce special cause of variation is totally different from the action required to reduce common causes variation from the system itself, and those common causes could be any or a combination of possibilities.

- poor design of product
- poor design of software
- insufficient and incorrect specifications
- poor instruction and poor supervision
- insufficient and incorrect testing
- failure to measure the effects of common causes and to reduce them
- failure to provide programmers with information in statistical form that shows them where they could improve their performance and the uniformity of the product
- incoming materials (such as computer languages, existing software mathematical packages) are not suited to the requirement

- others.

According to Dr. Deming's experience [15], the main cause of most troubles and the greatest possibility for improvement belong to the system not the workers. For instance, slow response and numerical error may require the whole module to be rewritten by use of a new algorithm. An event tree and fault tree analysis will be used to identify the critical flaws in the software development, to distinguish a special cause of variation from a common cause of variation, and to aid management in taking the proper action required to reduce the given cause of the variation. Consequently, this event tree and fault tree analysis reduces risk (e.g., high failure rate) due to common cause failures. Risk analysis has shown that no matter how small events are, they can be amplified to increase system failure. A cost factor incurred in each branch of the above analysis will be estimated. Quantity and variety of common causes will be determined.

6 CONCLUSIONS

In this study, a new procedure bases on system optimization concept for improving software reliability has been provided. The software reliability-related cost function and the reliability function of software redundancy with the common-cause failure model have been investigated and provided. In the middle of the concurrent coding and test phase, the system and component reliabilities are examined. Since more information about the developing software, such as failure intensity or failure rate, is readily available at this time, more accurate system and component reliabilities can be reevaluated with updated data.

If the system and module reliabilities don't meet the reliabilities required, the resources can be reallocated and the system optimization problem can be solved, again with the updated data. A set of solutions along with determined decision variables can be obtained. The management chooses a solution from among the new multi-optimal solutions obtained. The decision to be made is whether to improve module's reliabilities or to increase the number of redundancies of some modules through manageable ways. This iteration continues until the current reliabilities meet the requirement.

In the software quality management chapter, a standard procedure using statistical quality control for eliminating the causes of variation and improving the quality of software has been provided. In the preliminary control charting phase, the correlation of two variables based on input domain testing should be examined. When the distribution of two variables can not be clearly identified, the data of failure rate should be re-examined by changing the sample size. The simplest way of doing this is to divide the data into several subgroups by ignoring the number of test cases executed.

It is the management responsibility to detect all unusual variations and to remove all assignable causes. The remaining variation must be left to nonassignable causes, so that, the process remains in the state of statistical control.

A lot of work should be done in developing a good testing method. At least the standardization of testing method should be done. There are over 80 software reliability models proposed. However, many developers have claimed that none of those software reliability models works very well. The author believes that many software reliability models can effectively quantify the quality of software and have proven their accuracy and effectiveness in the application of many mid-size softwares. The problem is not in the correctness of those software models but in the collection of good quality data that is meaningful and sound in statistically. In order to get quality data, a good testing method should be developed. The testing method should be able to provide sound data consistently for any kind of software.

There is a question about the software reliability as being a good software quality measurement. The traditional design techniques and testing methods are too customized to get statistically sound data. Until there is a good testing method, the quality of software may be quantified by use of other tools, say, the reliability bypass models.

As the extension of this study, reliability models for different configurations of system can be derived. Cold standby redundant system and multi-version programming are good examples. The reliability model for cold standby redundancy can be derived by modifying the Markov process discussed and that of multi-version programming can be derived by use of probability theorem.

In both cases, the analytical system reliability function for generic N-component should be evaluated. In the optimization problem formulation, both the available resources and system parameters could vary over their expected range either because of unexpected resource change or because of the nature of statistical uncertainties of the estimated parameters.

The full set of perturbations can be ordered by investigating the sensitivity of all responses to one parameter in a single iteration. Therefore, the major drawback of the conventional perturbation method is that the same procedure has to be repeated for every decision parameter. A second difficulty arises if an analytical form of the system model is not readily available. In this case, the sensitivity coefficients obtained from the perturbation method are only approximations.

The adjoint method is a promising alternative to the above dilemma. The adjoint method requires a detailed system model, which is proposed in Chapter 4. Once the detailed system model is fabricated, a single adjoint run is to be designed to produce exact sensitivity coefficients for all input parameters.

This proposed adjoint method can

- provide the management quick response to check the robustness of the optimal design of Task 1, which will also help reevaluate all possible outcomes,
- identify the critical parameters employed in the system optimization and cross

examine the effects due to SIAD tree, fault tree, and event tree analysis provided in Chapter 5, and

- help maintain an "in-control" state for future software development.

7 ACKNOWLEDGEMENTS

I would like to express my appreciation to many persons who assisted me in this research and in the completion of my Ph.D. program. Dr. Way Kuo, my major professor, deserves special recognition for his encouragement and guidance throughout my research. Without his assistance, consideration, and patience, this research never would have been accomplished. Without his expertise, insight, and criticism, four invaluable papers which we have published never would have been accomplished.

I also would like to thank my committee members: Dr. Howard Meeks, Dr. Vincent Sposito, Dr. William Meeker, Dr. Geraldine Montag, all of whom provided valid criticisms and comments concerning my final dissertation. My special thanks go to Dr. Montag for the opportunity to work under her on the course 'work measurement' to broaden my experience in teaching and for professional growth while serving as a teaching assistant. Many thanks to Dr. Yasuo Amemiya for his hospitality and valuable assistance.

Last, but most importantly, I want to acknowledge the members of my family. First, I'd like to thank my parents for providing me with a lot of valuable encouragement and support throughout. This research would not have been possible without my parents' dedication to my education.

Next, I'd like to thank my daughters, Hanna and Min-sun, for never complaining about having a father who was frequently preoccupied. I can't say enough to thank my wife, Ik-Ja. She was always there to give me what I needed most, care, forgiveness, love. She has endured so much and asked so little. I do always love her and dedicate this work.

To all of these, other family members, and other friends, I am most humbly grateful.

8 BIBLIOGRAPHY

[1] Aho, A. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[2] Akiyama, F. "An Example of Software System Debugging." *IFIP Congress,* Ljubljana, Yugoslavia, 1971, 353-359.

[3] Barlow, R. and Scheuer, E. "Reliability Growth during a Development Testing Program." *Technometrics,* 8, No. 1 (February 1966): 53-60.

[4] Barlow, R., et al., eds. *Reliability and Fault Tree Analysis.* Society for Industrial and Applied Mathematics, Philadelphia, 1975.

[5] Basili, V. R. "Quantitative Software Complexity Models: A Panel Summary." *Proc. Workshop Quant. Software Models for Reliability, Complexity, and Cost: An Assessment of the State of the Art,* Oct. 9-11. *IEEE,* New York, 1979, 243-245.

[6] Basili, V. R. and Hutchens, D. H. "An Empirical Study of a Syntatic Complexity Family." *IEEE Trans. Software Engineering,* SE-9, No. 6 (1983): 664-672.

[7] Basili, V. R., Selby, R. W. and Phillips, T. Y. "Metric Analysis and Data Validation across Fortran Projects." *IEEE Trans. Software Engineering,* SE-9, No. 6 (1983): 652-663.

[8] Belady, L. A. "On Software Complexity." *Proc. Workshop Quant. Software Model for Reliability, Complexity, and Cost: An Assessment of the State of the Art,* Oct. 9-11. *IEEE,* New York, 1979, 90-94.

[9] Boehm, B. W. *Characteristics of Software Quality.* TRW and North-Holland Publishing Co., Amsterdam, The Netherlands, 1978.

[10] Brooks, H. " A Discussion of Random Methods for Seeking Maxima." *Operations Research,* 6 (March 1958): 244-251.

[11] Chen, E. T. "Program Complexity and Program Productivity." *IEEE Trans. Software Engineering,* SE-4, No. 2 (1978): 187-194.

[12] Cho, C. K. *Quality Programming.* John Wiley & Sons, New York, 1987.

[13] Cooper, J. D. and Fisher, M. J. *Software Quality Management.* A Petrocelli Book, New York, 1979.

[14] Coutinho, J. S. "Software Reliability Growth." *IEEE Symp. Comp. Software Reliability* (1973): 58-64.

[15] Deming, W. E. *Out of the Crisis.* Massachusetts Institute of Technology, Cambridge, Mass., 1986.

[16] Dickson, J., Hesse, J., Kientz, A. and Shooman, M. "Quantitative Analysis of Software Reliability." *Proc. Ann. Reliability and Maintainability Symp.,* IEEE (January, 1972): 148-157.

[17] Duran, J. W. and Wiorkowski, J. "Capture-recapture Sampling for Estimating Software Error Content." *IEEE Trans. Software Engineering,* SE-7, No. 1 (1981): 147-148.

[18] Echhardt, D. E., Jr. and Lee, L. D. "A Theoretical Basis for the Analysis of Multi-version Software Subject to Coincident Errors." *IEEE Trans. Software Engineering,* SE-11, No. 12 (1985): 1511-1517.

[19] Endres, A. "An Analysis of Errors and Their Causes in System Program." *IEEE Trans. Software Engineering,* SE-1, No. 2 (1975): 140-149.

[20] Freeman, H. and Lewis, P. M. *Software Engineering.* Academic Press, New York, 1980.

[21] Gardiner, J. S. and Montgomery, D. C. "Using Statistical Control Charts for Software Quality Control." *Quality and Reliability Engineering International,* 3 (1987): 15-20.

[22] Garfinkel, R. S. and Nemhauser, G. L. *Integer Programming.* John Wiley & Sons, New York, 1972.

[23] George, J. S. and Wolverton, R. W. "An Analysis of Competing Software Reliability Models." *IEEE Trans. Software Engineering,* SE-4, No. 2 (1978): 104-120.

[24] Glass, R. L. "Persistent Software Errors." *IEEE Trans. Software Engineering,* SE-7, No. 2 (1981): 162-168.

[25] Goel, A. L. and Okumoto, K. "Time-dependent Error Detection Rate Model for Software Reliability and Performance Measures." *IEEE Trans. Reliability,* R-28, No. 3 (1979): 206-211.

[26] Graham, R. M. *Performance Prediction.* Advanced Course on Software Engineering, No. 81. Springer-Verlag, New York, 1973, Chapter 4.

[27] Gupta, O. K. and Ravindran, A. "Branch-and-bound Experiments in Convex Nonlinear Programming." *Management Science*, 31, No. 12 (1985): 1533-1546.

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

[29] Hammersley, J. M. and Handscomb, D. C. *Monte Carlo Methods.* Methuen, London, 1975.

[30] *Handbook of Statistical Quality Control.* Western Electric Co., Charlotte, North Carolina, 1956.

[31] Hooke, R. and Jeeves, T. A. "A Direct Search Solution of Numerical and Statistical Problems." *J. Assoc. Comp. Mach.,* 8 (April 1961): 212-229.

[32] Huang, X. Z. "The Hypergeometric Distribution Model for Predicting the Reliability of Software." *Microelectronics and Reliability,* 24, No. 1 (1984): 11-20.

[33] *IMSL Library Reference Manual.* International Mathematical and Statistical Libraries, Inc., Houston, Texas, 1984.

[34] Iyer, R. K. and Valardi, P. "Hardware-related Software Errors: Measurement and Analysis." *IEEE Trans. Software Engineering,* SE-11, No. 2 (1985): 223-231.

[35] Jacoby, S. L. S., Kowalik, J. S., and Pizzo, J. T. *Iterative Methods for Nonlinear Optimization Problems.* Prentice-Hall, Englewood Cliffs, 1972.

[36] Jelinski, Z. and Moranda, P. B. "Software Reliability Research." in *Statistical Computer Performance Evaluation,* W. Freiberger, Ed., Academic Press, New York, 1972, 465-484.

[37] Jensen, R. W. and Tonies, C. C. *Software Engineering.* Prentice-Hall, Englewood Cliffs, 1979.

[38] Knight, J. C. and Leveson, N. G. "An Experimental Evaluation of the Assumption of Independence in Multi-version Programming." *IEEE Trans. Software Engineering,* SE-12, No. 1 (1986): 96-109.

[39] Kuo, W. and Lin, H. H. "Taxonomy and Validation of Software Reliability Model." *ACM Computing Surveys,* Submitted, 1987.

[40] Kuo, W., Lin, H. H., Xu, Z. and Zhang, W. "Reliability Optimization with the Lagrange Multiplier and Branch-and-bound Techniques." *IEEE Trans. Reliability,* R-36, No. 5 (1987): 624-630.

[41] Lin, H. H. and Kuo, W. "Reliability Related Software Life Cycle Cost Model." *Proc. 1987 Annual Reliability and Maintainability Symp.,* 1987, 364-368.

[42] Littlewood, B. and Verrall J. L. "Likelihood Function of a Debugging Model for Computer Software Reliability." *IEEE Trans. Reliability,* R-30, No. 2 (1981): 145-148.

[43] McCabe, T. J. "A complexity measure." *IEEE Trans. Software Engineering,* SE-2 (1976): 308-320.

[44] McCammon, S. "Applied Software Engineering: A Real-time Simulator Case History." *IEEE Trans. Software Engineering,* SE-1, No. 4 (December, 1975): 377-383.

[45] Miller, K. S. *Linear Differential Equations.* W. W. Norton and Company Inc., New York, 1963.

[46] Misra, K. B. "Reliability Optimization of a Series-parallel System." *IEEE Trans. on Reliability,* R-21, No. 4 (1972): 230-238.

[47] Morey, R. C. "Estimating and Improving the Quality of Information in a MIS." *Communications of the ACM,* 25, No. 5 (1982): 337-342.

[48] Motteler, Z. C. *Introduction to Ordinary Differential Equations.* Prindle, Weber and Schmidt, Boston Mass., 1972.

[49] Mourad, S. and Andrews, D. "The Reliability of the IBM MVS/XA Operating System." *Proc. Int'l Conf. on Fault-Tolerant Computing,* 1985, 93-98.

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

[51] Nelson, E. "Estimating Software Reliability From Test Data." *Microelectronics and Reliability,* 17, No. 1 (1978): 67-74.

[52] Okumoto, K. "A Statistical Method for Software Quality Control." *IEEE Trans. Software Engineering,* SE-11, No. 12 (1985): 1424-1430.

[53] *PORT Mathematical Subroutine Library.* AT&T Bell Laboratories, Inc., Murray Hill, New Jersey, 1984.

[54] Ramamoorthy, C. V. and Bastani, F. P. "Software Reliability-Status and Perspectives." *IEEE Trans. Software Engineering,* SE-8, No. 4 (1982): 354-371.

[55] Ramzan, M. T. "Seeded Bug Volume for Software Validation." *Microelectronics and Reliability,* 23, No. 5 (1983): 981-988.

[56] Reifer, R. J. "Software Failure Modes and Effects Analysis." *IEEE Trans. Reliability,* R-28, No. 3 (1979): 247-249.

[57] Rubey, R. J., Dana, J. A. and Biche, P. W. "Quantitative Aspect of Software Validation." *IEEE Trans. Software Engineering,* SE-1, No. 2 (1975): 150-155.

[58] Schick, G. J. and Wolverton, R. W. "An Analysis of Competing Software Reliability Models." *IEEE Trans. Software Engineering,* SE-4, No. 2 (1978): 104-120.

[59] Schick, G. J. and Wolverton, R. W. "Achieving Reliability in Large Software System." *Proc. Annual Reliability and Maintainability Symposium,* 1974, 302-319.

[60] Sharz, S. M. and Wang, J. P. "Introduction to Distributed-Software Engineering."*Computer,* 21 (Oct. 1987): 23-30.

[61] Shooman, M. L. *Quality Programming.* McGraw-Hill, New York, 1972.

[62] Shooman, M. L. *Probabilistic Reliability: An Engineering Approach.* McGraw-Hill, New York, 1968.

[63] Shooman, M. L. and Laemmel, A. "Statistical Theory of Computer Programs: Information Content and Complexity." Digest of Papers, Fall COMPCON'77, *IEEE,* New York, Sept. 6-9, 1977, 341-347.

[64] Trachtenberg, M. "Order and Difficulty of Debugging." *IEEE Trans. Software Engineering,* SE-9, No. 6 (1983): 746-747.

[65] Troy, R. and Roman, Y. "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.