RELIABILITY MODELS AND ANALYSES OF THE COMPUTING SYSTEMS

YUAN-SHUN DAI

(B.Eng. TSINGHUA UNIVERSITY)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF INDUSTRIAL AND SYSTEMS ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE

2003

ACKNOWLEDGEMENTS

This dissertation has evolved over the past four years. I would like to thank my supervisors, Prof. Min Xie and Prof. Kim-Leng Poh, who have advised me a lot on my research. I also appreciate Prof. Ang, Prof. Tang and other faculties for their help in my life and study in the department of Industrial and Systems Engineering.

We are fortunate to have worked closely with many colleagues, such as C.D. Lai, G. Levitin, O. Gaudoin, L.R. Cui, S.H. Ng among others. This has helped broaden our view which is needed for this dissertation as such.

My research is supported by the National University of Singapore. I am also grateful to Ow Lai Chun, Liu Guoquan, Xiang Yanping, and all the other staffs and students in the department of Industrial and Systems Engineering for their help in one way or another.

Finally, I would like to thank my families for their understanding and support during these years.

SUMMARY

Various computing systems have been rapidly developed in recent years and widely implemented in many fields. The functionality, size and complexity of the computing systems keep on increasing nowadays, which makes their quantitative evaluation more difficult than ever before.

Reliability is a useful measure for quantitatively evaluating the computing systems. Intensive studies on reliability models and analytical methods have been carried out to improve the chance that the computing systems will behave satisfactorily in operation. Since the computing operations become more essential and important nowadays, the objective of this dissertation is to study the reliability models and analysis for various computing systems.

As a powerful analytical tool, Markov models have been widely implemented in reliability analysis, so this dissertation comprehensively reviews many typical Markov models and further develops some new ones for different computing systems.

Software and hardware are two major building blocks in the computing systems. They interact together to complete many critical computing tasks. This dissertation systematically studies the reliability of software, hardware and integrated software/hardware systems.

Distributed computing system is widely used today. Its reliability is affected not only by software/hardware but also by network communication. Thus, this dissertation introduces some typical models in the distributed/networked system reliability, and then further develops some new models and analytical methods for it.

"Grid" computing system has emerged as an important new field, distinguished from the conventional distributed computing systems by its focus on large-scale resource sharing, innovative applications, and, in many cases, high-performance orientation. This dissertation originally constructs general reliability models for the grid and presents new analytical methods to estimate the grid reliability related to resource management system, wide-area network communication, and parallel running programs with multiple shared resources.

Multi-state system is also a popular topic in the reliability analysis, which is of recent interest to many researchers. This dissertation also presents some new reliability models for various multi-state systems considering multi-level protections and failure correlations.

Today's software development is no longer an isolated task of a single program. Large systems are usually developed in a multi-language environment and run simultaneously on various platforms. The software testing resource is a kind of entities, which can be measured and controlled early in the development cycle. Thus, for the development of large and complex systems, how to allocate the limited testing resources so that the overall system reliability is maximized is an important decision-making problem. Therefore, this work also studies some interesting optimization problems in the testing resource allocation.

Many models and results found in the literature and from our research are presented in this dissertation. It is hoped that these approaches are easily implemented in practice by engineers/practitioners. It is also hoped that this work is able to serve as reference for students, professors, and researchers in many related fields. Moreover, they may also find some useful ideas for their academic work out of this dissertation.

TABLE OF CONTENT

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

This dissertation focuses on some key issues mainly concerning reliability models and analysis in the area of the computing systems. These key issues include software/hardware/network reliability, reliability of homogeneous/heterogeneous distributed computing systems, grid computing reliability, multi-state system reliability, failure correlation, and optimal testing resource allocation.

This chapter demonstrates the necessity to study the reliability of the computing systems, briefly introduces some basic concepts, presents some commonly used techniques, and finally outlines the scope of this dissertation.

1.1. Need for the Computing System Reliability

Computer systems have been the fastest developing technology during the last century. They have been widely implemented in many areas, and are desired to achieve various complex and safety-critical missions. In our modern society, the applications of the computing systems have now crossed many different fields, for example, air traffic control, nuclear reactors, aircraft, real-time military, industrial process control, automotive mechanical and safety control, telephone switching, bank auto-payment, hospital patient monitoring systems, and so forth.

The size and complexity of the computing systems keep on increasing from one single processor to multiple distributed processors, from individual systems to networked systems, from small-scale program running to large-scale resource sharing, and from local-area computation to global-area collaboration.

The computing systems may contain many processors and communication channels and cover a wide area all over the world. They combine both software and hardware that function together to complete various tasks. They can also run diverse programs and share different resources. They may incorporate multiple states and their failures may be correlated with one another. These interacted factors make system modeling and analysis very complicated and difficult.

Thus, complete, scientific, quantitative measures are required to evaluate the computing systems. Reliability is one such useful measure for evaluating the computing systems. Hence, intensive studies on reliability models and analytical methods have been carried out to improve the chance that the computing systems will perform satisfactorily in operation. As the functionality of computing operations becomes more essential, there is a greater need for the reliability of the computing systems.

Moreover, in order to increase the behavior of the computing systems and to improve its development process, we must make thorough reliability analysis. Based on models and analysis, approaches to improve system reliability can be further implemented, such as optimization methods, heuristic algorithms etc, see e.g. Kuo and Zuo (2003). Although improving the system reliability is more or less studied by this thesis, the thesis still mainly focuses on the reliability measurements, modeling and analysis.

1.2. Computing System Reliability Concepts

In general, basic reliability concept is usually considered as the probability that a system will perform its intended function during a period of running time without any failures (Musa, 1998).

A failure makes system behavior deviate from its specified behavior. The failure may be caused by a fault or by other reasons, such as human mistakes. For example using wrong input data, incorrect printing of output result, misinterpretation of output, etc. may also cause failures. Thus, failure and success are two different possible states of the output. Usually we exclude those failures that are not caused by any faults, so in this case a failure corresponds to one or more faults in the system. After removing those faults, the same failure cannot occur again.

A fault is an erroneous state of a system. Although the definitions of the fault are different for specific systems or under diversified situations, a fault is always an existing part in the system and it can be removed by correcting the erroneous part of the system. Although various terminologies are used to represent the same meaning of the fault by different articles, such as “error”, “bug”, “deficiency”, “mistake”, “defect” and so on, this thesis uses the “fault”, which has been commonly accepted, to represent such meaning.

For the computing system, the basic reliability concept can be adapted to some specific forms such as “software reliability”, “system reliability”, “service reliability”, “system availability” etc for different purposes.

Most computing systems contain software programs to achieve various computing tasks. Software reliability is an important metric to evaluate the software behavior. Similar to the general reliability concept, software reliability is defined as the probability that software will be functioning without failure under a given environmental condition during a specified period of time (Xie, 1991). Here, a software failure generally means the inability of performing an intended task specified by some requirements.

The software reliability is only the measure of a specific software program. In order to evaluate a whole computing system that may contain multiple software programs, system reliability is a good measure. System reliability is defined as the probability that all the tasks for which the system is desired can be successfully completed (Kumar *et al.*, 1986). Those software programs may be structured in parallel or serial topologies or even in an arbitrarily distributed topology, so the system reliability should be respectively computed according to the structure and distributions of these programs. The system reliability is different from the above software reliability by the capacity in tolerating failures. For example, the software reliability does not allow any failure in running the given software program, but the system reliability can tolerate the failures of some programs if the redundancies of those failed programs can make up their functions.

Some computing systems are developed to provide different services for users. The users may only care whether the service they are using is reliable or not, no matter what the conditions of other services are in a system. Thus, from the users' point of view, service reliability is a good measure, which is defined as the probability for a given service to be achieved successfully (Dai *et al.*,2003a). Obviously, the service reliability is different from the above software or system reliability by its focus on part (not all) of the software programs contained in the system.

Moreover, whether the computing system is able to be used or not is also of concern to users. Thus, system availability is another good measurement for this purpose, which is defined as the probability of a system to be available at a time instant (Trivedi, 2001). Different from the reliability that focuses on a period of time when it is free of failures, the system availability is concerned with a time point at which the system is able to be used (i.e. available).

1.3. Approaches to the Computing System Modeling

The computing system reliability is an interesting, but difficult, research area. Although there are many reliability models suggested and studied, none of them are valid at all times and there is no unique model which can perform well for all situations. A reason for this is probably that assumptions made for each model are correct or are good approximations of the reality only in some specific situations.

In a computing system, hardware (such as computers, routers, processors, CPUs, memories, disks, etc.) provides fundamental configurations to support computing tasks. Many traditional reliability models mainly dealt with hardware reliability, such as Duane (1964), Akhtar (1994), Pukite and Pukite (1998), and Trivedi (2001).

Software is another important element in the computing system besides the hardware. Different from the hardware, the software does not wear-out and it can be easily reproduced. Furthermore, software is usually debugged during developing/testing phase so its reliability is changing over time when detecting and removing faults. Many software reliability models have been proposed for the study of software reliability, see e.g. Jelinski and Moranda (1972), Littlewood (1975), Goel and Okumoto (1979), Xie (1991), Lyu (1996), Musa (1998), Pham (2000), Gokhale and Trivedi (1999) and Kuo and Zuo (2003).

However, a computing system usually contains both hardware and software, which ought not to be separately studied. Both types of failures should be integrated together in analyzing the whole system reliability. Many reliability models for the integrated software and hardware systems have been presented recently, such as Goel and Soenjoto (1981), Siegrist (1988), Laprie and Kanoun (1992), Dugan and Lyu, (1994), Welke *et al.* (1995), Garg *et al.* (1999), Trivedi (2001) and Lai *et al.* (2002).

Accompanying the development of network techniques, many computing systems need to communicate through networks. The programs and resources of such systems are distributed or shared all over the networks. This kind of computing system is usually called distributed computing system. Reliability of the distributed computing system is determined not only by the software/hardware reliability but also by the reliability of communication. Therefore, there are many models and algorithms presented for the distributed system reliability, see e.g. Hariri *et al.* (1985), Kumar *et al.* (1986), Chen and Huang (1992), Chen *et al.* (1997), Lin *et al.* (1999, 2001) and Dai *et al.* (2003a).

As a special type of the distributed computing systems, "Grid computing" is a recently developed technique by its focus on various shared resources, large-scale networks, wide-area communications, real-time programs, diverse virtual organizations, heterogeneous platforms etc. Many experts believe that the grid computing will offer a second chance to fulfill the promises of the Internet, see e.g. Foster and Kesselman (1998). Although it is difficult to study the grid reliability due to its complexity, the reliability of the grid computing systems is of much concern now. Dai *et al.* (2002, 2003c) started some initial studies in this new field.

Most reliability models for computing systems assume only two possible states of the system. In reality, many practical computing systems may contain more than two states (Lisnianski and Levitin, 2003), especially for those real-time systems. For example, if some computing elements in a real-time system fail, the system may still continue working but its performance should be degraded. Such a degradation state is another state between the perfect working and completely failed states. To study these types of systems, Multi-State system reliability is also investigated recently by many researchers, e.g. Brunelle and Kapur (1999), Pourret *et al.* (1999), Levitin *et al.* (2003), Wu and Chan (2003) and Zang *et al.* (2003).

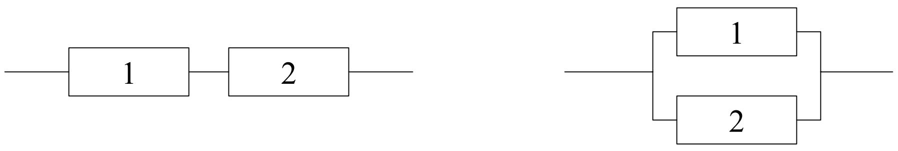
1.4. Common Techniques in Reliability Analysis

There are many techniques in reliability analysis. Some of the most widely used techniques in computing systems are introduced here. They are reliability block diagrams, network diagrams, fault tree analysis and Markov modeling.

1.4.1. Reliability block diagram

A reliability block diagram is one of the conventional and most common techniques of system reliability analysis. A major advantage of using the reliability block diagram approach is the ease of reliability expression and evaluation.

A reliability block diagram shows a system reliability structure. It is made up of individual blocks and each block corresponds to a system module or function. These blocks are connected with one another through certain basic relationships, such as series and parallels. The series relationship between two blocks is depicted by Fig. 1.1 (a) and parallel by Fig. 1.1 (b).



(a) Series connected blocks **(b) Parallel connected blocks**

Fig. 1.1. Basic relationships between two blocks.

The reliability of a block for module *i* is usually assumed to be known, and is denoted by R_i. Assuming that the blocks are independent from a reliability point of view, the reliability of a system with two serially connected blocks is

$$R_s = R_1 R_2$$

and that of a system with two parallel blocks is

$$R_p = 1 - \prod_{i=1}^{2} (1 - R_i)$$

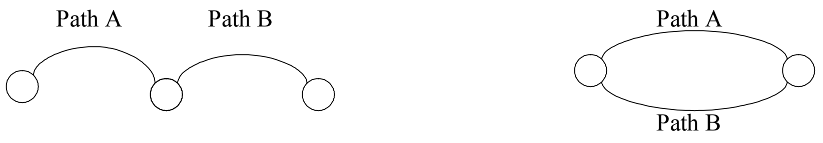
The blocks in either series or parallel structure can be merged into a new block with the reliability expression of the above equations. Using such combinations, any parallel-series system can be eventually merged to one block and its reliability can be easily computed by repeatedly using the above two equations.

Furthermore, a library for reliability block diagrams can be constructed in order to include other configurations or relationships. Additional notational description is needed and specific formulas for evaluating these blocks must be obtained and added to a library, see e.g. Sahner *et al.* (1995).

1.4.2. Network diagram

Network diagrams are commonly used in representing communication networks consisting of individual links. Most network applications are in a communication domain. The computation of network reliability is the primary application of network diagrams, see e.g. Sahner *et al.* (1995) and Findeisen (2000, pp. 48-58).

The purpose of a network is to execute programs by connecting different sites that contain processing elements and resources. For simple network diagrams, computation is not complex and reliability block diagrams can alternatively be used. For example, Fig. 1.2 shows the network diagrams that are connected through series or parallel links.



(a) Series connected links **(b) Parallel connected links**

Fig. 1.2. Network diagram representing series and parallel two links.

Fig. 1.2 can alternatively be represented by the reliability block diagrams if we view each link as a block, depicted by Fig. 1.1.

The choice of reliability block diagram or network diagram depends on the convenience of their usage and description for certain specific problems. Usually, the reliability block diagram is mainly used in a modular system that consists of many independent modules and each module can be easily represented by a reliability block. The network diagram is often used in networked system where processing nodes are connected and communicate through links, such as the distributed computing system, local/wide area networks and wireless communication channels etc.

However, a main disadvantage of the network diagram analysis is that individual links and nodes are assumed to be either operational or failed. This is a Boolean analysis, which limits the application domains of the network diagrams to contain multiple states.

1.4.3. Fault tree analysis

Fault tree analysis is a common technique in system safety analysis, see e.g. Fussell (1975) and Rai *et al.* (1995). Fault tree analyses have been adapted for a range of reliability applications.

The fault tree diagram is the underlying graphical model in fault tree analysis. Whereas the reliability block diagram is mission success oriented, the fault tree shows which combinations of component failures will result in a system failure.

Actually, the fault tree diagram represents the logical relationships of 'AND' and 'OR' among diverse failure events. Various shapes represent different meanings. In general, four basic shapes corresponding to four relationships are depicted by Fig. 1.3.

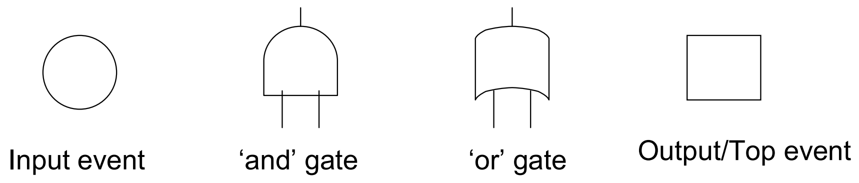


Fig. 1.3. Basic shapes of fault tree diagram.

Since any logical relationships can be transformed into the combinations of 'AND' and 'OR' relationships, the status of output/top event can be derived by the status of input events and the connections of the logical gates. Moreover, repair and maintenance are two important operations in system analysis that can also be expressed by a fault tree, see e.g. Malhotra and Trivedi (1994) and Trivedi (2001).

The fault tree diagram can clarify fault processes and, in particular, fault propagation in a system. However, complex systems exhibit complex failure behavior, including multiple failure modes. These failures will have different effects on a mission outcome. The basic fault tree analysis does not support this type of modeling.

1.4.4. Markov Modeling

The Markov model is another widely used technique in reliability analysis. It overcomes most disadvantages of other techniques and is more flexible to be implemented in reliability analysis for various computing systems, which will be applied in later chapters. We classify the Markov models into two major types: standard Markov models and non-standard Markovian models, which are respectively introduced here.

Standard Markov models

In general, there are four types of standard Markov models corresponding to four types of Markov processes classified according to their state-space and time characteristics as Table 1.1 shows below.

Table 1.1. Four distinct types of Markov processes.

Type	*State Space*	*Time Space*
1	Discrete	Discrete
2	Discrete	Continuous
3	Continuous	Discrete
4	Continuous	Continuous

The standard Markov models satisfy the *Markov property*, which is defined here: for a stochastic process that possesses *Markov property*, the probability of any particular future behavior of the process, when its current state is known exactly, is not altered by additional knowledge concerning its past behavior.

The discrete-state process is referred to as *chain*, so the discrete-state and discrete-time Markov process is usually called discrete time Markov chain (DTMC). Similar to the case of DTMC, discrete-state and continuous-time Markov process is usually called the continuous time Markov chain (CTMC). In the two types of Markov models, the *Chapman-Kolmogorov* equation is famous and often used to solve state probabilities. For details, please refer to Ross (2000).

Since little work has been done in the area of the continuous state (Type 3 and 4 in Table 1.1), the continuous-state Markov process will not be discussed any more in this work. For details about them, please refer to Kijima (1997).

Non-standard Markovian Models

The modeling framework presented above allows the solution of stochastic problems enjoying the Markov property. However, some important aspects of system behavior in a dependability model cannot be easily captured in certain types of the above Markov models. The common characteristic these problems share is that the Markov property is not valid at all time instants. This category of problems is jointly referred to as *non-Markovian* models and can be analyzed using several approaches, see e.g. Limnios and Oprisan (2000).

A set of techniques that are proved very powerful for the solution of non-Markovian models of dependability is based on concepts grouped under the umbrella of Markov renewal theory, e.g. Cinlar (1975) and Fricks *et al.* (1998). It is a collective name that includes Markov renewal sequences and two other important classes of stochastic processes with embedded Markov renewal sequences, named semi-Markov processes and Markov regenerative processes.

1.5. Scope of This Dissertation

This dissertation has nine chapters. A summary is given in Table 1.2.

Table 1.2. The Structure of the Dissertation.

Chapter	Title
1	Introduction
2	Literature Review
3	Parallel Homogeneous Distributed System Reliability
4	Centralized Heterogeneous Distributed System Reliability
5	Grid Computing Systems Reliability
6	Multi-Type Failure Correlation Models
7	Multi-State Systems with Multi-Level Protections
8	Optimal Testing-Resource Allocation
9	Conclusions and Future Work

Chapter 2 comprehensively reviews and systematically classifies the existing work in the area of the computing system reliability.

Chapter 3 to 5 study three types of computing systems that are of much concern now. Chapter 3 studies parallel homogeneous distributed systems considering both software and hardware failures. A perfect debugging case is modeled first, and then an imperfect debugging model is further analyzed. However, chapter 3 does not consider heterogeneous property of the computing systems, so Chapter 4 further studies centralized heterogeneous distributed systems, and service reliability is presented and analyzed. Since the “grid” is a newly developed technology in which its reliability has not been systematically explored, the grid computing system reliability is then studied in Chapter 5.

After studying those specific computing systems, Chapter 6 to 8 solve some difficult problems in the system reliability analysis, such as failure correlations, multi-state systems, multi-level protections, and testing resource allocation. Chapter 6 studies the cases of correlated multi-type failures between successive runs. Chapter 7 extends the optimization of Multi-State System (MSS) structure into a more general case with multi-level protections. Finally, Chapter 8 studies the optimization problems of testing resource allocation on both independent modules and dependent versions.

The last chapter 9 gives conclusions and possible further extensions related to the thesis.

As many models, analyses and algorithms are studied throughout the thesis, it is hoped that these approaches are easily adapted by practitioners. In addition, many examples and case studies are illustrated to help understand and apply those demonstrated techniques.

CHAPTER 2

LITERATURE REVIEW

Many models have been developed in the area of software and system reliability analysis. This chapter reviews and categorizes these models according to their characters, applications and so on. Section 2.1 discusses the existing literatures on Markov models in software reliability, and then Section 2.2 goes over some basic Nonhomogeneous Poisson process (NHPP) models, a special type of Markov models. Finally, Section 2.3 reviews the reliability models for integrated software and hardware systems.

2.1. Markov Models in Software Reliability

Software is an important element in computing systems. Different from hardware, the software does not wear-out and it can be easily reproduced. Furthermore, software systems are usually debugged during developing/testing phase so that its reliability is changing over time when detecting and removing faults. As a result, debugging process usually makes software reliability increase over time. Many software reliability growth models have been proposed for the study of software reliability, e.g. Xie (1991) and Lyu (1996).

Markov model are one of the first type of models proposed in the software reliability analysis. This chapter summarizes software reliability models of this type. As a special type of Markov models, Nonhomogeneous Poisson Process (NHPP) models that are often used in software reliability analysis, are discussed in this chapter too.

2.1.1. Basic Markov model

A basic Markov model in software reliability is the model originally developed by Jelinski and Moranda (1972). It is one of the earliest models and has had a strong influence on many later Markov models which can be considered as modifications or extensions of this basic Markov model.

Model description

The underlying assumptions of the Jelinski-Moranda (JM) model are:

(1) the number of initial software faults is an unknown but fixed constant;

(2) a detected fault is removed immediately and no new fault is introduced;

(3) times between failures are independent, exponentially distributed random variable;

(4) all remaining faults in the software contribute the same amount to the software failure rate.

The initial number of faults in the software before the testing starts is denoted by N_0. According to the assumptions (3) and (4), the initial failure rate is then equal to $N_0 \cdot \phi$, where ϕ is a constant of proportionality denoting the failure rate contributed by each fault. It follows from the assumption (2) that, after a new fault is detected and removed, the number of remaining faults is decreased by one. Hence after the *i*:th failure, there are $N_0 - i$ faults left, and the failure rate decreases to $\phi(N_0 - i)$. This Markov chain is depicted by Fig. 2.1 where state *k* means that there are *k* faults left in the software.

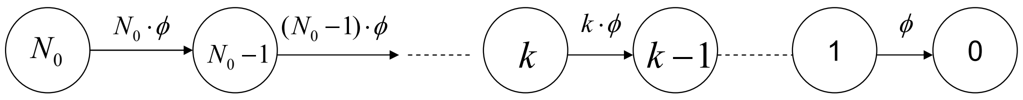


Fig. 2.1. Markov chain of the JM-model.

The time between the (*i*-1):st and the *i*:th failures is denoted by T_i, $i = 1,2,\ldots,N_0$. By the assumptions, T_i's are then exponentially distributed random variables with parameter

$$\lambda(i) = \phi[N_0 - (i-1)] = \phi(N_0 - i + 1), \ i = 1,2,\ldots,N_0 \tag{2.1}$$

The distribution of T_i is given by

$$P(T_i < t_i) = \phi(N_0 - i + 1)\exp\{-\phi(N_0 - i + 1)t_i\}, \ i = 1,2,\ldots,N_0 \tag{2.2}$$

The main property of the JM-model is that the failure rate is constant between the detection of two consecutive failures. It is reasonable if the software is unchanged and the testing is random and homogeneous.

Parameter estimation

The parameters of the JM-model may easily be estimated by using the method of maximum likelihood. Let t_i denotes the observed *i*:th failure-free time interval during the testing phase, i.e. t_i is the observed time between the (*i*-1):st and the *i*:th failure. The number of faults detected is denoted here by *n* which will be called the sample size. If the failure data set $\vec{t} = \{t_1, t_2, ..., t_n; n > 0\}$ is given, the parameters ϕ and N_0 in the JM-model can easily be estimated by maximizing the likelihood function.

The likelihood function of the parameters ϕ and N_0 is given by

$$L(t_1, t_2, ...; N_0, \phi) = \prod_{i=1}^{n} \phi(N_0 - i + 1)\exp\{-\phi(N_0 - i + 1)t_i\}$$

$$= \phi^n \prod_{i=1}^{n} (N_0 - i + 1)\exp\left\{-\phi \sum_{i=1}^{n} (N_0 - i + 1)t_i\right\} \tag{2.3}$$

The natural logarithm of the above likelihood function is

$$\ln L = \ln\left[\phi^n \prod_{i=1}^{n} (N_0 - i + 1)\exp\left\{-\phi \sum_{i=1}^{n} (N_0 - i + 1)t_i\right\}\right]$$

$$= n\ln\phi + \sum_{i=1}^{n} \ln(N_0 - i + 1) - \phi \sum_{i=1}^{n} (N_0 - i + 1)t_i \tag{2.4}$$

By taking the partial derivatives of this log-likelihood function above with respect to N_0 and ϕ, respectively, and equating them to zero, the following likelihood equations can be obtained,

$$\frac{\partial \ln L}{\partial N_0} = \sum_{i=1}^{n} \frac{1}{N_0 - i + 1} - \sum_{i=1}^{n} \phi \cdot t_i = 0 \tag{2.5}$$

and

$$\frac{\partial \ln L}{\partial \phi} = \frac{n}{\phi} - \sum_{i=1}^{n} (N_0 - i + 1)t_i = 0 \tag{2.6}$$

Usually numerical procedures have to be used to solve the two equations. However, the equation system can be simplified as follows. By solving ϕ from the second equation above, we get

$$\phi = n\left[\sum_{i=1}^{n} (N_0 - i + 1)t_i\right]^{-1} \tag{2.7}$$

and by inserting this into (2.5), we obtain an equation independent of ϕ

$$\frac{1}{N_0} + \frac{1}{N_0 - 1} + \ldots + \frac{1}{N_0 - n + 1} = \frac{n\sum_{i=1}^{n} t_i}{\sum_{i=1}^{n} (N_0 - i + 1)t_i} \tag{2.8}$$

The estimation of N_0 can then be obtained by solving this equation. Inserting the estimated N_0 into the expression of ϕ, we can then get the maximum likelihood estimate of ϕ.

Note that the estimation of the number of initial faults might be unreasonable. The problem is expected since the probability of getting disordered data such as observing more failures when their probability should be less, is high initially, see Joe and Reid (1985). Usually more failure data should be accumulated for an estimate to be accurate.

In many cases, the basic Markov model (JM-model) is not good enough and this has led to models with more realistic assumptions. Some extended models, which relax some assumptions of the JM model, are introduced in this section.

2.1.2. Proportional models

Moranda (1979) presented an extended Markov model whose basic assumptions are same as JM model except assume that the (*i*+1):st failure rate is proportional to the *i*:th failure rate, i.e.

$$\lambda_{i+1} = C_i \lambda_i,\ i=0,1,2\ldots \tag{2.9}$$

Its Markov process can be depicted by a Markov chain in Fig. 2.2, where state *i* represents that *i* failures have occurred.

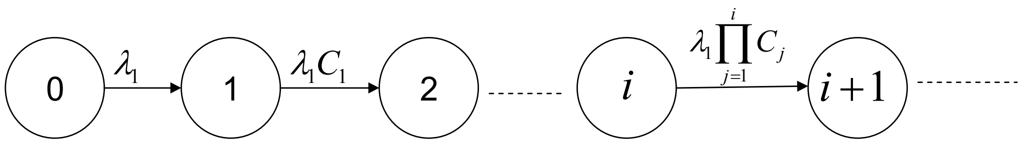


Fig. 2.2. Markov chain for the proportional model.

This kind of model is called proportional model in Gaudoin *et al.* (1994). The idea is to consider that the difference between two successive failure rates is due only to the debugging, and practical constraints lead us to believe that the effect of this debugging is multiplicative. A proportional model is completely defined, given the rate λ_1 and the set *C*= $\{C_1, C_2, C_3, \ldots\}$.

In the simplest proportional model, all parameter values are fixed constants, i.e. λ_1 and *C* are constant. Hence it is called Deterministic Proportional Model. The Deterministic Proportional Model, with parameters λ and θ, is the software reliability model where the random variable T_i are independent and exponentially distributed with parameter

$$\lambda \cdot \exp\{-(i-1)\theta\},\ i \ge 1 \tag{2.10}$$

This model was suggested by Moranda (1979) as geometric de-eutrophication model.

Its detailed statistical property was studied by Gaudoin and Soler (1992).

In fact, the assumption of Deterministic Proportional Model that the C_i (mean quality) is constant, is not realistic. A plausible assumption would be that the mean qualities of the successive debugging are independent random variables Q_i with a homogeneous normal distribution. Then,

$$C_i = \exp(-Q_i)$$

is a lognormal distribution. Gaudoin *et al.* (1994) presented a lognormal proportional model with

$$\lambda_{i+1} = \exp(-Q_i)\lambda_i$$

in which Q_i is normally distributed with mean θ and standard deviation σ.

2.1.3. DFI (Decreasing Failure Intensity) model

A serious critique of the JM-model is that not all software faults are of the same size. Some faults are more easily detected than others. By incorporating this fact, some generalizations and modifications of the JM-model are presented in Xie (1987). We briefly describe this general formulation together with some special cases in this section.

General DFI formulation

The JM-model can be modified by using other function for $\lambda(i)$. Note that $\lambda(i)$ is defined as the rate of the occurrence of the next failure after the removal of the (i-1):st fault. The failure intensity is DFI (Decreasing Failure Intensity) if $\lambda(i)$ is a decreasing function of *i*. A DFI model is thus a Markov counting process model with decreasing failure intensity. DFI models relax the assumption of JM-model that all faults contributed the same amount to the failure probability was not used. The other assumptions of the DFI Markov model are the same as those for the JM-model (Xie, 1991).

It can be observed that the failure rate function $\lambda(i)$ for the JM-model is a linear function of the number of remaining faults. In fact, since at the beginning big faults are likely to be detected, the decrease of the failure rate is probably larger at the beginning than that in the end of the testing phase. As a function of the number of remaining faults, the failure rate function is likely to be convex function.

Under the general assumptions above, the cumulative number of faults detected and removed, $\{N(t), t \geq 0\}$, is a Markov process with decreasing failure rate $\lambda(i)$. The theory for CTMC can be applied.

If $P_i(t) = P\{N(t) = i\}$, $i = 0,1,...,N_0$, the Chapman-Kolmogorov equations are given as

$$P_0'(t) = -\lambda(1)P_0(t)$$

$$P_i'(t) = -\lambda(i+1)P_i(t) + \lambda(i)P_{i-1}(t),\ i = 2,3,...,N_0 - 1 \tag{2.11}$$

$$P_{N_0}'(t) = -\lambda(N_0)P_{N_0-1}(t)$$

with the initial conditions

$$P_0(0) = 1 \text{ and } P_i(0) = 0 \text{ for } i > 0$$

The above equations can easily be solved and the solution is as follows

$$P_0(t) = \exp\{-\lambda(1)t\}$$

$$P_1(t) = \frac{\lambda(1)}{\lambda(2) - \lambda(1)}(e_1 - e_0)$$

$$P_i(t) = \sum_{j=0}^{N_0-1} A_j^{(N_0-1)} e_j\,,\ i = 2,3,\ldots,N_0 - 1$$

and for $i = N_0$, we have

$$P_{N_0}(t) = -\sum_{j=0}^{N_0-1} A_j^{(N_0-1)} \frac{\lambda(N_0)}{\lambda(j+1)} e_j$$

where the quantities e_j, j=0,1,…,$N_0 - 1$, are defined as

$$e_j = \exp\{-\lambda(j+1)\cdot t\},\ j\text{=}0,1,\ldots,\ N_0 - 1$$

and $A_j^{(i)}$ can be calculated recursively through

$$A_j^{(i)} = \frac{\lambda(i)}{\lambda(i+1) - \lambda(j+1)} A_j^{(i-1)},\ j < i$$

$$A_i^{(i)} = -\sum_{j=0}^{i-1} A_j^{(i)}$$

Some specific DFI models

A direct generalization of the JM-model is to use a power-type function for $\lambda(i)$. The power type DFI Markov model was studied by Xie and Bergman (1988) assuming the failure rate

$$\lambda(i) = \phi[N_0 - (i-1)]^\alpha, i = 1,2,\ldots,N_0$$

It is reasonable to assume that $\lambda(i)$ is a convex function of *i* and α is likely to be greater than one, since in this case, the decrease of the failure rate is larger at the beginning.

Another special case of the DFI model is an exponential-type Markov model which assumes that the failure rate is an exponential function of the number of remaining faults. It is characterized by the failure rate function

$$\lambda(i) = \phi[\exp\{-\beta(N_0 - i + 1)\} - 1], \quad i = 1,2,\ldots,N_0$$

For the exponential-type DFI model, the decrease of the failure intensity at the beginning is much larger than that at a later phase.

It is interesting to note that some of the proportional models can also be attributed to DFI model. If all the $C_i < 1$ $(i = 1,2,\ldots,N_0)$ in a proportional model, the failure rate $\lambda(i)$ is actually a decreasing function of the number of remaining faults, which follows the DFI definition.

2.1.4. Time-dependent transition probability models

Sometimes the failure rate function depends not only on the number of detected faults i but also on a local time t_i whose Markov process is shown as Fig. 2.3. This Markov chain is a type of NHCTMC (Non-Homogeneous Continuous Time Markov Chain) model, see e.g. Trivedi (2001).

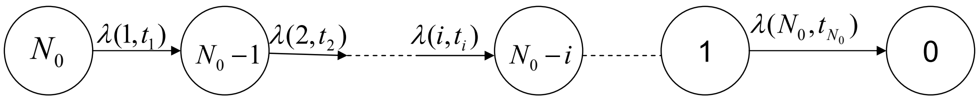

$$\begin{tikzcd} N_0 \arrow[r, "\lambda(1,t_1)"] & N_0-1 \arrow[r, dashed, "\lambda(2,t_2)"] & \cdots \arrow[r, "\lambda(i,t_i)"] & N_0-i \arrow[r, dashed, no head] & 1 \arrow[r, "\lambda(N_0,t_{N_0})"] & 0 \end{tikzcd}$$

Fig. 2.3. NHCTMC for time dependent transition probability models.

There are some existing models which extend the JM-model by assuming that the probability of state change is also time-dependent. Schick-Wolverton model is one of the first models of this type, presented by Schick and Wolverton (1978). The general assumptions made by the Schick-Wolverton model are the same as those for the JM-model except that the times between failures are independent with density function given by

$$f(t_i) = \phi(N_0 - i + 1)t_i \exp\left\{\frac{-\phi(N_0 - i + 1)t_i^2}{2}\right\},\ i = 1,2,..., N_0 \tag{2.12}$$

in which N_0 is the number of initial faults and ϕ is another parameter.

Hence, the main difference between the Schick-Wolverton model and the JM-Model is that the times between failures are not exponential. In the Schick-Wolverton model the failure rate function after detecting the *i*:th fault is

$$\lambda(i, t_i) = \phi(N_0 - i + 1)t_i \tag{2.13}$$

Note that the failure rate function of the Schick-Wolverton model depends both on *i*, the number of removed faults and t_i, the time since the removal of last fault.

The Schick-Wolverton model with time-dependent failure rate was further extended by Shanthikumar (1981). Shanthikumar (1981) model supposes that there are N_0 initial software faults and assumed that after *i* faults are removed, the failure rate of the software is given by

$$\lambda(i, t) = \phi(t)(N_0 - i),\ i=0,1,\dots \tag{2.14}$$

where $\phi(t)$ is a proportionality factor. The parameter estimation can also be carried out using the method of maximum likelihood.

2.1.5. Imperfect debugging models

The imperfect removal of a detected fault is a common situation in practice and the JM model does not take this into account. This section extends the JM-model by relaxing the assumption of perfect debugging process. During an imperfect debugging process, there are two kinds of imperfect removal:

1) A detected fault is not removed successfully while no new fault is introduced;
2) A detected fault is not removed successfully while new faults are generated due to incorrect diagnoses.

For the former type of imperfect condition, it is still a pure death process in the number of remaining faults; while the latter one is in fact a birth-death process in the number of remaining faults. The following will discuss both types of imperfect debugging models, respectively.

Monotonous death process

Goel (1985) suggested a Markov model by assuming that each detected fault is removed with probability *p*. Hence, with probability *q*=1-*p*, a detected fault is not perfectly removed and the quantity *q* can be interpreted as the imperfect debugging probability. This process can be modeled by a DTMC as depicted by Fig. 2.4 where *i* is the number of detected failures.

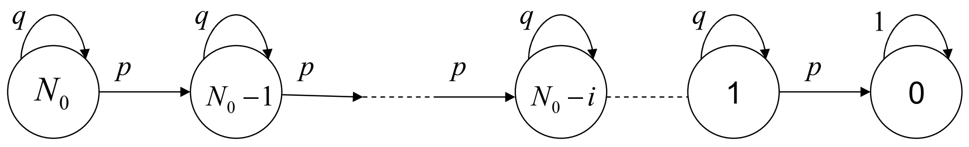

$$\begin{tikzcd} N_0 \arrow[loop above, "q"] \arrow[r, "p"] & N_0-1 \arrow[loop above, "q"] \arrow[r, "p"] & \cdots \arrow[r, "p"] & N_0-i \arrow[loop above, "q"] \arrow[r, dashed] & 1 \arrow[loop above, "q"] \arrow[r, "p"] & 0 \arrow[loop above, "1"] \end{tikzcd}$$

Fig. 2.4. DTMC for the monotonous death process of imperfect debugging model.

The counting process of the cumulative number of detected faults at time t is modeled as a Markov process with transition probability depending on the probability of imperfect debugging. Still it is assumed that times between the transitions are exponential with a parameter which depends only on the number of remaining faults. After the occurrence of the (i-1):st failure, $p \cdot (i-1)$ faults are removed on the average. Hence, approximately, there are $N_0 - p(i-1)$ faults left, where N_0 denotes the number of initial faults as before. The failure rate between the (i-1):st and the i:th failures is then

$$\lambda(i) = \phi[N_0 - p(i-1)]$$

Using this transition function, other reliability measures can be calculated as for the JM-model. Note that the above rate function can be rewritten as

$$\lambda(i) = \phi \cdot p\left[\frac{N_0}{p} - (i-1)\right]$$

and from this it can be seen that it is just the same as that for the JM-model with ϕ replaced by $\phi \cdot p$ and N_0 replaced by N_0/p.

As a consequence, p, N_0 and ϕ are indistinguishable. However, $\phi \cdot p$ and N_0/p can still be estimated similar to that for the parameters in the JM-model and N_0/p can be interpreted as the expected number of failures that will eventually occur. Another advantage of this model is when we know the probability of imperfect debugging, p. For example, from previous experience or by checking after correction, the number of initial faults N_0 and the constant of proportionality ϕ can be estimated.

Birth-death process

Furthermore, if we allow the imperfect debugging process to introduce new faults into the software due to wrong diagnoses or modifications, the debugging process becomes a birth-death Markov process. Kremer (1983) assumes that when a failure occurs, the fault content is assumed to be reduced by 1 with probability *p*, the fault content is not changed with probability *q*, and a new fault is generated with probability *r*. The obvious equality is that

$$p+q+r=1$$

This implies that we have a birth-death process with a death rate $v(t)=r\cdot\lambda(t)$ and a birth rate $\mu(t)=p\cdot\lambda(t)$. It can be depicted by a CTMC as Fig. 2.5.

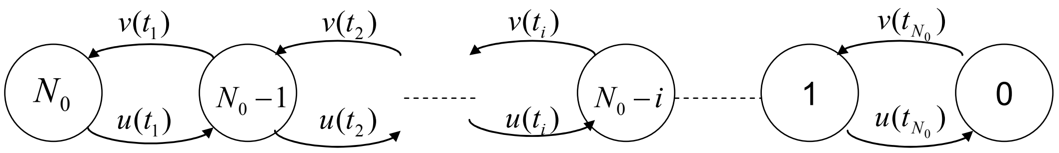


Fig. 2.5. CTMC for the birth-death process of imperfect debugging model.

However, in order to fit failure data and obtain further applicable results, assumptions on the failure rate function $\lambda(t)$ must be made.

Denoted by $N(t)$ the number of remaining faults in the software at time *t* and let

$$P_i(t)=\Pr\{N(t)=i\},\quad i=0,1,\ldots,N_0.$$

We obtain the forward *Kolmogorov* equations of this Markov process as

$$P'_i(t)=(i-1)v(t)P_{i-1}(t)-i[v(t)+\mu(t)]P_i(t)+(i+1)\mu(t)P_{i+1}(t),\ i\geq 0 \quad (2.15)$$

Generally, by inserting $v(t)$ and $\mu(t)$ and using the initial conditions $P_{N_0}(0)=1$, the differential equations can be solved by using the probability generating function suggested in Kremer (1983).

Multi-type failure model considering imperfect debugging

In practice, software failures can be classified into different types according to their severity or characteristics. Different types of failures may cause different software reliability behavior. Although many Markov models assume that the failures have one unique effect on the software, the realistic situation requires the models to treat those failures differently.

Tokuno and Yamada (2000) presented a Markov model with two types of failures that have different kinds of failure rates at the same time incorporating the imperfect debugging process. The first type of failures is due to faults originally latent in the system prior to the testing, denoted by F1. The second type of failures is due to faults randomly introduced or regenerated during the testing phase, denoted by F2.

They assumed that

1) The failure rate for F1 is constant between failures and decreases geometrically as each fault is corrected, and the failure rate for F2 is constant throughout the testing phase;

2) The debugging activity for the fault is imperfect: denoted by p the probability for a fault to be removed successfully and $q(=1-p)$ the probability that fails to remove the fault, similar to the above Monotonous death model (Goel, 1985);

3) The debugging activity is performed without distinguishing between F1 and F2;

4) The probability that two or more software failures occur simultaneously is negligible;

5) At most one fault is corrected when the debugging activity is performed, and fault-correction time is negligible or not considered.

Let $X(t)$ be a counting process representing the cumulative number of faults corrected up to testing time t. From the assumption 2, when i faults have been corrected by an arbitrary testing time t, after the next software failure occurs,

$$X(t) = \begin{cases} i, & (\text{with probability } q) \\ i+1, & (\text{with probability } p) \end{cases} \tag{2.16}$$

from the assumptions 1 and 3, when i faults have been corrected, the failure rate for the next software failure-occurrence is given by

$$\lambda(i) = D \cdot k^i + \theta\,,\ i = 0,1,2,\ldots,\ \ D > 0\,,\ 0 < k < 1\,,\ \theta \geq 0 \tag{2.17}$$

where D is the initial failure rate for F1, k is the decreasing ratio of the failure rate, and θ is the failure rate for F2. The expression of the above equation is from the point of view that software reliability depends on the debugging efforts, not the residual fault content.

The reliability function to the next software failure is given by

$$R_i(t) = \exp\{-(D \cdot k^i + \theta)t\} \tag{2.18}$$

2.1.6. Modular software models

If possible, architecture of software should be taken into account instead of considering the software as a black-box system. Markov models can also be applied in analyzing the modular software reliability.

The Littlewood semi-Markov model

Littlewood (1979) incorporated the structure of the software using a semi-Markov model. This model assumed that the system architecture of continuously running application can be described by an irreducible semi-Markov process.

The program is comprised of a finite number of modules and the transfer of control between modules is described by the probability

$$p_{ij} = \Pr\{\text{program transits from module } i \text{ to module } j\}$$

The time spent in each module has a general distribution $F_{ij}(t)$ which depends upon *i* and *j*, with finite mean m_{ij}. When module *i* is executed, failures occur according to a Poisson process with parameter λ_i. The transfer of control between modules (interfaces are themselves subject to failure; when module *i* calls module *j* there is a probability v_{ij} of a failure occurring.

The interest of the composite model is focused on the total number of failures of integrated software system in time interval $(0,t]$, denoted by $N(t)$. This is the sum of the failures in different modules during their sojourn times, together with the interface failures.

The asymptotic Poisson process approximation for $N(t)$ is obtained under the assumption that failures are very infrequent. The times between failures will tend to be much larger than the times between exchanges of control, that is, many exchanges of control would take place between successive program failures. The failure occurrence rate of this Poisson process is given by

$$\lambda_s = \sum_i a_i \lambda_i + \sum_{i,j} b_{ij} v_{ij}$$

where

$$a_i = \frac{\pi_i \sum_j p_{ij} m_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}}$$

represents the proportion of time spent in module *i*, and

$$b_{ij} = \frac{\pi_i p_{ij}}{\sum_i \pi_i \sum_j p_{ij} m_{ij}}$$

is the frequency of transfer of control between *i* and *j*. These terms depend only on parameters that characterize modular software architecture: transition probabilities p_{ij}, mean execution times m_{ij}, and steady-state probabilities of the embedded Markov chain π_i.

User-oriented model

Similar to the Littlewood semi-Markov model, an interesting model called *user-oriented model*, was developed by Cheung (1980) where user profile was incorporated into the modular software reliability model. The model is a Markov model based on the reliability of each individual module and the inter-modular transition probabilities as the user profile. Also the most critical module of the system can be determined by using sensitivity analysis techniques.

Assume that software is decomposed into a number of modules. It is also assumed that the program flow graph of a terminating application has a single entry and a single exit node, and that the transfer of control among modules can be described by an absorbing DTMC with a transition probability matrix $\boldsymbol{P}=\{p_{ij}\}$. Modules fail independently and the reliability of the module *i* is the probability R_i that the module performs its function correctly, i.e., the module produces the correct output and transfers control to the next module correctly.

Two absorbing states ***C*** and ***F*** are added, representing the correct output and failure, respectively, and the transition probability matrix ***P*** is modified appropriately to $\hat{P}$. The original transition probability p_{ij} between the modules *i* and *j* is modified to $R_i p_{ij}$. This represents the probability that the module *i* produces the correct result and the control is transferred to module *j*. From the exit state *n*, a directed edge to state *C* is created with transition probability R_n to represent the correct execution. The failure of a module *i* is considered by creating a directed edge to failure state *F* with transition probability $1-R_i$. Hence, DTMC defined with transition probability matrix $\hat{P}$ is a composite model of the software system. The reliability of the program is the probability of reaching the absorbing state *C* of the DTMC.

Let Q be the matrix obtained from $\hat{P}$ by deleting rows and columns corresponding to the absorbing states *C* and *F*. $Q^k(1,n)$ represents the probability of reaching state *n* from 1 through *k* transitions. From initial state 1 to final state *n*, the number of transitions *k* may vary from 0 to infinity. It is not difficult to show that

$$S = I + Q + Q^2 + Q^3 + \cdots = \sum_{k=0}^{\infty} Q^k = (I-Q)^{-1} \tag{2.19}$$

and it follows that the overall system reliability can be computed as

$$R = S(1,n)\cdot R_n \tag{2.20}$$

Task-oriented model

A modular software is usually developed to complete certain tasks. Kubat (1989) presented a *task-oriented model* which considered the case of a terminating software application composed of n modules designed for K different tasks. Each task may require several modules and the same module can be used for different tasks. Transitions between modules follow a DTMC such that with probability $q_i(k)$ task k will first call module i and with probability $p_{ij}(k)$ task k will call module j after executing in module i. The sojourn time during the visit in module i by task k has the density function $g_i(k,t)$. Hence, the architecture model for each task becomes an semi-Markov process.

The failure rate of module i is λ_i. As derived by Kubat (1989), the probability that no failure occurs during the execution of task k, while in module i is

$$R_i(k) = \int_0^\infty e^{-\lambda_i t} g_i(k,t)dt \tag{2.21}$$

The expected number of visits in module i by task k, denoted by $V_i(k)$, can be obtained by solving

$$V_i(k) = q_i(k) + \sum_{j=1}^{n} V_j(k) p_{ij}(k);\ i = 1,2,...,n,\ k = 1,2,...,K \tag{2.22}$$

The probability that there will be no failure when running for task k can be approximated by

$$R(k) \approx \prod_{i=1}^{n} [R_i(k)]^{V_i(k)} \tag{2.23}$$

and the system failure rate is calculated by

$$\lambda_s = \sum_{k=1}^{K} r_k [1 - R(k)] \tag{2.24}$$

where r_k is the arrival rate of task *k*.

Multi-type failure model in modular software

Ledoux (1999) further proposed a Markov model to include multi-type failures into modular software reliability analysis. They constructed an irreducible CTMC with transition rates q_{ij} to model the software composed of a set of components *C*. In their Markov model, two types of failures are considered: primary failures and secondary failures. The primary failure leads to an execution break; the execution is restarted after some delay. A secondary failure does not affect the software because the execution is assumed to be restarted instantaneously when the failure appears. For an active component c_i, a primary failure occurs with constant rate λ_i', while the secondary failures are described as Poisson process with rate λ_i''. When control is transferred between two components *i* and *j* then a primary (secondary) interface failure occurs with probability v_{ij}' (v_{ij}'').

Following the occurrence of a primary failure, a recovery state is occupied, and the delay of the execution break is a random variable with a phase type distribution. Denoting by *R* the set of recovery states, the state space becomes $C \cup R$. Hence, the CTMC that defines the architecture is replaced by a CTMC that models alternation of operational-recovery periods. The associated generator matrix defines the following transition rates: from c_i to c_j with no failure; from c_i to c_j with a secondary failure; from c_i to c_j with a primary failure; from recovery state *i* to recovery state *j*;

and from recovery state *i* to c_j.

Thus, the Markov model can be constructed according to the architecture of different modules and their states. Based on the CTMC, the Chapman-Kolmogorov equations can be obtained and solved by certain computational tools.

2.2. NHPP Models in Software Reliability

Although some basic and advanced Markov models are presented in the previous sections, some NHPP models (as a special type of Markov models) are mentioned here due to their significant impact on the software reliability analysis. NHPP is a special class of counting process $\{N(t), t \geq 0\}$ to cumulate the number of events (such as software failures) in a time interval $[0, t)$. It can be classified as a very special case of the NHCTMC (Non-Homogeneous Continuous Time Markov Chain) models, see e.g. Gokhale et al. (1997).

2.2.1. The Goel-Okumoto (GO) model

In 1979, Goel and Okumoto presented a simple model for the description of software failure process by assuming that the cumulative failure process is NHPP with a simple mean value function. Although NHPP models have been studied before, see e.g. Schneidewind (1975), the GO-model is the basic NHPP model that later has had a strong influence on the software reliability modeling history.

Model description

The general assumptions of the GO-model are

1) The cumulative number of faults detected at time *t* follows a Poisson distribution;
2) All faults are independent and have the same chance of being detected;
3) All detected faults are removed immediately and no new faults are introduced;

Specifically, the GO-model assumes that the failure process is modeled by an NHPP model with mean value function $m(t)$ given by

$$m(t) = a[1-\exp(-bt)], \quad a > 0, b > 0$$

The failure intensity function can be derived by

$$\lambda(t) = \frac{d}{dt} m(t) = ab\exp(-bt)$$

where *a* and *b* are positive constant. Note that $m(\infty) = a$. The physical meaning of parameter *a* can be explained as the expected number of faults which are eventually detected. The quantity *b* can be interpreted as the failure occurrence rate per fault.

The expected number of remaining faults at time *t* can be calculated as

$$E[N(\infty) - N(t)] = m(\infty) - m(t) = a - a[1-\exp(-bt)] = a\exp(-bt)$$

A heuristic derivation of the GO-model

Suppose that the expected number of faults detected in a time interval $[t, t+\Delta t)$ is proportional to the number of remaining faults, we have that,

$$m(t+\Delta t) = b[a - m(t)]\Delta t$$

where *b* is a constant of proportionality.

The above difference equation can be transformed into a differential equation. Divide both sides by Δt and take limits by letting Δt tend to zero, we get the following equation,

$$m'(t) = a \cdot b - b \cdot m(t)$$

It can easily be verified that the solution of this differential equation, together with the initial condition $m(0) = 0$, we get the mean value function of the GO-model.

Both the GO-model and JM-model give the exponentially decreasing number of remaining faults. It can be shown that these two models cannot be distinguished using only one realization from each model. However, the models are different because the JM-model assumes a discrete change of the failure intensity at the time of the removal of a fault while the GO-model assumes a continuous failure intensity function over the whole time domain.

It should be pointed out here that the GO-model makes the assumption that all faults contribute the same amount to the software failure intensity which is unrealistic. Some extended models, which relax this assumption, will be discussed later.

Parameter estimation

Denoted by n_i the number of faults detected in time interval $[t_{i-1}, t_i)$, where $0 = t_0 < t_1 < \cdots < t_k$ and t_i are running times since the software testing begins. The estimation of model parameters *a* and *b* can be carried out by maximizing the likelihood function, see e.g. Goel and Okumoto (1979). The likelihood function can be reduced to

$$\sum_{i=1}^{k} \frac{n_i[t_i \exp(-bt_i) - t_{i-1} \exp(-bt_{i-1})]}{\exp(-bt_{i-1}) - \exp(-bt_i)} = \frac{t_k \exp(-bt_k) \cdot \sum_{i=1}^{k} n_i}{1 - \exp(-bt_k)} \tag{2.25}$$

Solving this equation to calculate the estimate of *b*, and then *a* can be estimated as

$$a = \frac{\sum_{i=1}^{k} n_i}{1 - \exp(-bt_k)} \tag{2.26}$$

Usually, the above two equations has to be solved numerically. It can also be shown that the estimates are asymptotically normal and a confidence region can easily be established. A numerical example is illustrated below.

2.2.2. S-shaped NHPP models

The mean value function of the GO-model is exponential-shaped. Based on experience, it is observed that the curve of the cumulative number of faults is often S-shaped as shown by Fig. 2.6, see e.g. Yamada *et al.* (1984).

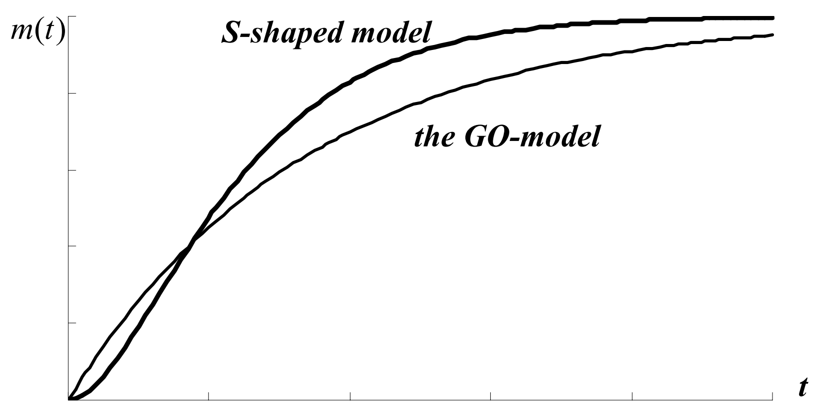


Fig. 2.6. The S-shaped mean value function.

Generally, the S-shaped curve can be explained by the fact that faults are neither independent nor of the same size. At the beginning of the testing, some faults might be “covered” by other faults. Removing a detected fault at the beginning does not decrease the failure intensity very much since the same test data will still lead to a failure caused by other faults. In a later phase, large faults are already removed and the remaining faults have small size so that the fault-detection rate is of moderate size. Also because there are not many faults left in the software, the coverage has no significant effect at the end of the testing phase. Another reason of the S-shaped behavior is the learning effect as indicated in Yamada *et al.* (1984).

Several different S-shaped NHPP models have been proposed in the existing literature. The most interesting ones are the delayed S-shaped NHPP model and the inflected S-shaped NHPP model.

Delayed S-shaped NHPP model

The mean value function of the *delayed S-shaped* NHPP model is

$$m(t) = a[1-(1+bt)\exp(-bt)]\,;\;\; b>0\,, \tag{2.27}$$

This is a two-parameter S-shaped curve with parameter a denoting the number of faults to be detected and b corresponding to a fault detection rate. The corresponding failure intensity function of this delayed S-shaped NHPP model is

$$\lambda(t) = \frac{dm(t)}{dt} = ab(1+bt)\exp(-bt) - ab\exp(-bt) = ab^2 t\exp(-bt)$$

The expected number of remaining faults at time t is then

$$m(\infty) - m(t) = a(1+bt)\exp(-bt)$$

Inflected S-shaped NHPP model

The mean value function of the *inflected S-shaped* NHPP model is

$$m(t) = \frac{a[1-\exp(-bt)]}{1+c\exp(-bt)};\ \ b>0, c>0$$

In the above *a* is again the total number of faults to be detected while *b* and *c* are called the fault detection rate and the inflection factor, respectively. The intensity function of this inflected S-shaped NHPP model can easily be derived as follows.

$$\lambda(t) = \frac{dm(t)}{dt} = \frac{ab(1+c)\cdot\exp(-bt)}{[1+c\exp(-bt)]^2}$$

Given a set of failure data, for both delayed and inflated S-shaped NHPP models, numerical methods have to be used to solve the likelihood equation so that estimates of the parameters can be obtained.

2.2.3. Some other NHPP models

Besides the S-shaped models, there are many other NHPP models that extend the GO-model for different specific conditions.

Duane NHPP models

Here, we will briefly describe some existing reliability growth models which inherit or modify the Weibull model properties. The first is the Duane model interpreted as a NHPP model for reliability growth, see e.g. Duane (1964). A modification of this model due to Littlewood (1984) is also presented.

The Duane model is referred to as the Weibull process model assumes that the mean value function satisfies

$$m(t) = \left(\frac{t}{\alpha}\right)^{\beta}, \quad \alpha > 0, \ \beta > 0 \tag{2.28}$$

In the above, α and β are parameters which can be estimated by using collected failure data. The mean value functions with $\alpha = 100$ and different $\beta = \{0.5, 1, 2\}$ are depicted by the Fig. 2.7. When $\beta = 1$, the Duane NHPP model is reduced to a Poisson process whose mean value function is a straight line.

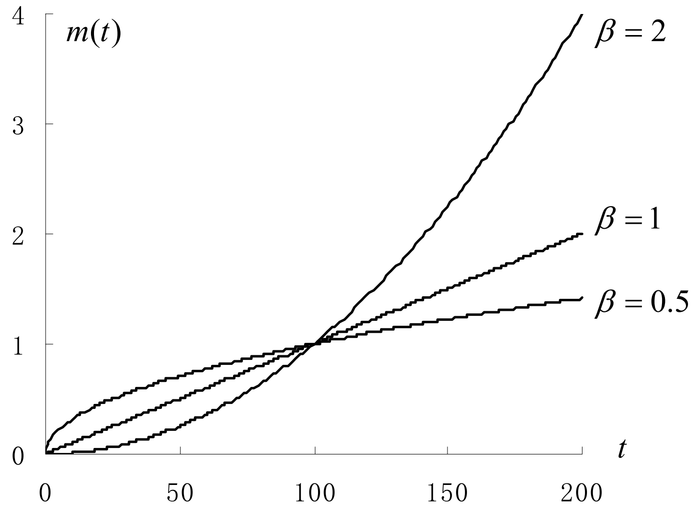


Fig. 2.7. Mean value functions of Duane NHPP models.

The failure intensity function, $\lambda(t)$, can thus be derived as

$$\lambda(t) = \frac{d}{dt} m(t) = \frac{\beta}{\alpha}\left(\frac{t}{\alpha}\right)^{\beta-1}, \quad \alpha > 0, \ \beta > 0$$

One of the most important advantages of the Duane model is that if we plot the cumulative number of failure versus the cumulative testing time on a log-log-scale, the plotted points tends to be close to a straight line if the model is valid. This can be seen from the fact that the relation between $m(t)$ and t can be rewritten as

$$\ln m(t) = -\beta \ln \alpha + \beta \ln t = a + b \ln t$$

where $a = -\beta \ln \alpha$ and $b = \beta$. Hence, $\ln m(t)$ is a linear function of $\ln t$ and due to this linear relation, the parameters α and β may be estimated graphically and the model validity can easily be verified.

The Duane model gives an infinite failure intensity at time zero. Littlewood (1984) proposed a *modified Duane model* with the mean value function

$$m(t) = k\left[1 - \left(\frac{\alpha}{\alpha + t}\right)^{\beta}\right], \ \alpha > 0, \ \beta > 0, \ k > 0$$

The parameter k can be interpreted as the number of faults eventually to be detected.

Log-power model

Xie and Zhao (1993) presented a log-power model. The mean value function of this model can be written as

$$m(t) = a \ln^{b}(1 + t); \ a, b > 0, \ t \geq 0 \qquad (2.29)$$

This model has shown to be useful for software reliability analysis as it is a pure reliability growth model. It is also easy to use due to its graphical interpretation. The plot of the cumulative number of failures at time t against t+1 will tend to be a straight line on a log-loglog scale if the failures follow the log-power model. This can be seen from the following relationship

$$\ln m(t) = \ln a + b \ln \ln(1 + t)$$

The slope of the fitted line gives an estimation of b and its intercept on the vertical axis gives an estimation of $\ln a$.

The above graphical approach makes it easy to validate the model and carry out the estimation of the parameters. This approach is also called as “first model validation and then parameter estimation”, see for details in Xie and Zhao (1993).

The failure intensity function of the log-power model can be obtained as

$$\lambda(t) = \frac{ab\ln^{b-1}(1+t)}{1+t}, \quad t \geq 0 \tag{2.30}$$

The failure intensity function is interesting from a practical point of view. The log-power model is able to analyze both the case of strictly decreasing failure intensity and the case of increasing-then-decreasing failure intensity function. For example, if $b \leq 1$, then $\lambda(t)$ of the above equation is a monotonic decreasing function of *t*; Otherwise given $b > 1$, $\lambda(t)$ is increasing if $0 \leq t < \exp(b-1)$ and decreasing if $t \geq \exp(b-1)$.

The estimation of the parameters *a* and *b* is simple. Suppose total *n* failures are detected during the a testing period $(0,T]$ and the times to failures are ordered by $0 < t_1 < t_2 < \cdots < t_n \leq T$. The maximum likelihood estimation of *a* and *b* is then given by:

$$\hat{b} = \frac{n}{n\ln\ln(1+T) - \sum_{i=1}^{n}\ln\ln(1+t_i)}$$

and

$$\hat{a} = \frac{n}{\ln^{\hat{b}}(1+T)}$$

They can thus be simply calculated without numerical procedures.

Musa-Okumoto model

Musa and Okumoto (1984) is another model for infinite failures. This NHPP model is also called the logarithmic Poisson model. The mean value function is

$$m(t) = a\ln(1+bt),\ t > 0 \tag{2.31}$$

This model can deal with the case that faults with larger size are found earlier. The failure intensity function is derived as

$$\lambda(t) = \frac{ab}{1+bt}$$

Given a set of failure time data $\{t_i, i = 1,2,...,n\}$, the maximum likelihood estimates of the parameters are the solutions of the following equations:

$$\begin{cases} \hat{a} = \dfrac{n}{\ln(1+\hat{b}t_n)} \\ \dfrac{1}{\hat{b}}\displaystyle\sum_{i=1}^{n}\dfrac{1}{1+\hat{b}t_i} - \dfrac{nt_n}{(1+\hat{b}t_n)\ln(1+\hat{b}t_n)} = 0 \end{cases} \tag{2.32}$$

These equations have to be solved numerically.

2.2.4. Other software reliability models

Software reliability is an important research area that has thousands of papers. Some books that comprehensively review them are Xie (1991), Lyu (1996), Musa (1998) and Pham (2000).

Other than Markov models discussed in this chapter, Limnios (1997) analyzed the dependability of semi-Markov systems with finite state space based on algebraic calculus within a convolution algebra. Tokuno and Yamada (2000) constructed a

Markov model, which related the failure and restoration characteristics of the software system with the cumulative number of corrected faults, and also considered the imperfect debugging process together with the time-dependent property. Becker *et al.* (2000) presented a semi-Markov model for software reliability allowing for inhomogenities with respect to process time. Rajgopal and Mazumdar (2002) also presented a Markov model for the transfer of control between different software modules.

For the NHPP models, Yamada and Osaki (1985b) summarized some existing software reliability growth models. Recently, some other NHPP models have been further developed. For example, Kuo *et al.* (2001) proposed a scheme for constructing software reliability growth models based on a NHPP model. The main focus is to provide an efficient parametric decomposition method for software reliability modeling. Huang *et al.* (2003) further described how several existing software reliability growth models based on NHPP can be comprehensively derived by applying the concept of weighted arithmetic, weighted geometric, or weighted harmonic mean. Pham (2003) recently presented studies in software reliability that includes NHPP software reliability models, NHPP models with environmental factors, and cost models.

Although the Markov and NHPP models are widely used in software reliability, some other models and tools might be also useful. Miller (1986) introduced "Order Statistic" models in studying the software reliability, which can also be found in the later research of Block *et al.* (1987), Kaufman (1996), Aki and Hirano (1996) etc. Xie *et al.* (1998) described a double exponential smoothing technique to predict software failures. Helander *et al.* (1998) presented planning models for distributing development effort among software components to facilitate cost-effective progress toward a system reliability goal. Trivedi (2002) presented a tool called SREPT that allows users to analyze the effect of non-zero debug times to reflect more realistic scenarios. Littlewood *et al.* (2003) used the Bayesian inference to estimate the reliability of diverse fault-tolerant software-based systems.

2.3. Models in Integrated Software and Hardware Systems

A computing system usually integrates both software and hardware, and software cannot work without hardware's support. Hence, system reliability should be analyzed by integrating both software and hardware influences.

This chapter presents some reliability models on the system level in which the reliability analysis considers both software and hardware failures. First, a single processor integrating software and hardware is studied. Second, modular system reliability is discussed. Following that, Markov models for clustered computing system are presented. Then, a unified model that integrates NHPP software model into the Markov hardware model is shown. Finally, some other models developed for the integrated software and hardware systems are briefly reviewed.

2.3.1. Single-processor model

The simplest case for the integrated software and hardware system is to view it as a single processor which can be generally separated in two subsystems: software and hardware subsystems. The software subsystem and hardware subsystem are considered as two different blackboxes, although integrated together, due to their distinct properties. Considering such type of system, Goel and Soenjoto (1981) presented one of the first Markov models. The assumptions of the model are listed below:

1) A computing processor consists of a hardware subsystem and a software subsystem. The faults in the software subsystem are independent from one another and each has a failure occurrence rate of λ.
2) Failures of hardware subsystem are also independent and have a failure occurrence rate of λ_h.
3) The time to remove a software fault, when there are *i* such faults in the system follows an exponential distribution with parameter μ_i.
4) The time to remove the cause of a hardware failure also follows an exponential distribution with parameter μ_h.
5) Failures and repairs of the hardware subsystem are independent of both the failures and repairs of the software subsystem.
6) At most one software fault is removed and no new software faults are introduced during the fault correction stage.
7) When the system is inoperative due to the occurrence of a software failure, the fault causing the failure is corrected with probability p_s. Also, $q_s = 1 - p_s$, is the probability of imperfect repair of software.
8) After the occurrence of a hardware failure, the hardware subsystem is recovered with probability, p_h and $q_h = 1 - p_h$ is the probability for the hardware still

staying at the failed state after the repair.

Let $X(t)$ denote the state of the system at time *t* and '$X(t) = i$', *i*=0,1,…,*N*, implies that the system is operational while there are *i* remaining software faults,. Here *N* is the initial number of software faults. Also, '$X(t) = i_s$', $i_s = 1_s, 2_s, ..., N_s$, implies that the system is down for repair of software with *i* remaining software faults at the time of failure. Similarly, '$X(t) = i_h$', $i_h = 1_h, 2_h, ..., N_h$, implies the system is down for repair of hardware with *i* remaining software faults at the time of failure. The Markov chain is shown in Fig. 2.8.

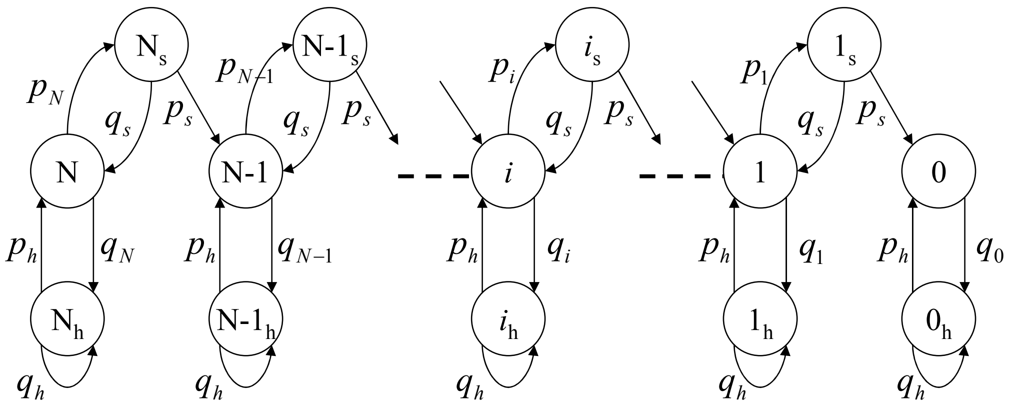


Fig. 2.8. Markov chain for the transitions between states of $X(t)$.

Suppose that the system is at state *i* (an operational state containing *i* software faults), *i*=1,2,…,*N*. The system may fail due to the software failure with probability p_i to state i_s and due to the hardware failure with the probability q_i to state i_h. At state i_s, debugging process is undertaken to remove the fault that causes the software failure. With probability p_s, the software fault is successfully removed and the system goes to state *i*-1. Otherwise with probability q_s, the fault is not removed and the software is only restarted at state i. For state i_h, maintenance personnel will try to recover the hardware failure and it has a probability p_h to return to the operational state *i* and probability q_h to remain at the failure state i_h. After the software is fault-free, i.e. at the state 0, the system reduces to a hardware system subject to hardware failures only.

Then, the basic equations describing the stochastic process as a CTMC can be formulated, see for details in Goel and Soenjoto (1981). The solutions are used to derive some system-performance measures, such as time to a specified number of software faults, system operational probabilities, system reliability and availability, and expected number of software, hardware and total failures by time *t*.

2.3.2. Modular system model

Similar to the case of modular software presented in the previous chapter, integrated software and hardware systems can also be decomposed into a finite number of modules. Markov models can also be used in analyzing such modular systems as shown below.

Siegrist (1988) might be one of the first models using Markov processes to analyze modular software/hardware systems. He assumed that the control of a system is transferred among modules according to a Markov process. Each module has an associated reliability which gives the probability that the module will operate correctly when called and will transfer control successfully when finished. The system will eventually either fail or complete its task successfully so that to enter a terminal state.

The modules (or states) of the system is denoted by *i* (*i*=1,2,…,*n*). Usually, state

1 is designated as the initial state. The ideal (failure free) system is described by a Markov chain with state space $\{1,2,\ldots,n\}$ and transition matrix **P**. That is, P_{ij} is the conditional probability that the next state will be *j* given that the current state is *i*. The reliability of state *i*, denoted by R_i, is the probability that state *i* will function correctly when called and will transfer control successfully when finished. Equivalently, the system will fail with probability $1-R_i$ each time state *i* is entered. The imperfect system is modeled by adding an absorbing state *F* (failure state) and modifying the transition probabilities appropriately. Specifically, the imperfect system is described by a Markov chain with state space $\{1,2,\ldots,n,F\}$ and transition matrix $\hat{P}$ given by

$$\begin{aligned} &\hat{P}_{ij} = R_i P_{ij}, \quad \text{for } i, j=1,\ldots,n \\ &\hat{P}_{iF} = 1 - R_i, \quad \text{for } i=1,\ldots,n \\ &\hat{P}_{FF} = 1 \end{aligned} \tag{2.33}$$

Suppose that $R_i < 1$ for each *i* and hence each of the states $1,2,\ldots,n$ eventually leads to the absorbing state *F*. Note that the dynamics of the imperfect system are completely described by the state reliability function ***R*** and the transition matrix ***P*** since this description is equivalent to specifying the transition matrix $\hat{P}$ of the imperfect system. Then, based on the Markov model, Siegrist (1988) further presented the expected number of transitions until failure as the measure of system reliability.

2.3.3. Clustered system model

Introduction

Traditionally, highly reliable systems have employed proprietary fault tolerant software and hardware, implemented with tightly coupled replicated processors and programs. Through very effective in providing high levels of reliability, proprietary fault tolerant systems are expensive to develop and usually cannot keep pace with the computing industry technology curve, see e.g. Mendiratta (1998).

Clustered computing systems uses commercially available computers networked in a loosely-coupled fashion. It can provide high levels of reliability if appropriate levels of fault detection and recovery software are implemented in the middleware (an application layers). The application, therefore, can be made as reliable as the user requires and it is constrained only by the upper bounds on reliability imposed by the architecture, performance and cost considerations.

A cluster is a collection of computers in which any member of the cluster is capable of supporting the processing functions of any other member. A clustered computing system has a redundant $n+k$ configuration, where n processing nodes are actively processing the application and k processing nodes are in a standby state, serving as spares. In the event of a failure of an active node, the application that was running on the failed node is moved to one of the standby nodes.

The simplest cluster system is one *active* and one *standby*, in which one node is actively processing the application and the other node is in a standby state. Other common cluster systems include *simplex* (one active node, no spare), n+1 (n active nodes, 1 spare), and n+0 (all n active nodes). In a system with n active nodes, the applications from the failed node are redistributed among the other active nodes using a pre-specified algorithm.

Consider a general clustered computing system with *n* active processors and *k* spares, see e.g. Mendiratta (1998). In this system, there is a Power Dog (PD) attached to each processor that can *power cycle* or *power down* the processor, and a Watch Dog (WD) with connections to each processor that monitors behavior from each processor and initiate *failover* if it detects a processor failure. Then, the failover information is transferred to a switching system (SS) that can turn on the Power Dog of the standby processors to replace the failed ones. The block diagram for this clustered system architecture is shown in Fig. 2.9 and represents the system to be modeled.

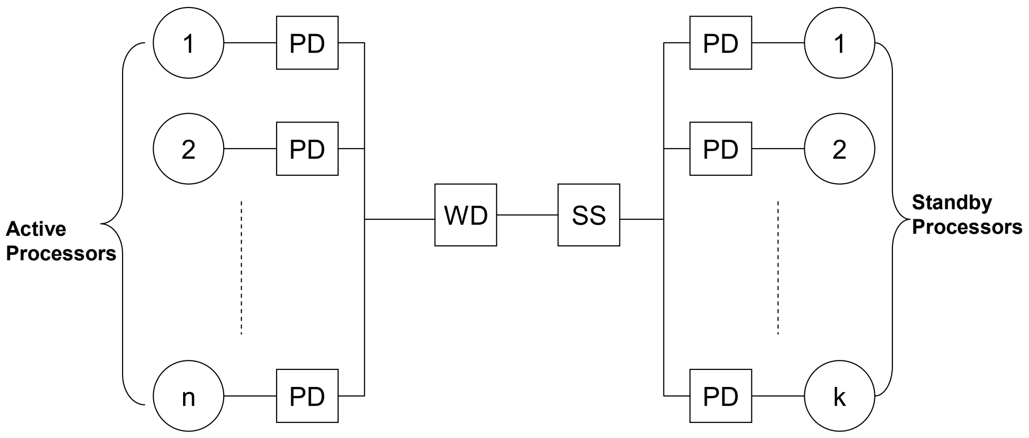
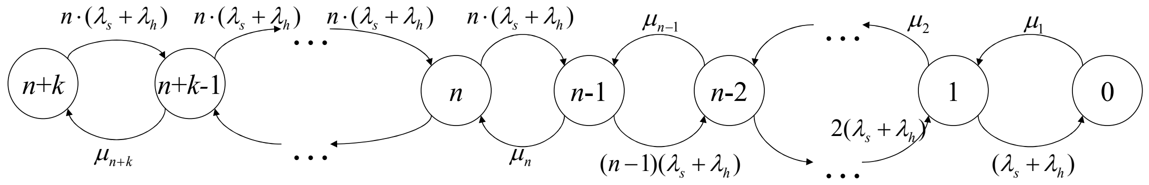


Fig. 2.9. A general architecture of $n+k$ clustered computing systems.

Markov Modeling

For each processor, there are two types of failures: software and hardware failures. Suppose the failure rate for software is λ_s and for hardware λ_h. If a system is repairable, the failed processor can be recovered with a repair rate μ_i from state *i*-1 back to state *i*. The Markov model is built as the CTMC of Fig. 2.10.



$n\cdot(\lambda_s+\lambda_h)$
$n\cdot(\lambda_s+\lambda_h)$
$n\cdot(\lambda_s+\lambda_h)$
$n\cdot(\lambda_s+\lambda_h)$
μ_{n-1}
μ_2
μ_1
$n+k$
$n+k$-1
n
n-1
n-2
1
0
$2(\lambda_s+\lambda_h)$
μ_{n+k}
μ_n
$(n-1)(\lambda_s+\lambda_h)$
$(\lambda_s+\lambda_h)$

Fig. 2.10. CTMC for repairable clustered systems.

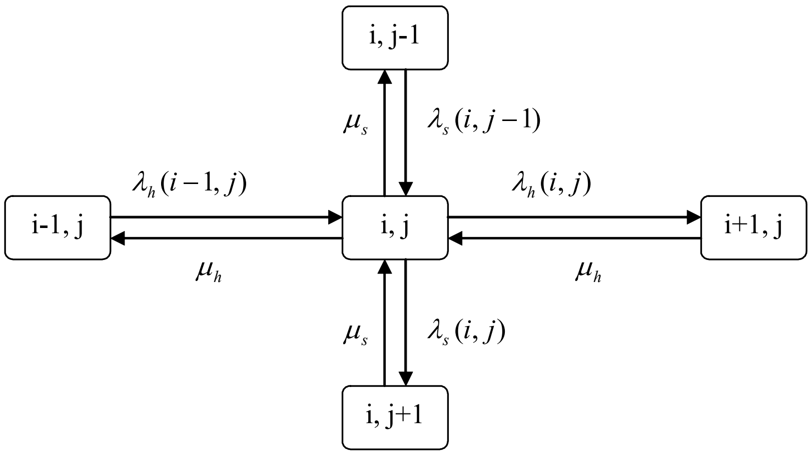
In the above model for repairable clusters, μ_i is the expected system repair rate no matter whether the failed processors are caused by software failures or hardware failures. Actually, the rate for repairing software failure should be different from that for repairing hardware failure.

Let μ_s be the rate to repair one failed processor caused by software failure and μ_h by hardware failure. Then part of the CTMC can be depicted by the Fig. 2.11. In the figure,

$$\lambda_s(i,j)=\begin{cases} n\lambda_s & \text{if } i+j\le k \\ (n-i-j)\lambda_s & \text{if } i+j>k \end{cases}$$

and

$$\lambda_h(i,j)=\begin{cases} n\lambda_h & \text{if } i+j\le k \\ (n-i-j)\lambda_h & \text{if } i+j>k \end{cases}$$



State(i,j): i hardware down, j software down (on different processors)

Fig. 2.11. CTMC for repairable cluster with different software/hardware repair rate.

The corresponding Chapman-Kolmogorov differential equation for the probability that the system is in the state (i, j) at time t is, for $i, j \neq 0, n+k;\ i+j \leq n+k-1$,

$$\begin{aligned} P'_{i,j}(t) &= \mu_h P_{i+1,j}(t) + \lambda_h(i-1, j)P_{i-1,j}(t) + \lambda_s(i, j-1)P_{i,j-1}(t) \\ &+ \mu_s P_{i,j+1}(t) - [\mu_s + \lambda_h(i, j) + \lambda_s(i, j) + \mu_h]P_{i,j}(t) \end{aligned} \tag{2.34}$$

The initial conditions are

$$P_{0,0}(0) = 1 \text{ and } P_{i,j}(0) = 0, \ for\ i, j \neq 0 \tag{2.35}$$

The boundary conditions are:

$$\begin{aligned} &P'_{0,0}(t) = \mu_h P_{1,0}(t) + \mu_s P_{0,1}(t) - n(\lambda_s + \lambda_h)P_{0,0}(t) \\ &P'_{0,j}(t) = \mu_h P_{1,j}(t) + \lambda_s(0, j-1)P_{0,j-1}(t) + \mu_s P_{0,j+1}(t) \\ &\qquad -[\mu_h + \mu_s + \lambda_s(0, j) + \lambda_h(0, j)]P_{0,j}(t) \qquad \text{for } j = 1,2,...,n+k-1 \\ &P'_{i,0}(t) = \mu_h P_{i+1,0}(t) + \lambda_h(i-1,0)P_{i-1,0}(t) + \mu_s P_{i,1}(t) \\ &\qquad -[\mu_h + \mu_s + \lambda_s(i,0) + \lambda_h(i,0)]P_{i,0}(t) \qquad \text{for } i = 1,2,...,n+k-1 \\ &P'_{i,j}(t) = \lambda_h(i-1, j)P_{i-1,j}(t) + \lambda_s(i, j-1)P_{i,j-1}(t) \\ &\qquad -(\mu_s + \mu_h)P_{i,j}(t) \qquad \text{for } i+j = n+k; 0 < i, j < n+k \\ &P'_{n+k,0}(t) = \lambda_h P_{n+k-1,0}(t) - \mu_h P_{n+k,0}(t) \end{aligned} \tag{2.36}$$

and

$$P_{0,n+k}^{'}(t) = \lambda_s(t)P_{0,n+k-1}(t) - \mu_s P_{0,n+k}(t)$$

The above equations can be numerically solved by certain computing programs, and then the system availability for the *n*+*k* clustered system can be calculated by

$$A(t) = \sum_{i+j<n+k} P_{i,j}(t) \tag{2.37}$$

2.3.4. A unified NHPP Markov model

In order to incorporate the NHPP software reliability model into the Markov hardware reliability model, Welke *et al.* (1995) developed a unified NHPP Markov model. The unified model is accomplished by determining a transition probability for a software failure and then incorporating the software failure transitions into the hardware reliability model. Based on this unified model, the differential equations can be easily established and solved despite the time-varying software failure rates.

The basic assumptions of this unified model are listed below:

1) Software failures are described by a general NHPP model, with the probability function

$$P(n,t) = \Pr\{N(t)=n\} = \frac{[m(t)]^n}{n!}\exp\{-m(t)\},\ n=0,1,2,\ldots \tag{2.38}$$

where $m(t)$ is the mean value function and *n* is the number of failures occurring up to time *t*.

2) The times between hardware failures are exponentially distributed random variable.

For more details, see Welke *et al.* (1995). Based on the above equations, the differential equations can be obtained and solved as usual.

2.3.5. Other models for integrated software/hardware systems

Similar to the single-processor model presented in this section, Hecht and Hecht (1986) also studied the reliability in system context considering both software and hardware. Fryer (1985) implemented the fault tree analysis in analyzing the reliability of combined software/hardware systems, which determines how component failures can contribute to system failure. Sumita and Masuda (1986) developed a combined hardware/software reliability model where both lifetimes and repair times of software and hardware subsystems are considered together. Kim and Welch (1989) examined the concept of distributed execution of recovery blocks as an approach for uniform treatment of hardware and software faults. Keene and Lane (1992) reviewed the similarities and differences between hardware, software and system reliability. Recently, Pukite and Pukite (1998) summarized some simple models for the reliability analysis of the hardware and software system.

For the clustered systems, Laprie and Kanoun (1992) presented Markov models for analyzing the system availability. Later, Dugan and Lyu (1995) discussed the modeling and analysis of three major architectures of the clustered system containing multiple versions of software/hardware, and they combined fault tree analysis techniques and Markov modeling techniques to incorporate transient and permanent hardware faults as well as unrelated and related software faults. Later, Lyu and Mendiratta (1999) studied the reliability modeling and analysis of the clustered system by defining the hardware, operating system, and application software reliability techniques that need to be implemented to achieve different levels of reliability and comparable degrees of data consistency.

Recently, Zhang and Horigome (2001) studied the availability and reliability on the system level considering the time-varying failures that are dependent among the software/hardware components. Lai *et al.* (2002) studied the reliability of the distributed software/hardware systems, where Markov models were implemented by assuming that the software failure rate is decreasing while the hardware has a constant failure rate. Dai *et al.* (2003a) further studied the reliability and availability of distributed services which combined both software program failures and hardware network failures altogether.

CHAPTER 3

PARALLEL HOMOGENEOUS DISTRIBUTED SYSTEM RELIABILITY

Parallel homogeneous distributed systems are widely used in many areas. This chapter models and analyzes the reliability of such systems combining both software and hardware failures. Section 3.1 constructs a Markov model to analyze this type of systems assuming the debugging process is perfect, and then Section 3.2 extends the model to include the condition of imperfect debugging. Furthermore, Section 3.3 makes thorough analysis of cost for the systems and presents an optimization model to determine the number of the redundancies.

3.1. Models with Perfect Debugging Process

3.1.1. Introduction

The distributed computing systems have gained in popularity due to low-cost processors in the recent years. A distributed system is composed of several hosts connected to a network where computing functions are shared among the hosts. It provides many advantages over centralized systems, including high throughput, cost/performance benefits, and potential for enhanced reliability. A typical application on distributed systems is distributed software of which identical copies run on each host. Examples of such applications can be found in communication protocols, networking software, and distributed database management systems, etc.

System availability is a major concern in the distributed systems. It represents the percentage of time the system is available to users. Availability of distributed systems in terms of hardware can be obtained from conventional reliability theories. A more interesting measure is the availability of the whole software/hardware system. There are some related studies. Goel and Soejoto (1982) first considered the behavior of combined software and hardware system. A generalized model is also proposed in Sumita and Masuda (1986). Goyal and Lavenburg (1987) dealt with the availability issue. Some other related references are Laprie and Kanoun (1992), Garg *et al.* (1999), Trivedi (2001), Liu *et al.* (2002) and Chen *et al.* (2002).

In fact, it is difficult to evaluate system availability of combined software/hardware systems, as explained by Lin *et al.* (1999), even for simple systems. Some models describing system availability of single-host based software/hardware systems with only one computer are presented in Goel and Soejoto (1982) and Sumita and Masuda (1986). We will emphasize on this topic in this section by extending some of their results.

A typical kind of application on distributed systems has a homogeneously distributed software/hardware structure. The physical system is assumed to contain N software subsystems (SW1-SWN) running on N hosts (HW1-HWN) as depicted in Fig.

3.1.

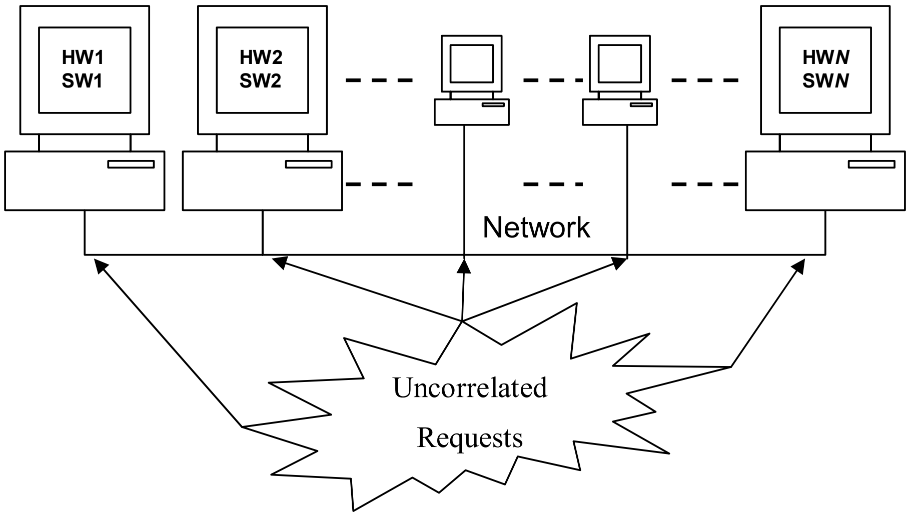


Fig. 3.1. A general homogeneous distributed software/hardware system.

That is, identical copies of distributed application software run on the same type of hosts, called Homogeneous Distributed Software/Hardware System. This system may be implemented to provide services for uncorrelated random requests of customers.

In this system, the software is usually improved during the testing phase. Since the system considers combined software and hardware failures as well as maintenance process, its reliability cannot be simply estimated by the above analytical methods for computing the distributed program reliability. The availability models and analyses of the homogeneous distributed software/hardware system are studied here.

3.1.2. Availability model

Actually, homogeneous distributed software/hardware system is a type of cluster system, which is a collection of computers in which any member of the cluster is capable of supporting the processing functions of any other member Mendiratta (1998) and Lyu and Mendiratta (1999). A cluster has a redundant n+k configuration, where n processing nodes are necessary and k processing nodes are in spare state, serving as backup. In this subsection, our model is a cluster of N homogeneous hosts that are working in parallel. This means that if all of the N hosts failed, the system fails. Otherwise whenever one or more hosts can work, the system is still working.

The following are the assumptions concerning this system:

(a) All the hosts have the same hardware failure rate λ_h arising from an exponential distribution.

(b) Each of the hosts runs a copy of the same software with a failure rate function $\lambda_s(t)$ of a given software model.

(c) Both the software and hardware have only two states, up (working state) and down (malfunctioning state), which means all the failures of software or hardware are crash failures.

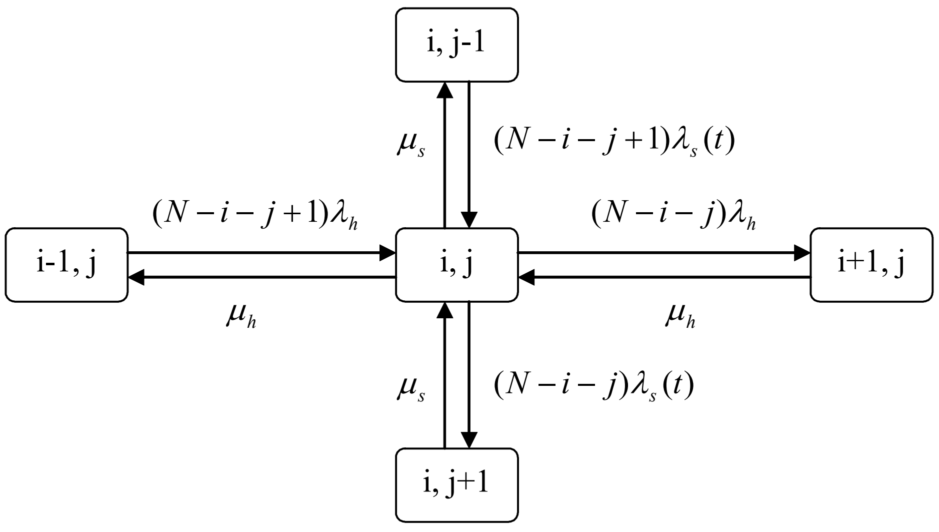
(d) There are maintenance personnel to repair the system upon software or hardware malfunction. The repair time has an exponential distribution with parameter μ_s for software and parameter μ_h for hardware, respectively.

(e) All the failures involved (either software or hardware) are mutually independent.

(f) No two or more failures (either software or hardware) occur at the same time.

There are some real cases of homogeneously distributed software/hardware system in which all the hosts can work independently for random/unknown request. Such applications can also be found in telephone switching system and bank system etc. Most homogeneous distributed software/hardware systems that work independently under the case of uncorrelated random requests can implement our models.

Systems in practice can be complex and usually we have a multi-host situation. Lai *et al.* (2002) implemented a Markov process to model it. Fig. 3.2 illustrates a partial system state transition of the Markov process, in which (*i*, *j*) is the state when *i* hosts suffer hardware failures and *j* hosts suffer software failures.



State(i,j): i hw down, j sw (on different hosts) down

Fig. 3.2. The partial state transition graph for the *N*-host system.

The corresponding Kolmogorov differential equation for the probability that the system is in the state (*i,j*) at time *t* is, for $i, j \neq 0, N;\ i+j \leq N-1$,

$$\begin{aligned} P^{'}_{i,j}(t) = \mu_h P_{i+1,j}(t) + (N-i-j+1)\lambda_h P_{i-1,j}(t) \\ +(N-i-j+1)\lambda_s(t) P_{i,j-1}(t) + \mu_s P_{i,j+1}(t) - x_{i,j} P_{i,j}(t) \end{aligned} \tag{3.1}$$

where

$$x_{i,j} = \mu_s + (N-i-j)\lambda_h + (N-i-j)\lambda_s(t) + \mu_h \tag{3.2}$$

The initial conditions are

$$P_{0,0}(0) = 1 \text{ and } P_{i,j}(0) = 0, \text{ for } i, j \neq 0$$

The equations for the boundary states are:

$$\begin{aligned}
P^{'}_{0,0}(t) &= \mu_h P_{1,0}(t) + \mu_s P_{0,1}(t) - N[\lambda_s(t) + \lambda_h] P_{0,0}(t) \\
P^{'}_{0,j}(t) &= \mu_h P_{1,j}(t) + (N-j+1)\lambda_s(t) P_{0,j-1}(t) + \mu_s P_{0,j+1}(t) \\
&\quad -\{\mu_h + \mu_s + (N-j)[\lambda_s(t) + \lambda_h]\} P_{0,j}(t) \text{ for } j = 1,2,...,N-1 \\
P^{'}_{i,0}(t) &= \mu_h P_{i+1,0}(t) + (N-i+1)\lambda_h P_{i-1,0}(t) + \mu_s P_{i,1}(t) \\
&\quad -\{\mu_h + \mu_s + (N-i)[\lambda_s(t) + \lambda_h]\} P_{i,0}(t) \text{ for } i = 1,2,...,N-1 \\
P^{'}_{i,j}(t) &= (N-i-j+1)\lambda_h P_{i-1,j}(t) + (N-i-j+1)\lambda_s(t) P_{i,j-1}(t) \\
&\quad -(\mu_s + \mu_h) P_{i,j}(t) \qquad \text{for } i + j = n + k; 0 < i, j < n + k \\
P^{'}_{N,0}(t) &= \lambda_h P_{N-1,0}(t) - \mu_h P_{N,0}(t) \\
P^{'}_{0,N}(t) &= \lambda_s(t) P_{0,N-1}(t) - \mu_s P_{0,N}(t)
\end{aligned} \tag{3.3}$$

The system availability for the *N*-host based system can be calculated by

$$A(t) = \sum_{i+j<N} P_{i,j}(t) \tag{3.4}$$

Here, we assume each copy of software suffers a failure rate of the JM model (Jelinski and Moranda, 1972), i.e.

$$\lambda_s(t) = k_t \phi$$

To solve the above differential equations, we need to know the expected number of remaining software faults (k_t). However, since k_t changes with software debugging, it is usually a function of time. We have used the following scheme for the numerical calculation, as shown by Lai *et al.* (2002). According to the JM model, the probability of software having *k* remaining faults at time *t* is

$$P(k,t)=\binom{K_0}{k}\exp(-k\phi t)\cdot[1-\exp(-\phi t)]^{K_0-k} \quad \text{for } 0\le k\le K_0 \tag{3.5}$$

Based on this equation, the expected number of remaining software faults at time *t* can be computed as

$$k_t=\sum_{k=0}^{K_0}k\cdot P(k,t)$$

The system availability can be computed using any available numerical algorithm to solve the differential equations, such as the SHARP and so on. An example using our above Markov model to analyze availability of homogeneous distributed software/hardware system is numerically illustrated below.

Example 3.1. We assume that the hardware failure rate is 0.02 and software failure rate per fault is 0.006. The repair rate for hardware is 0.1 while that for software is 0.12. Fig. 3.3 depicts the result of system availability of a triple-host system with different number of initial faults.

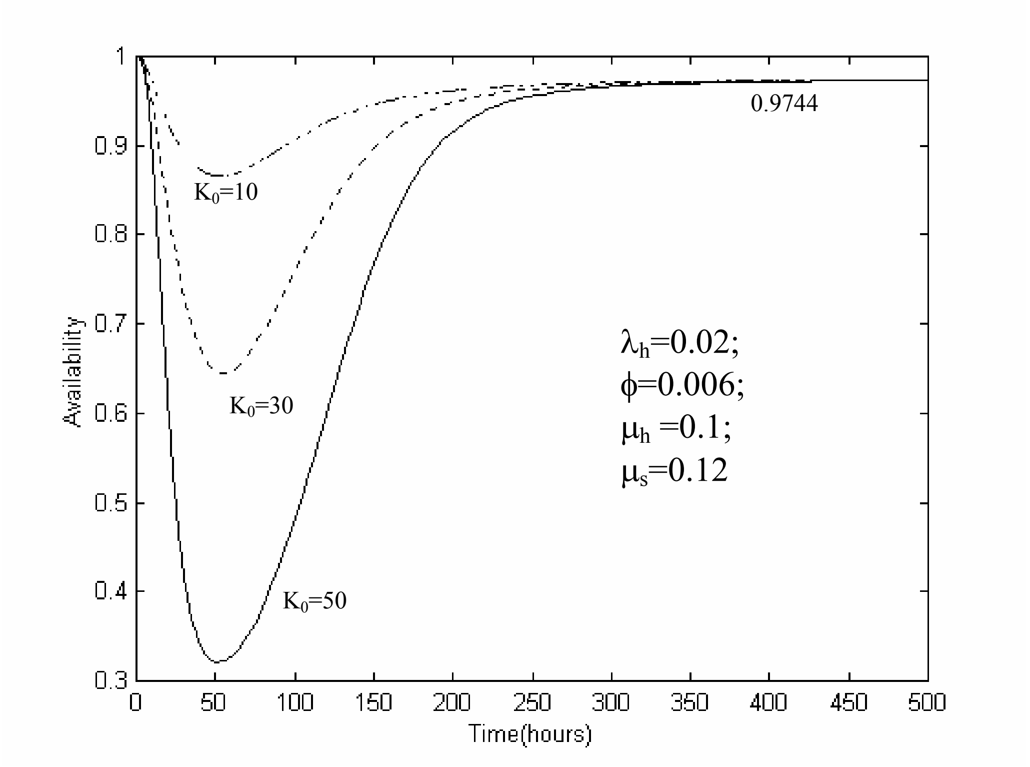


Fig. 3.3. A typical curve of the system availability function.

It can be seen from Fig. 3.3 that the system availability reaches the lowest point at an early stage. This is because a large number of faults are identified when software system testing begins. System availability starts recovering after the lowest point and approaches a certain value less than 1 asymptotically after a longer period of time. This is because identified faults are fixed and as a result software failure rate decreases.

The initial software fault number affects the system availability only at the early stage. The more the testing time passes, the less effect the K_0 has. In the end, the steady availability will be same as "fault free", no matter what the initial software fault number is.

3.2. Models with Imperfect Debugging Process

In the above section, the model assumed that the debugging process was a perfect one. However, it is possible in reality that the fault that is supposed to have been removed may cause a failure again. It may be due to the spawning of a new fault by the imperfect debugging process, see e.g. Fakhre-Zakeri and Slud (1995), Sridharan and Jayashree (1998), Pham *et al.* (1999) and Tokuno and Yamada (2000).

3.2.1. Markov modeling

The assumptions used in this imperfect debugging model are almost the same as the assumptions (*a-f*) in earlier model except add the following assumption.

(g) When a software failure occurs, repair starts with the following debugging probabilities:

The software fault content is reduced by one with probability p

The software fault content remains unchanged with probability r

The software fault content is increased by one with probability q.

This assumption is same as the birth-death process that was introduced in Kremer (1983).

Fig. 3.4 illustrates a partial system state transition, in which (i, j, k) is the state when i hosts suffer hardware failures, j hosts suffer software failures and k is the number of remaining software faults at that time. Here N is the total number of hosts in the system.

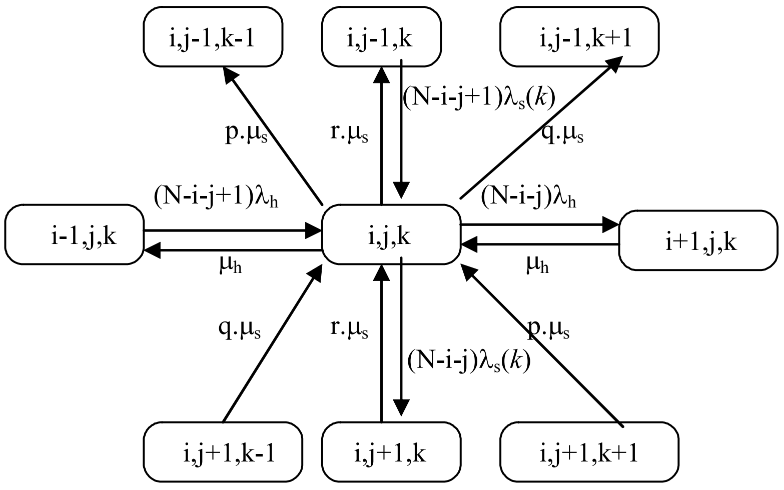


Fig. 3.4. The state transition graph for the *N*-host system.

The corresponding Chapman-Kolmogorov differential equation for the probability that the system is in the state (i, j, k) at time t can be obtained as:

$$\begin{aligned} P'_{i,j,k}(t) &= \mu_h P_{i+1,j,k}(t) + p\mu_s P_{i,j+1,k+1}(t) + r\mu_s P_{i,j+1,k}(t) + q\mu_s P_{i,j+1,k-1}(t) \\ &+ (N-i-j+1)\lambda_h P_{i-1,j,k}(t) + (N-i-j+1)\lambda_s(k) P_{i,j-1,k}(t) - A_{i,j,k} P_{i,j,k}(t) \end{aligned}$$

$$(i, j = 1,2,....,N-1; i+j \le N-1; k = 1,2...K_0 - 1) \qquad (3.6)$$

where

$$A_{i,j,k} = \mu_s + (N-i-j)[\lambda_h + \lambda_s(k)] + \mu_h$$

The boundary conditions are $(i, j = 0, N;\ i+j \le N;\ k = 0, K_0)$:

$$\begin{aligned} P'_{i,j,k}(t) &= z_1\mu_h P_{i+1,j,k}(t) + z_2 p\mu_s P_{i,j+1,k+1}(t) + z_3 r\mu_s P_{i,j+1,k}(t) \\ &+ z_4 q\mu_s P_{i,j+1,k-1}(t) + z_5(N-i-j+1)\lambda_h P_{i-1,j,k}(t) \\ &+ z_6(N-i-j+1)\lambda_s(k) P_{i,j-1,k}(t) - B_{i,j,k} P_{i,j,k}(t) \end{aligned} \qquad (3.7)$$

where

$z_1 = 0$ for $i = N$, and 1 for otherwise

$z_2 = 0$ for $j = N$ or $k = K_0$, and 1 for otherwise

$z_3 = 0$ for $j = N$, and 1 for otherwise

$z_4 = 0$ for $j = N$ or $k = 0$, and 1 for otherwise

$z_5 = 0$ for $i = 0$, and 1 for otherwise

$z_6 = 0$ for $j = 0$, and 1 for otherwise

$$B_{i,j,k} = (d_1 p + d_2 r + d_3 q)\mu_s + d_1(N-i-j)\lambda_h + d_5(N-i-j)\lambda_s(k) + d_6\mu_h$$

and in the above

$d_1 = 0$ for $j = 0$ or $k = 0$, and 1 for otherwise

$d_2 = 0$ for $j = 0$, and 1 for otherwise

$d_3 = 0$ for $j = 0$ or $k = K_0$, and 1 for otherwise

$d_4 = 0$ for $i = N$, and 1 for otherwise

$d_5 = 0$ for $j = N$, and 1 for otherwise

$d_6 = 0$ for $i = 0$, and 1 for otherwise.

Let K_0 be the initial number of faults in the software. Then the initial conditions are

$$P_{0,0,K_0}(0) = 1 \text{ and others are } 0 \tag{3.8}$$

The solutions can be obtained by solving the above equations.

The system availability at time *t* can be calculated as

$$A(t) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-i-1} \sum_{k=0}^{K_0} P_{i,j,k}(t) \tag{3.9}$$

Although those differential equations can be solved, the procedure becomes difficult when the number of hosts is large. Hence, some computing tools can be used to solve them. An example is illustrated below.

Example 3.2. In this numerical case, the software failures are assumed to follow the JM-model. For the multi-host systems with different number of hosts, the system availability functions can be obtained numerically. The curves of system availability functions for (N=2,3,4,5) are depicted in Fig. 3.5 with parameters

$$\mu_h=0.1536,\quad \mu_s=0.1331,\quad p=0.831,\quad q=0.078,$$

$$r=0.091,\quad K_0=42,\quad \phi = 0.0013 \quad \text{and } \lambda_h = 0.005.$$

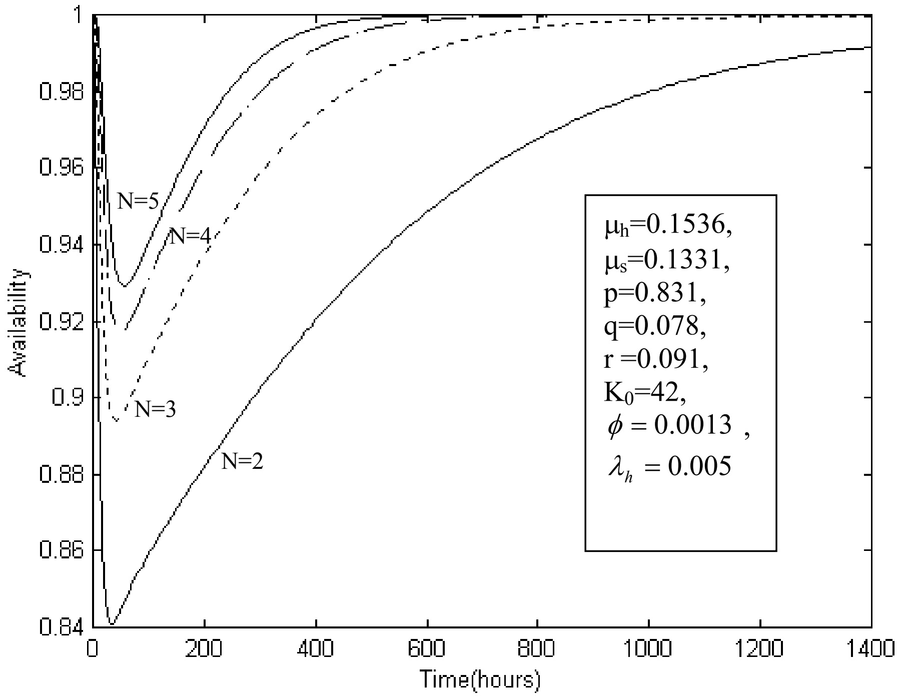


Fig. 3.5. The curves of system availability of different number of hosts.

Fig. 3.5 shows a similar trend as that of Fig. 3.3. System availability reaches the lowest point at an early stage. After that period, system availability starts recovering because identified faults are fixed and as a result software failure occurrence rate decreases.

3.2.2. Other measures on the debugging process

Besides the availability function, some other measures related to the imperfect debugging process are also important, such as the expected number of remaining/removed faults and the density function for removing the faults etc.

The expected number of remaining/removed faults at time t

The expected number of remaining/removed faults is an important measure in software quality analysis. The function of the expected number of remaining faults is denoted by $E(t)$, which can be expressed as

$$E(t)=\sum_{k=0}^{K_0} kP_k(t) \tag{3.10}$$

in which $P_k(t)$ is the distribution of *k* remaining faults is described by the probability for the remaining software faults to be *k* at time *t*. It can be expressed by

$$P_k(t)=\sum_{i=0}^{N}\sum_{j=0}^{N} P_{i,j,k}(t) \tag{3.11}$$

The expected number of removed faults at time *t* can be derived as

$$F(t) = K_0 - E(t)=K_0 - \sum_{k=0}^{K_0} kP_k(t) \tag{3.12}$$

Continued with the above example, the expected numbers of remaining/removed faults as a function of time *t* are depicted in Fig. 3.6.

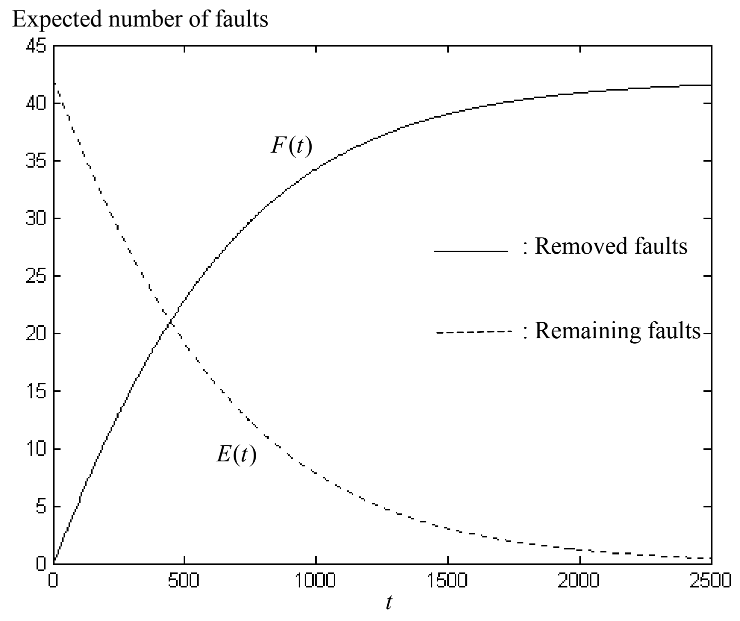


Fig. 3.6. Expected number of remaining/removed faults.

From the Fig. 3.6, it can be observed that the expected number of remaining/removed faults is a decreasing/increasing function when testing time goes by. It is because the faults are debugged during the testing period and all of the faults are expected to remove eventually and ideally.

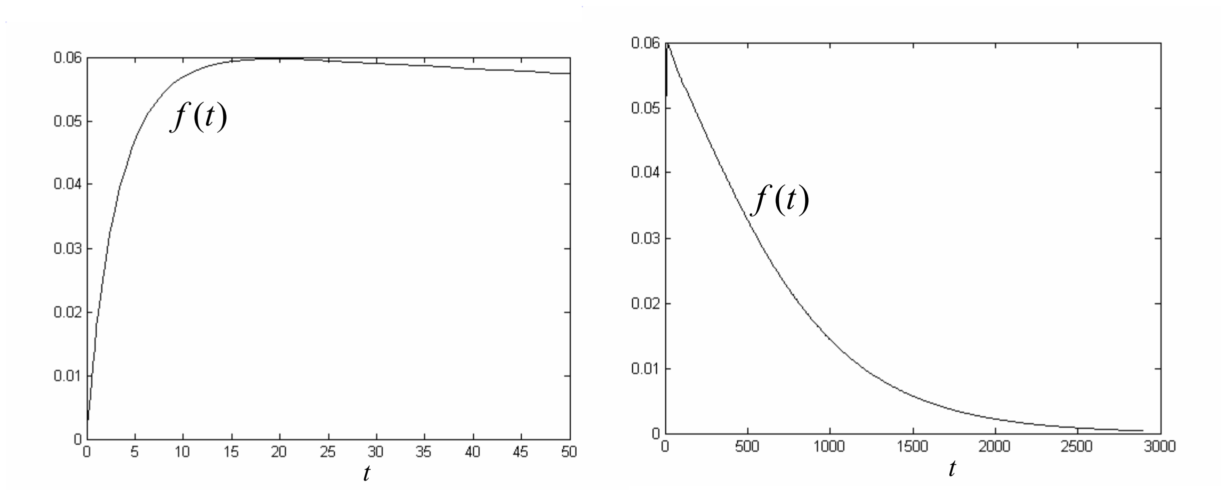
The density function of the expected number of removed faults

The density function of the expected number of removed faults is also important as it reflects the effectiveness of the imperfect debugging process. Therefore, this density function can be derived by

$$f(t)=F'(t)=-\sum_{k=0}^{K_0} kP_k'(t)=-\sum_{k=0}^{K_0}\sum_{i=0}^{N}\sum_{j=0}^{N} kP'_{i,j,k}(t) \tag{3.13}$$

Substituting $P'_{i,j,k}(t)$ into the above equation, the fault removal rate can then be numerically computed. Continued to the above example, the density function is depicted by Fig. 3.7.

At the beginning, the density function increases quickly as curve (a) in Fig. 3.7. The reason may be that the faults are being emerged at the beginning time. After the peak point, it decreases and trends to 0 as curve (b) in Fig.3.7. This property of density function to increase first and then decrease shows that the density function of removed faults for the imperfect debugging is slightly S-shaped.



(a) Initial phase **(b) Long-term trend**

Fig. 3.7. Density function of the expected number of removed faults.

3.3. Optimal Number of Redundant Hosts

An important goal in the design of the parallel homogeneous distributed systems is to achieve a high reliability or availability through some kind of redundancy (such as redundant hosts) or fault tolerance. Many systems are developed in the environment with redundant hosts. The number of redundant hosts has significant influence on the cost and system availability because it can be very costly while they are able to improve system availability easily. How to optimally design the number of redundant hosts is an important decision in the systems. The objective here in system design is to minimize the total cost based on the following cost model.

3.3.1. The cost model

In order to illustrate the relationships among the decisions and cost, an influence diagram, which provides simple graphical representations of decision situations, is displayed in Fig. 3.8. Different decision elements are shown in the influence diagram as of different shapes, see e.g., Clemen (1995 pp. 50-65).

The number of redundant hosts will affect the optimal decision of the release time. Both the number of redundant hosts and release time will affect the system availability. These three factors determine the development cost. The number of hosts also determines the cost of redundant hosts. The release time determines the rewards or penalty depending on whether the release is before or after the deadline. If the system is unavailable after release, a risk cost is incurred. Hence, the cost of redundant hosts, the development cost, reward and penalty should be considered together when deriving the total expected cost. Each cost component will be described in the following.

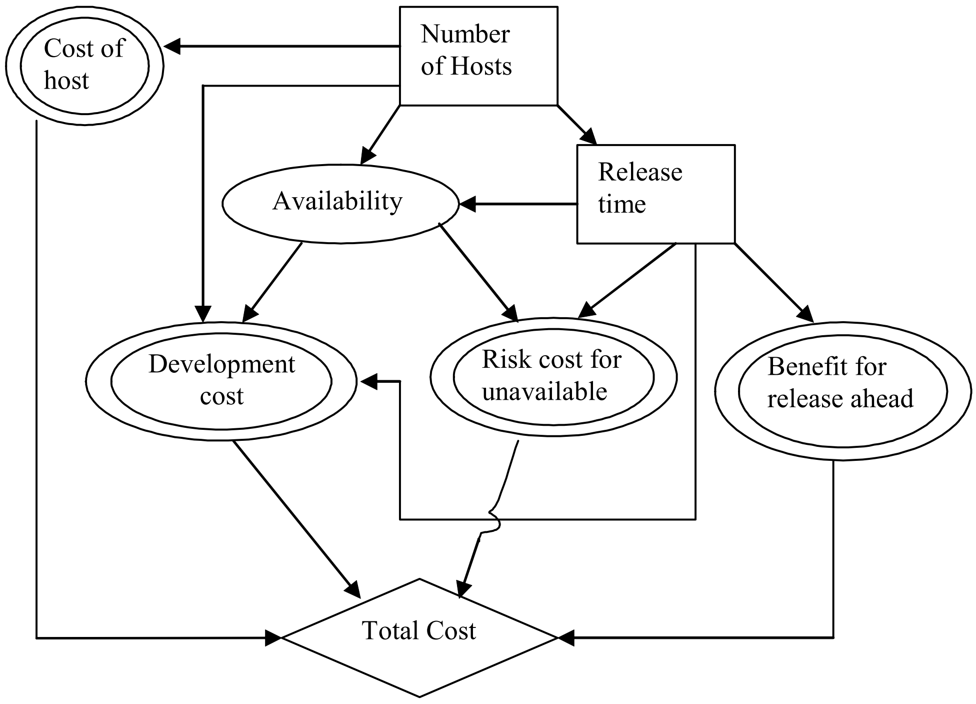


Fig. 3.8. Influence diagram for the cost affected by redundant hosts.

Cost of redundant hosts

The cost function for a multi-version fault-tolerant system can be described as a linear function to the number of versions. This is used in Laprie *et al.* (1990) originated from Boehm (1981), and we have

$$C_h(N) = a_1 N + b_1 \tag{3.14}$$

where *N* is the number of hosts, b_1 is a constant, and a_1 is defined as the expected cost per host. Here we have assumed the redundant hosts used in the system are of the same type.

Reward for early release

Usually there is a deadline for release. This is the case when the penalty cost for delay is very high and the system has to be released even if it is not as reliable, with the consequence of high maintenance and other costs. On the other hand, because of the competitive market place, there is a reward for releasing the system earlier. We assume b_2 is a constant rewarded if the system can be released in time, no matter how early the release time is and a_2 is the expected reward per unit time before the deadline. Thus, the reward function of the release time can be expressed as

$$B(t_r) = a_2(T_d - t_r) + b_2, \quad t_r \leq T_d \tag{3.15}$$

where T_d is the deadline for release, t_r is the release time so that $T_d - t_r$ is the time ahead of the schedule.

Risk cost for system being unavailable

After the system is released, there is a risk for it to be unavailable, and there are contractual consequences. This cost factor is generated by the unavailable system after releasing, termed risk cost as in Pham and Zhang (1999). Here we assume the risk cost for unavailable system is a function of system availability and release time:

$$C_r(N, t_r) = a_3 \int_{t_r}^{T_e} [1 - A_N(t)]dt \tag{3.16}$$

where t_r is the release time, T_e is the ending time for contracted maintenance after release, $A_N(t)$ is the availability function at time t for N-host system, and a_3 is the risk cost per unit time when the system is not available. In the equation above, $1 - A_N(t)$ is the probability for the system to be unavailable at time t. Hence, the integral above is the expected time for the system to be unavailable from t_r to T_e.

Development cost

Since our focus is on system integration testing with the emphasis on software testing and debugging, software development cost includes the cost occurring in testing/debugging phase to improve the software reliability. The development cost function for a single software module proposed in Kumar and Malik (1991) is

$$C_i(R_i) = H_i \exp(B_i R_i - D_i) \qquad (3.17)$$

where H_i, B_i and D_i are constants and R_i is the individual module software reliability achieved at the end of testing. These parameters are explained in Kumar and Malik (1991). Briefly, the cost is exponentially increasing to the improved reliability of a single module.

Then, the total expected cost can be expressed as

$$C(N,t_r) = C_h(N) + C_r(N,t_r) + C_t(R(t_r)) - B(t_r) \qquad (3.18)$$

3.3.2. System availability

An important problem is to obtain the system availability function for calculating the risk cost. The availability of a system is affected by both software and hardware components. The system availability model for a homogeneous distributed software/hardware system can be obtained straightforward from section 3.1. A numerical example is shown below.

Example 3.3. Suppose $K_0 = 32$ and λ=0.006, λ_h=0.01, μ_h=0.1 and μ_s=0.13, the system availability for different number of hosts can be obtained from the analysis presented in section 3.1. The results are depicted in Fig. 3.9.

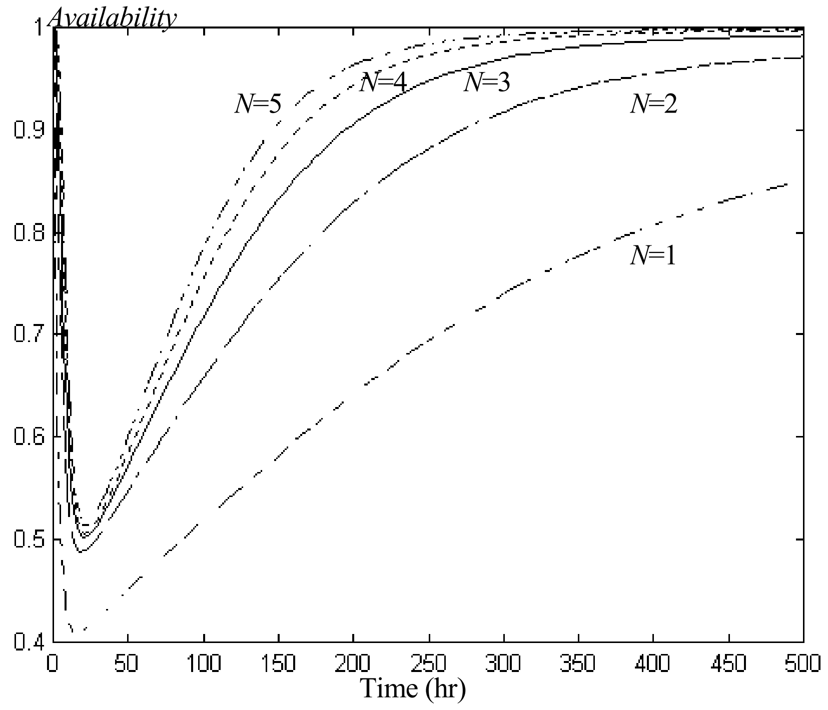


Fig. 3.9. System availability for different number of redundant hosts.

We can observe that when the number of redundant hosts increases, the system availability increases. The system availability function can be used in the optimization model which will be described in the following.

3.3.3. Optimization model and solution procedure

The optimization model is based on the cost criteria and the decision variables are the number of redundant hosts and the release time. Its objective is to minimize the expected total cost. There are three types of constraints in this decision problem. First, the customers may require a least system availability A^* after the release. Second, there is a deadline for the system to be released so the release time should be earlier than that. Finally, the customers may limit the maximum number of redundant hosts N^* due to their budget and other physical restrictions.

The optimization model is then constructed as follows.

Decision variables: N and t_r.

Objective function: Minimize $\{C(N,t_r)\}$

Subject to: $A_N(t_r) \geq A^* \geq 0$

$$0 \leq t_r \leq T_d$$

$$N = 1,2,3,\ldots,N^*$$

where A^* is the required system availability after the release, T_d is the deadline for release and N^* is the maximum number of redundant hosts allowed because of any physical such as space constraints. If there is no such constraint, we can assume a large enough value of N^* in this model. However, usually only a small number of redundant hosts will be practical.

For obtaining the optimal solution, the solving procedures are described as follows.

Step 1: Derive the system availability function of the distributed system with N redundant hosts,

Step 2: Derive each cost function and obtain the expected total cost,

Step 3: Let N take each integer value from 1 to N^* to obtain the expected total cost and save the results from $C(1,t_r)$ to $C(N^*,t_r)$, which does not break the constraints,

Step 4: For each expected total cost in $C(1,t_r)$ to $C(N^*,t_r)$, compute the optimal release time, and save the results as OpTr(1) to OpTr(N^*) , so that we can get the minimal expected total cost and save them in MinC(1) to MinC(N^*) that MinC(n)=C(n,OpTr(n)) (n=1,2,…, N^*),

Step 5: Compare the minimal total mean cost from MinC(1) to MinC(N^*) in order to select the optimum number of redundant hosts $OpN = Min(\text{MinC(n)})$ (n=1,2,…, N^*) and output the results.

The above procedure can be easily realized in Matlab or other computational programs. It is noted that usually the number of redundant hosts is not very large. A numerical example is presented to illustrate the optimization procedures.

Example 3.4. This application example is based on a telephone switching system development. Company X was awarded a contract to develop the system for a customer. After the development, testing and debugging are carried out, especially on the software systems. In this case, the hardware hosts are brought from external suppliers, but the software is developed in house and tested with the system. The management is concerned of how many redundant hosts are needed and also when the system can be released so that the total cost is minimized. For illustrative purpose, the following input values are used:

1. The customer requires the system availability to be higher than 0.88 when it is released.

2. The deadline for releasing the system is about 800 hours from now on.

3. By contract, the company will be penalized for unavailable system about $8000 per hour during the first 300 hours after release.

4. Each redundant host cost about $17600 and the other fee for all the hosts is about $1293, such as installation fee software copyright fee and etc.

5. The maximum number of redundant hosts is five.

6. If the company can release the system earlier than the deadline, there is a constant reward of $2123.7 and a cumulative reward of $31.5 per hour less than the deadline.

Based on the conditions and the assumptions given above, the values of the parameters can be obtained as

a_1 =17600, b_1 =1293, a_2 =31.5, b_2 =2123.7, a_3=8000, T_d=800 hours,

and

$T_e = 800 + 300 = 1100$ hours.

The parameters for software development cost (4.27) are assumed as H=10232, B=16, D=14. The optimization problem can be solved with the required system availability when releasing, A^*, of 0.88 and the maximum number of redundant hosts, N^*, equal to 5.

Here we assume the system is a kind of homogeneous distributed software/hardware system whose availability function is depicted by Fig. 3.8. With the values of parameters given above, we can obtain the total mean cost through Eq. (3.18) as

$$C(N,t_r)=17600N+31.5t_r+8000\int_{t_r}^{1100}[1-A_N(t)]dt+10233\exp\{16R(t_r)-14\}-26030.7$$

Finally, the total expected cost as a function of release time for different number of redundant hosts are depicted by Fig. 3.10 and the overall results are given in Table 3.1.

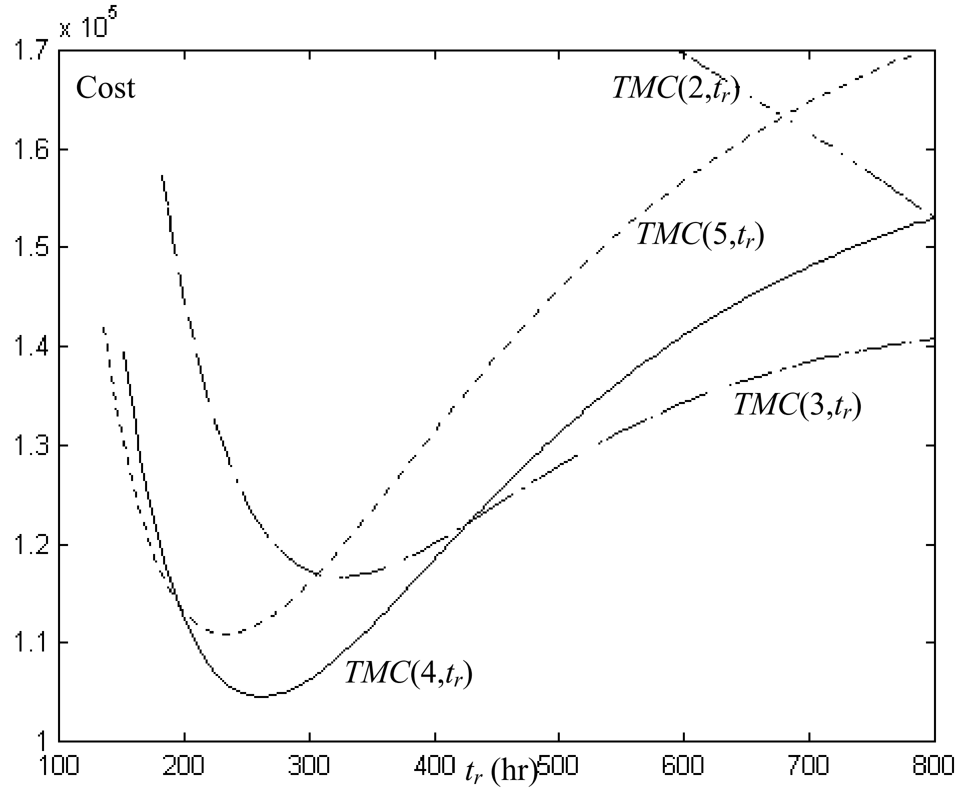


Fig. 3.10. Total mean cost *vs.* release time of different number of hosts.

Table 3.1. Numerical values of the minimum cost for different *N*.

N	1	2	3	4	5
MinC(N)	326970	153060	116690	104580	110880
OpTr(N)	800	800	324.2	261.7	232

From the Table 3.1, the global minimum cost is 104580 (Units) with the number of redundant hosts *N*=4 and the optimum release time t_r =261.7 (hrs). The optimum results indicate that there should be four redundant hosts and the system is tested for 261.7 hours.

3.4. Conclusion

This chapter studied the parallel homogeneous distributed systems with perfect and imperfect debugging process, respectively. Then, it conducted a cost analysis for the systems and presented an optimization model to determine the number of the parallel redundancies and the best release time.

However, this chapter assumed that all the processors in the systems were homogeneous, which is not always true in the distributed system analysis. Hence, the next chapter will extend it to study a kind of heterogeneous distributed system and analyze its behavior and reliability from the service point of view.

CHAPTER 4

CENTRALIZED HETEROGENEOUS DISTRIBUTED SYSTEM RELIABILITY

4.1. Introduction

Most of the computing systems can be modeled as a centralized heterogeneous distributed system, which is the same common as the parallel homogeneous distributed system discussed in the previous chapter. This type of system consists of some subsystems managed by a control center. For example of a system with Client/Server structure, every Client in the sub-distributed systems is managed by a control center of Servers.

Since the computing systems are developed to provide different services with specific objectives such as running a computer program, controlling a production process, and completing some other tasks, the service reliability of the distributed system is a key point of the QoS (Quality of Service). A definition of distributed service reliability can be the probability to successfully provide the service in a distributed environment. This is the definition that will be adopted in this chapter. The service reliability in a centralized heterogeneous distributed system is determined not only by the system availability of the control center, but also by distributed program reliabilities of the subsystems.

The system availability of the control center is of major concern because an unavailable control center will sometimes cause critical problems to a service, see e.g. Pham *et al.* (1997) and Sols and Nachlas (1995). Srinivasan and Jha (1999) described a method to determine an allocation that introduces safety into a heterogeneous distributed system and at the same time attempts to maximize its availability. There is some research on increasing system availability, see e.g. Lutfiyya *et al.* (2000). Goel and Soejoto (1981) first considered the behavior of combined software and hardware system. A generalized model is also proposed in Sumita and Masuda (1986). Markov models are also implemented to analyze the system availability, which combines both software and hardware failures and maintenance processes, see e.g. Welke *et al.* (1995) and Lai *et al.* (2002).

On the other hand, the reliability of each program in the system is also important to a service. The distributed program reliability is defined as the probability of successful execution of a program running on multiple processing elements and it needs to retrieve data files from other processing elements. Kumar *et al.* (1986) proposed a useful notion called a Minimal File Spanning Tree (MFST) and developed an algorithm also called MFST to find MFSTs within a graph. To improve the MFST algorithm, there are some further developed algorithms, see e.g. Kumar *et al.* (1988), Chen and Huang (1992), Kumar and Agrawal (1993), Chen *et al.* (1997) and Lin *et al.* (1999).

However, most of the earlier research on system availability or distributed program/system reliability cannot be simply implemented to analyze the service reliability of the centralized heterogeneous distributed systems because it is affected by many factors including system availability, software/hardware/network reliability. This chapter studies the property of centralized heterogeneous distributed system and develops a general model for the analysis. Based on the model, algorithms to obtain the service reliability of the system are also presented.

This part is organized as follows. Section 4.2 presents a model for a centralized heterogeneous distributed system (CHDS), and develops a solution algorithm for the distributed service reliability in CHDS. Then, an application example is illustrated to illustrate the procedure and the feasibility of the algorithm. Furthermore, we analyze the behavior and sensitivity of the system availability function in Section 4.3, which are important issues in the application of this type of model.

4.2. CHDS and Analysis

4.2.1. Description of the systems and services

Service of Centralized Heterogeneous Distributed System and its reliability

The structure of the Centralized Heterogeneous Distributed System is depicted by Fig. 4.1. The control center may consist of many servers. These servers support a virtual machine. The virtual machine can manage programs and data from heterogeneous subsystems through virtual nodes. The virtual nodes can mask the differences among various platforms. They are a kind of virtual executing elements, which only includes a basic unit for executing data, i.e. CPU and Memory. The entities of virtual machine and virtual nodes are supported by the software and hardware in the control center.

The heterogeneous sub-distributed systems are composed of different types of computers with various operating systems connected by different topologies of networks. These subsystems exchange data with virtual machine through System Service Provider Interface (SSPI). They are connected with virtual nodes by routers. They can cooperate to achieve a distributed service under the management of the virtual machine.

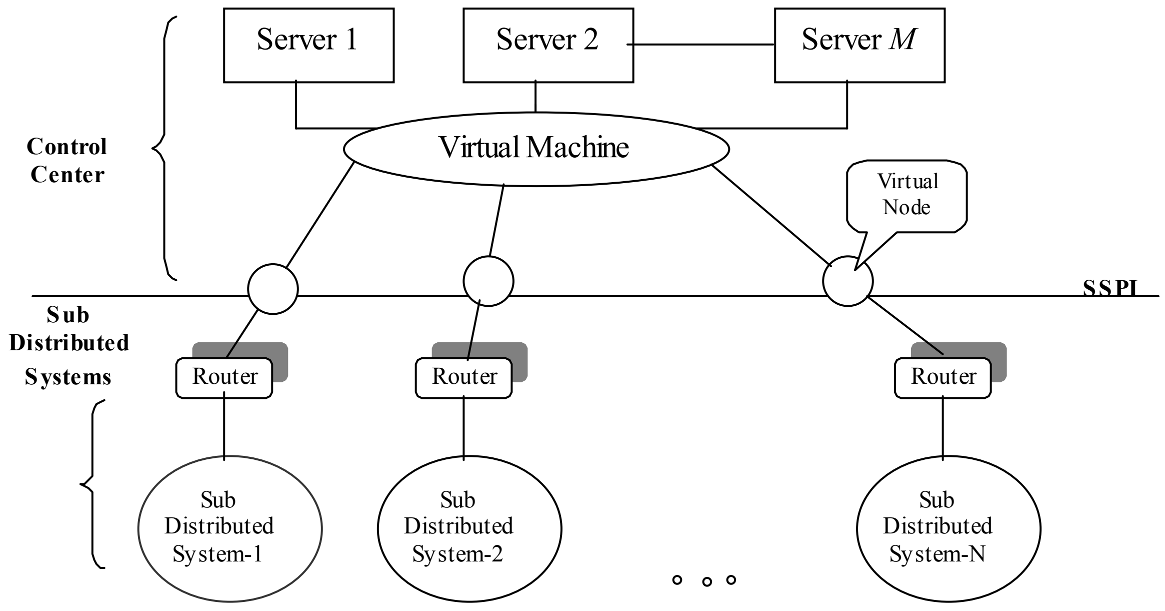


Fig. 4.1. Structure of the centralized heterogeneous distributed service system.

Most of the computing systems can be categorized as centralized heterogeneous distributed systems. For example, in modern warfare, each soldier can be considered as an element in a military system and furnished with different electrical equipments for diverse purposes. The information collected from each soldier is sent back to a control center through wireless communication channels. Then, the control center can analyze all the information and send out commands to respective soldiers. The functions of different groups of soldiers are diversified in a war (such as attacking, defending, supplying, saving etc.) so their electrical equipments should also be heterogeneous. Thus, it is a typical Centralized Heterogeneous Distributed System, as depicted by Fig. 4.2.

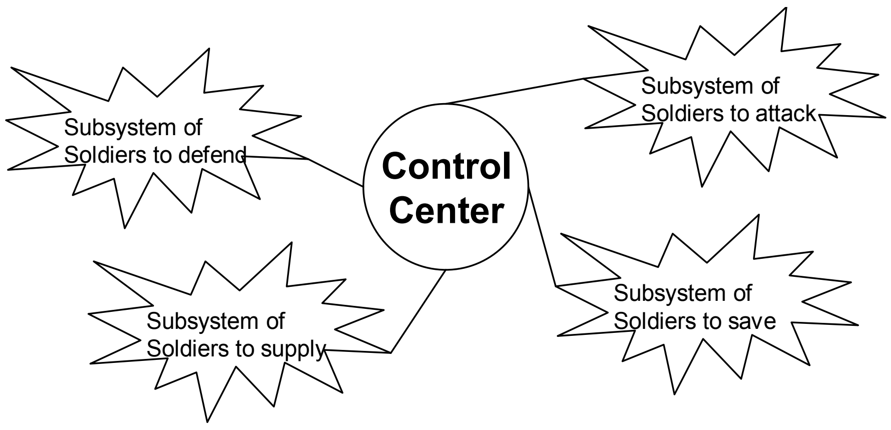

Subsystem of Soldiers to defend
Control Center
Subsystem of Soldiers to attack
Subsystem of Soldiers to supply
Subsystem of Soldiers to save

Fig. 4.2. A simple and abstracted military system.

The whole process for a service provided by a system is repeated so the reliability analysis of a distributed service is crucial for a distributed system. The distributed service reliability is defined as below.

Definition 6.5. *Distributed service reliability* is the probability for a service to be successfully achieved in a distributed computing system.

Model of distributed service reliability

In a distributed service system, a service includes various distributed programs completed by diverse computers. Some later programs might require several precedent programs to be completed. Every program requires a certain execution time. The execution of some programs might require certain input files that are saved or generated in different computers of the distributed systems. The overall distributed service reliability depends on the reliability of a program, the availability of input files to the program and the system reliability of the subsystem.

The reliability of a service is determined by the distributed programs reliability in each subsystem and the availability of the control center. If a service can be achieved successfully, the programs running in every subsystem must be successful. The virtual machine should be available at the moment when any program needs certain input file prepared in the virtual machine. It has to be also available during the period when the programs are being executed in the virtual machine.

It can be obtained through the critical path method, see e.g. Hillier and Lieberman (1995), that the time point when the programs require the files prepared in the virtual machine (T_{bf}^{j}), j=1,2,…J. We can also obtain the starting time when the programs run in the virtual machine (T_{bp}^{k}) and the corresponding execution time period for those programs (T_{ex}^{k}), k=1,2,…,K.

It is noted that $A(t)$ is the availability of the virtual machine at time t. We also assume that the programs require input files at the beginning time, T_{bf}^{j}, so the availability of the input files can be calculated as

$$P_f(j) = A(T_{bf}^{j}) \quad = A(T_{bf}^{j}), \; j=1,2,\ldots J \tag{4.1}$$

It is assumed that the virtual machine has to be available from the beginning to the end when a program runs in it; otherwise, the program fails. The average availability of the programs, which start at time T_{bp}^{k} with the execution time period T_{ex}^{k}, can be calculated as

$$P_{pr}(k) = \int_{T_{bp}^{k}}^{T_{bp}^{k}+T_{ex}^{k}} A(t)dt / T_{ex}^{k} = \int_{T_{bp}^{k}}^{T_{bp}^{k}+T_{ex}^{k}} A(t)dt / T_{ex}^{k}, \quad k=1,2,\ldots,K \tag{4.2}$$

Let N be the number of subsystems. The distributed system reliability for the i:th subsystem is denoted by DSR_i (i=1,2,…,N) where the virtual machine is viewed as a perfect node in each sub-distributed systems at first. The DSR_i (i=1,2,…,N) can be computed by the various algorithms presented in the previous section. Then, the availability of the virtual machine is incorporated into the distributed service reliability together with the DSR_i.

In order to calculate distributed service reliability, some additional assumptions on statistical independence are needed:

1) DSR_i (i=1,2,…,N) is assumed to be mutually independent;
2) The files prepared in the virtual machine are also mutually independent;
3) The programs running in the virtual machine are mutually independent.

Although the independence assumption may not always be true, they are first order approximation.

The distributed service reliability function to the initial time, t_b, can be calculated by

$$R_s(t_b) = \prod_{i=1}^{N} DSR_i \prod_{j=1}^{J} P_f(j) \prod_{k=1}^{K} P_{pr}(k) \tag{4.3}$$

In this calculation, the evaluation has two steps: 1) given the virtual machine (VM) is always available (or called perfect) when its programs are executed and its files are used, the service reliability is determined by the reliability of those subsystems, calculated by $\prod_{i=1}^{N} DSR_i$; 2) the availability for the programs and files in the virtual machine can be computed by $\prod_{j=1}^{J} P_f(j) \prod_{k=1}^{K} P_{pr}(k)$, so that according to the conditional probability, we get

$R_s(t_b)$ =Pr(service reliability| VM's programs and files are available)$\times$Pr(VM's programs and files are available) $= \prod_{i=1}^{N} DSR_i \prod_{j=1}^{J} P_f(j) \prod_{k=1}^{K} P_{pr}(k)$.

Algorithm for distributed service reliability

In applying the general approach, we will need the system structure and then the above model can be used. The algorithm for the calculation of the distributed service reliability can be presented as the following six steps:

Step 1: Identify the structure of Centralized Heterogeneous Distributed System and relationship between programs and files;

Step 2: Obtain the availability function of the virtual machine with any existing models;

Step 3: Let the virtual machine to be a perfect node in every subsystem and calculate DSR_i (i=1,2,…,N);

Step 4: Using the critical path method to determine T_{bf}^{j} (j=1,2,…J) and T_{bp}^{k}, T_{ex}^{k}

(k=1,2,…,K);

Step 5: Calculate $P_f(j)$ and $P_{pr}(k)$;

Step 6: Calculate the distributed service reliability function to the initial time, t_b.

Note that we can implement different models and methods to calculate distributed service reliability. For subsystems, the DSR_i can be calculated through the algorithms, e.g. *MFST* (Kumar *et al.*, 1986), *FST* (Chen and Huang, 1992), *HRFST* (Chen *et al.*, 1997) etc. For the availability function of the virtual machine $A(t)$, it can be calculated through the models presented by Lai *et al.* (2002).

4.2.2. A case study

The system structure

The structure of this distributed service system is described in Fig. 4.3. The service includes the programs and files as shown in Table 4.1. The execution time, required files and precedent programs are given in Table 4.2.

In Fig. 4.3, there are three subsystems. The network topologies are various, in which "*R*" means router. Table 4.1 shows the programs and prepared files arranged in the distributed system. Table 4.2 shows the relationship between programs and their precedent programs. If there are no precedent programs for a program, it can run at initial time if input files are available. Table 4.2 also shows the input files and execution time for every program. If there are no input files required by a program, it means the program can run immediately after its precedent programs are completed.

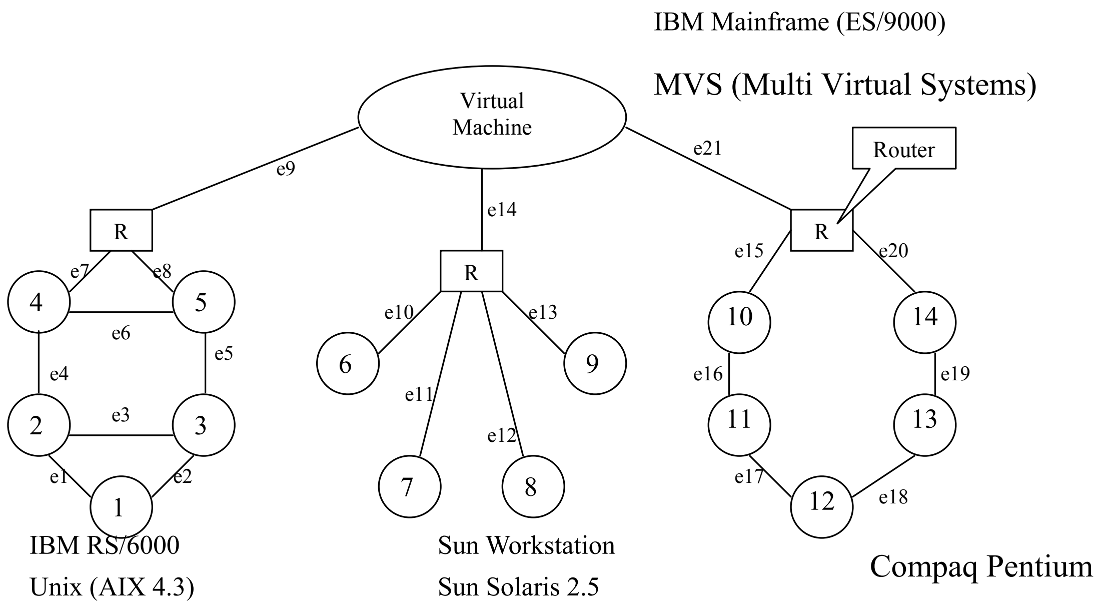


Fig. 4.3. A centralized distributed service system.

Table 4.1. The programs and prepared files in different nodes.

Node	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Progs	P1	P4	P2,P3	P2,P3	P4	P5,P7	P6	P7
Files	F1,F5	F1,F2	F2,F5	F2,F5	F3,F6	F6	F7, F8, F9	F7,F8,F9
Node	**9**	**10**	**11**	**12**	**13**	**14**	**Virtual Machine**	
Progs	P5,P6	P8,P11	P9	P10	P9,P10	P8,P11	SP1,SP2,SP3,SP4	
Files	F6	F10,F11, F12	F11	F10	F12	F10,F11 F12	F4,F13,F14	

Table 4.2. Required files, precedent programs and execution time for programs.

Programs	Required Files	Precedent Programs	Execution Time (T_{ex})
P1	F1,F2,F3	------	5
P2	F2,F4,F6	------	25
P3	F1,F3,F5	P1,P2	32
P4	F1,F2,F4,F6	SP1,SP2	33
SP1	F6	P3,P6	43
P5	------	------	17
P6	F6,F13,F9	P5	19
P7	F6,F8	SP2,SP3	21
SP2	F2,F11	P9,P10	16
P8	------	P1	45
P9	F11,F12	P5	121
P10	F11,F14	SP1	37
SP3	F3,F8	P8,P10	21
P11	F14,F10,F12	SP3	32
SP4	F5,F12	P4,P7,P11	20

"------" means no precedent programs or no input files.

The availability function

The failures of the virtual machine is assumed to follow the JM model (Jelinski and Moranda, 1972) with the failure intensity function

$$\lambda(k_t) = \phi \cdot k_t,\ \ 0 \le k_t \le K_0 \tag{4.4}$$

where ϕ is the expected failure rate per fault, k_t is the expected number of remaining faults at time *t* and K_0 is the initial number of faults. The parameters of K_0 and ϕ are assumed to be 10 and 0.01, respectively.

We also assume that there are maintenance personnel to repair the failure of the virtual machine and the repair time is exponentially distributed with parameter $\mu = 0.5$.

The virtual machine has two states: state 0 is the working state and state 1 the malfunctioning state, with transition rate $\lambda(k_t)$ (from 0 to 1) and μ (from 1 to 0). Let $P_0(t)$ be the probability for the virtual machine to be working at time *t*, and $P_1(t)$ be the probability for it to be at a malfunctioning state at time *t*. The corresponding Chapman-Kolmogorov differential equations are

$$P_0^{'}(t) = \mu P_1(t) - \lambda(k_t)P_0(t) \tag{4.5}$$

where k_t can be obtained by

$$k_t = \sum_{k=0}^{K_0} k \cdot P(k,t) \tag{4.6}$$

in which

$$\begin{aligned} P(k,t) &= \Pr(k \text{ faults remaining in the virtual machine at time } t) \\ &= \binom{K_0}{k} e^{-k\phi x} \cdot [1 - e^{-\phi x})]^{K_0 - k} \end{aligned} \tag{4.7}$$

Also the following equation is satisfied at any time *t*,

$$P_1(t) = 1 - P_0(t) \tag{4.8}$$

Together with the initial conditions $P_0(0)=1$, $P_1(0)=0$, the availability function can be obtained as

$$A(t) = P_0(t) = \left[\int_0^t \mu \exp\{\mu x - K_0 e^{-\phi x}\} dx + \exp(-K_0) \right] \cdot \exp\{-\mu t + K_0 e^{-\phi x}\} \tag{4.9}$$

The distributed system reliability

The distributed system reliability from the left subsystem to the right subsystem in Fig. 4.3 is denoted by DSR_i (*i*=1,2,3). The three subsystems can be separated as shown in Fig. 4.4.

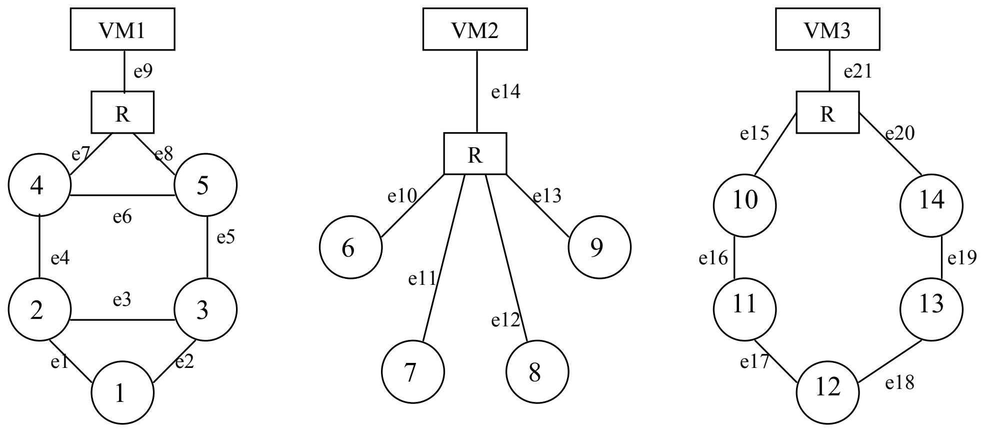


Fig. 4.4. The separated subsystems from Fig. 4.3.

VM*i* (*i*=1,2,3) represents the virtual machine used in subsystem *i*. DSR_i (*i*=1,2,3) can then be calculated numerically with the assumptions that all the nodes are perfect and the probability for every communication edge to be available is 0.99. Hence, we can obtain the result of DSR_1=0.9998 through the *HRFST* algorithm (Chen *et al.*, 1997). In the same way, we get DSR_2=0.9699 and DSR_3=0.9975.

The distributed service reliability function

The critical path graph, see Hillier and Lieberman (1995, pp. 389-395) for details, of this example given in above Table 4.2 is drawn in Fig. 4.5.

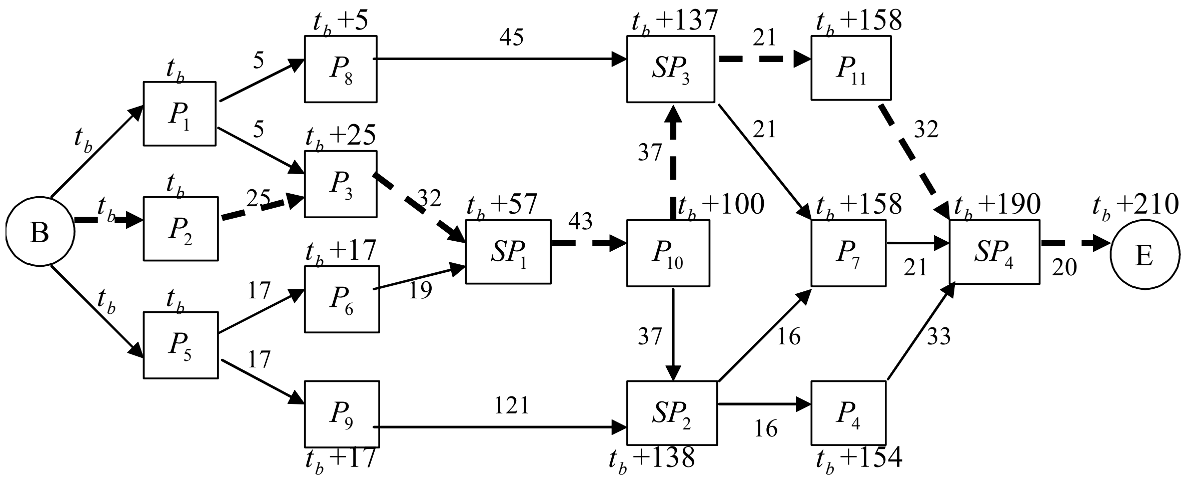


Fig. 4.5. Critical Path for Table 4.2. The value marked on the edge is the execution time and those on the node is the starting time and the black-dashed lines is the critical path.

From the critical path shown in Fig. 4.5 and Table 4.2, T_{bf}^{j} (j=1,2,…,5) can be shown to be $\{t_b,\ t_b+17,\ t_b+100,\ t_b+154,\ t_b+158\}$ for the programs {P2, P6, P10, P4, P11} using the files prepared in the virtual machine. We can also get T_{bp}^{k} (k=1,2,3,4) to be $\{t_b+57,\ t_b+137,\ t_b+138,\ t_b+190)$ and the corresponding execution time period T_{ex}^{k} to be {43, 21, 16, 20} for the programs {SP1, SP2, SP3, SP4} executed in the virtual machine.

Then, we can get

$$P_f(j) = A(T_{bf}^{j}),\ j=1,2,\ldots,5$$

in which T_{bf}^{j} is $\{t_b,\ t_b+17,\ t_b+100,\ t_b+154,\ t_b+158\}$ and

$$P_{pr}(k) = \int_{T_{bp}^{k}}^{T_{bp}^{k}+T_{ex}^{k}} A(t)dt / T_{ex}^{k},\ k=1,2,3,4$$

in which T_{bp}^{k} is $\{t_b+57,\ t_b+137,\ t_b+138,\ t_b+190)$ and T_{ex}^{k} is {43, 21, 16, 20}.

Thereafter, we can obtain the distributed service reliability function to service starting time t_b as

$$R_s(t_b)=\prod_{i=1}^{3} DSR_i \prod_{j=1}^{5} P_f(j) \prod_{k=1}^{4} P_{pr}(k)$$

This distributed service reliability function has the form displayed in Fig. 4.6.

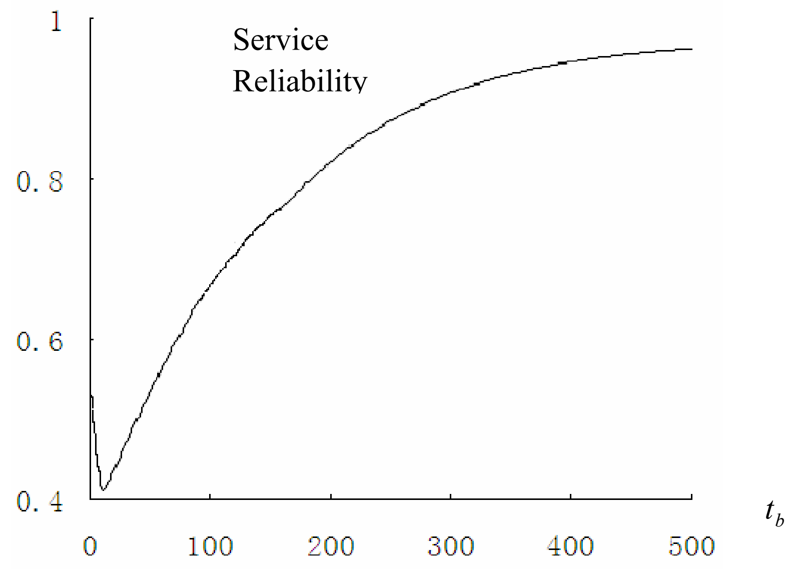


Fig. 4.6. Typical distributed service reliability function to service starting time.

From Fig. 4.6, it can be observed that the lowest service reliability is not at the initial time point when the software failure rate of the virtual machine is the highest. This is because we assumed that the initial state for the virtual machine is up (working). When t_b is larger than the lowest point, the distributed service reliability increases. It is because those identified bugs of the virtual machine are fixed, which results a decrease in the failure rate. Towards the end, the distributed service reliability approaches a steady value of 0.9673. It is obtained by the reliability of subsystems that cannot be improved by debugging the control center:

$$\prod_{i=1}^{3} DSR_i = 0.9998 \times 0.9699 \times 0.9975 = 0.9673$$

When the availability of the virtual machine approaches 1, the distributed service reliability is approaching to 0.9673.

4.3. Further Analysis and Application of the General Model

With specific input parameters, the distributed service reliability can be computed. Via the modeling and further analysis, some general conclusions can be drawn. The VM in the control center is the heart of the CHDS, and hence, the system availability $A(t)$ of the VM is critical to the distributed service reliability. In order to achieve a high reliability of the service, the control center should be equipped with sufficient maintenance personnel to repair the failures of the VM. The availability function of the VM can help the decision maker to allocate maintenance personnel effectively at different stages and decide the release time that reaches certain pre-required system availability. In this section we discuss some related analysis that makes use of the general model that could be of importance in practical applications.

4.3.1. A general approach

The system availability reaches the lowest point at an early stage. This is because a large number of faults are identified when system testing begins. The system availability starts recovering after the lowest point and approaches to a steady value after a long period of time. This is the case when identified faults are fixed. The time at which the system availability reaches its minimum is important. Around the minimal system availability time point t^*, a significant amount of effort needs to be put into fault fixing and system testing to help increase system availability of the VM quickly. When the faults are fixed, the system availability recovers, and effort on fault fixing and testing can be reduced accordingly. Eventually, only a few faults will be left. At this stage, the manpower for the fault fixing and system testing of the VM can be moved to somewhere else. Hence, the minimum system availability time point t^* is an important indicator for the managers of the control center to distribute the resources on the VM at different stages.

It is easy to calculate the time of minimum system availability if the availability function of the VM, $A(t)$, is known. By differentiating $A(t)$, and then solving $A^{'}(t)$ =0, we can get the solution that is the minimum time point t^*.

Furthermore, if the management wants to know the time when the system of VM reaches certain availability level A_L, the system availability function $A(t)$ can also be used by solving the equation of $A(t) = A_L$. Its solution can help the managers to decide the release time of the VM accordingly. For example, the customers may require the system availability to be at least A_L. Hence, we need to know the time point when the system availability reaches this required system availability level. The testing can be stopped and the system can be released after that.

Another important issue in this type of analysis is the sensitivity studies. Usually the model parameters are assumed to be known. A deviation from the assumed value could lead to significant difference between the actual and the calculated values. To minimize the error, effort should be allocated to obtain accurate estimates of the important parameters. Since a number of parameters are involved, it is useful to identify the ones that influence the results most. Sensitivity analysis of the parameters is highly recommended. This type of results can help decision makers and analysts to better allocate the resources.

4.3.2. The application example revisited

To clearly address some of the issues raised in the previous section, we revisit the application example in Section 4.2 with some further analysis. This type of study is important in system studies and for the management to fully make use of the modeling and analysis.

Minimum system availability of the VM

The minimum availability point can be obtained by taking the derivative of Eq. (4.9) and set it to be zero. That is

$$A^{'}(t) = \mu + \left[\int_0^t \exp\{\mu x - K_0 e^{-\phi x}\}dx + \frac{1}{e^a}\right] \cdot \exp\{-\mu x + K_0 e^{-\phi x}\} \cdot (-\mu - K_0 \phi e^{-\phi t})$$

Let $A^{'}(t)$ =0 and let t^* be the solution, i.e.,

$$\mu + \left[\int_0^t \exp\{\mu x - K_0 e^{-\phi x}\}dx + \frac{1}{e^a}\right] \cdot \exp\{-\mu x + K_0 e^{-\phi x}\} \cdot (-\mu - K_0 \phi e^{-\phi t}) = 0 \quad (4.10)$$

We can easily obtain the value of t^* numerically or using *Maple* or *Mathematica*, or other symbolic software.

For example, with parameters K_0=10, ϕ=0.01 μ=0.5, Eq. (4.10) can be solved by *Maple* that t^*=8.88 and the minimum system availability $A(t^*)$=0.8453.

Time to achieve a required system availability

Suppose that the customers require the system availability to be at least A_L when release. From the Eq. (4.9), the result can be obtained by solving the following equation

$$A(t) = \left[\int_0^t \mu \exp\{\mu x - K_0 e^{-\phi x}\} dx + \exp(-K_0) \right] \cdot \exp\{-\mu t + K_0 e^{-\phi x}\} = A_L \tag{4.11}$$

Since there are two solutions, we require that $t \geq t^*$ where t^* can be solved by (4.10) first.

A simple approximation is presented here for solving (4.11) and carrying out further analytical study. In a Markov Chain, there is a transition time from initial state to steady state. We assume that it takes more time between the initial time and the release time of the test than the transition time of the Markov process. Based on the assumption, from the equations for long-run Markov chain (Hillier and Lieberman, 1995 pp. 640-642), we get

$$A(t) = P_0(t) = \frac{\mu}{K_0 \phi e^{-\phi t} + \mu} \tag{4.12}$$

In order to calculate the time point that is satisfied with the customers' requirement A_L, let $A(t) = A_L$ and *t* can be obtained as

$$t = -\frac{1}{\phi} \cdot \ln\left((\frac{1}{A_L} - 1) \cdot \frac{\mu}{K_0 \phi} \right) \tag{4.13}$$

Sensitivity analysis

There are three parameters in the availability function (4.9), K_0, ϕ and μ. The sensitivity of different parameters is described in Fig. 4.7 and Fig. 4.8.

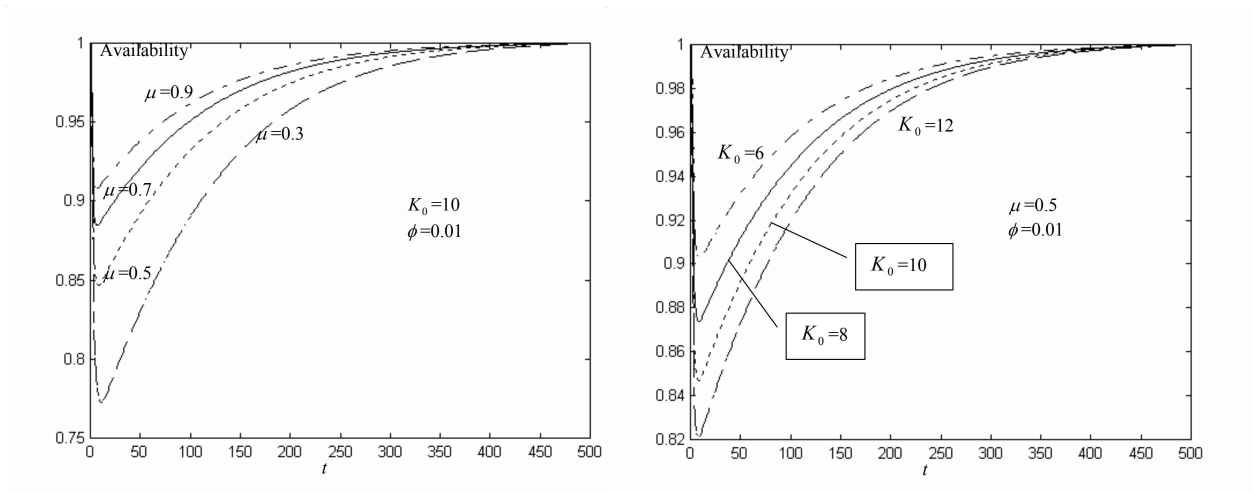


Fig. 4.7. Sensitivity of μ (left) and K_0 (right).

As expected, a greater repair rate implies higher system availability. Similarly, when K_0 increases, the system availability decreases because the failure intensity function increases. However, the case of parameter ϕ is not obvious, as shown in Fig. 4.8.

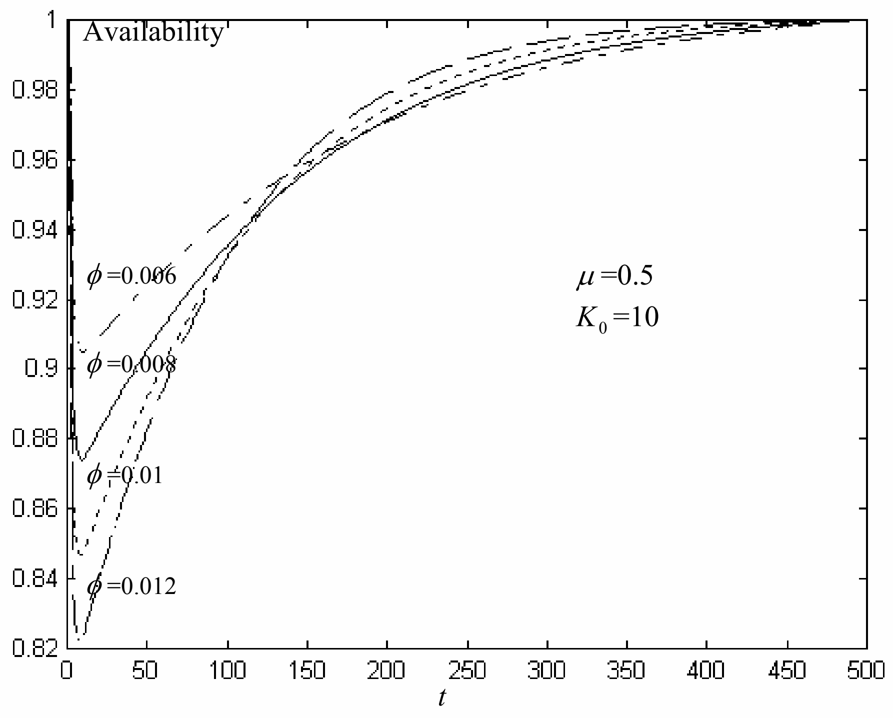


Fig. 4.8. Sensitivity of ϕ.

The curves in the Fig. 4.8 are crossed with one another. It means when ϕ increases, system availability decreases at the early stage and increases at the later stage.

With Eq. (4.10), we can calculate the time point of the minimum system availability and the time a certain availability is achieved. On the other hand, it would be useful to see the influence of the repair rate on these two quantities. We analyze the Markov model with the numerical example presented above. It is assumed that K_0=10 and ϕ =0.01. Let μ change from 0.3 to 0.7 to calculate the minimum system availability point through Eq. (4.10). The time of the minimum system availability t^* *vs.* the repair rate μ is described in the left curve of Fig. 4.9. The minimum system availability $A(t^*)$ *vs.* μ is depicted in the right curve of Fig. 4.9.

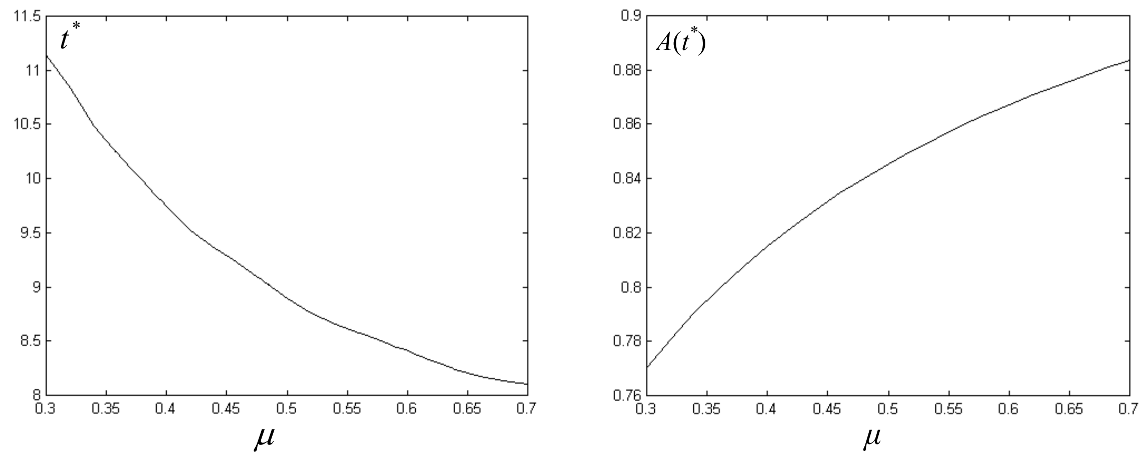


Fig. 4.9. Sensitivity analysis of repair rate.

From Fig. 4.9, we can see the rate of decrease in t^* (rate of increase in $A(t^*)$) as the repair rate μ increases. We can also see that $A(t^*)$ is a convex function of μ. This means that adding $\Delta\mu$ on a small μ improves more availability than adding the same $\Delta\mu$ on a large μ. The curve of "t^* *vs.* μ" is concave, which means that adding $\Delta\mu$ on a small μ reduces more time of minimum availability than adding the same $\Delta\mu$ on a large μ. This type of study is useful when allocating the maintenance personnel optimally although is another interesting problem for further research.

4.4. Conclusions

In this part, a general model was presented for the centralized heterogeneous distributed system. Based on this model, solution algorithm was presented and the time for the virtual machine to reach its minimum system availability or required system availability was studied. An application of the model on an actual bank automatic payment system was shown. Furthermore, sensitivity analysis of the intrinsic parameters to affect the system availability and the lowest availability point was conducted.

Since our approach is general and the CHDS has been applied in different areas, the algorithm for the distributed service reliability analysis can be used to estimate the reliability of the service in a distributed system during both the testing phase and the operational phase. During the testing phase, the service reliability function can help to allocate testing resources accordingly. For example, around the minimum service reliability time, more maintenance persons and testing resources should be allocated to test and repair the system; and at the later stage, the service reliability is high so that the amount of testing resource can be reduced. Also, if given a requirement on the service reliability after release, the time for release can also be determined. Moreover, for projects with fixed deadline, the model can help system managers to determine the testing intensity or manpower according to the estimated reliability given different levels of testing intensity. Furthermore, during the operational phase, the quality of service can also be assessed through the service reliability measure.

For wide-area distributed computing, the structure of centralized heterogeneous distributed systems is just a classical and conventional structure. Nowadays, the grid computing system is a newly developed system in wide-area distributed systems by focusing on large-scale resource sharing. The structure of the grid is not only CHDS but also a kind of peer-to-peer structure. Thus, the next chapter will study the behavior and reliability of the grid computing systems.

CHAPTER 5 GRID COMPUTING SYSTEM RELIABILITY

The grid computing system is a recently developed technique for complex systems with large-scale resource sharing, wide-area program communicating, and multi-institutional organization collaborating etc. Many experts believe that the grid technologies will offer a second chance to fulfill the promises of the Internet. However, it is difficult to analyze the grid reliability due to its complexity.

This chapter first constructs a grid reliability model and then presents approaches to estimate the grid reliability related to different aspects of the grid, including the resource management system, networks and programs/resources. Section 5.1 introduces the evolutionary of the grid technology and builds a general model for the grid architecture. Section 5.2 studies the grid reliability related to the resource management system. Then, Section 5.3 presents a new reliability model for the grid networks. Finally, Section 5.4 further extended the previous model by combining the reliability of software programs and resources together with the hardware reliability of the network.

5.1. Introduction of the Grid Computing System

5.1.1. Grid technology

The term “Grid” was created in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering (Foster and Kesselman, 1998). Grid concepts and technologies were first developed to enable resource sharing within far-flung scientific collaborations. Applications include collaborative visualization of large scientific datasets (pooling of expertise), distributed computing for computationally demanding data analyses (pooling of compute power and storage), and coupling of scientific instruments with remote computers and archives (increasing functionality as well as availability).

The Grid concept is motivated by real and specific problems and there is an emerging, well-defined Grid technology that addresses significant aspects of this problem. The Grid technology is distinct from other major technology trends, such as Internet, enterprise, distributed, and peer-to-peer computing.

The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations (Foster *et al.*, 2001). The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals or institutions are defined by such sharing rules form what we call *virtual organization* (VO).

For example, in a data grid project thousands of physicists at hundreds of laboratories could be involved, and they can be divided into different virtual organizations according to their locations or functions. It is depicted by Fig. 5.1.

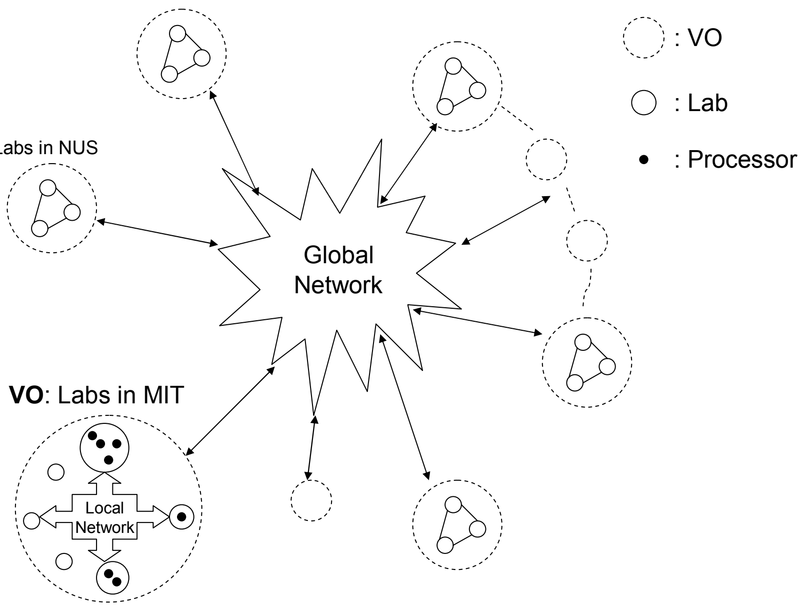


Fig. 5.1. A grid computing system containing many virtual organizations.

In this case, virtual organizations can vary tremendously in their purpose, scope, size, duration, structure, community, and sociology. Nevertheless, careful study of underlying technology requirements leads us to identify a broad set of common concerns and requirements and current distributed computing technologies do not address the concerns and requirements of the grid.

Over the past several years, research and development efforts within the grid community have produced protocols, services, and tools that address precisely the challenges that arise when we seek to build scalable virtual organizations, e.g. Foster and Kesselman (1998), Hoschek *et al.* (2000), Foster *et al.* (2001, 2002), Frey *et al.* (2002) and Buyya *et al.* (2003).

Because of their focus on dynamic, cross-organizational sharing, Grid technologies complement rather than compete with the existing distributed computing technologies. For example, enterprise distributed computing systems can use the grid technologies to achieve resource sharing across institutional boundaries. The grid technologies can also be used to establish dynamic markets for computing and storage resources, hence overcoming the limitations of current static configurations.

The continuing decentralization and distribution of software, hardware, and human resources make it essential that we achieve the desired quality of service (QoS) on resources assembled dynamically from enterprise, service provider, and customer systems. This requires new abstractions and concepts that let applications access and share resources across wide-area networks. This also requires to providing common security semantics, system reliability, distributed resource management performance, or other QoS metrics that are of importance in a particular context. For some time, such problems have been of central concern to developers of distributed systems for large-scale scientific research. Work within this community has led to the development of Grid technologies, which have been widely adopted in scientific and technical computing.

Although the development tools and techniques for the grid have been studied and equipped, the analytical tools for the grid reliability analysis are just inchoate and hard to approach perhaps due to the complexity of the grid. Thus, as one of the important measures of QoS for the grid, the grid reliability needs to be precisely and effectively assessed using new analytical tools. There are many analytical tools for the reliability analysis of the conventional distributed systems. However, they cannot be directly implemented in the reliability analysis of the grid because of some of their impractical assumptions.

Models and tools are needed to analyze the grid reliability. This chapter presents some new results based on general grid reliability models that relax some unsuitable traditional assumptions in the small-scale distributed computing systems.

5.1.2. General architecture of grid computing system

The general architecture of the grid computing systems can be depicted as Fig. 5.2. The virtual node is a general unit in the grid, which can execute programs or share resources. Virtual nodes are connected with one another through the virtual links. Virtual organizations are made up of a number of virtual nodes.

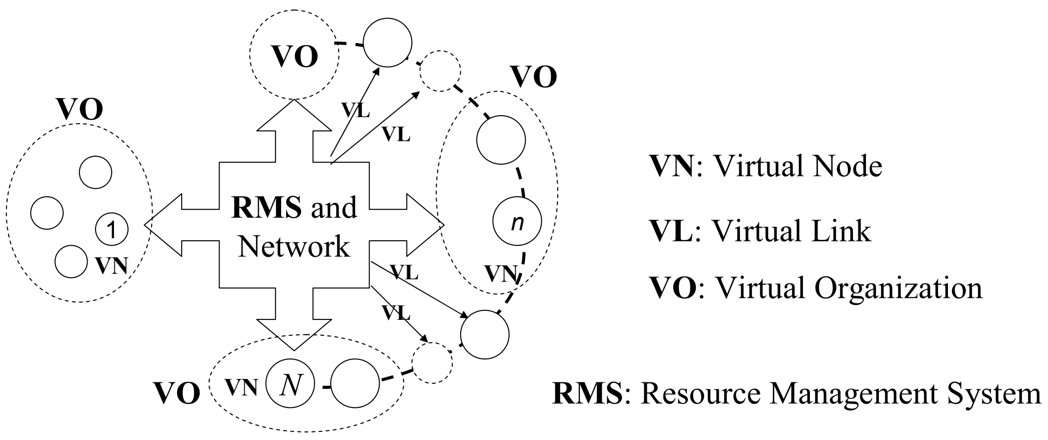


Fig. 5.2. General architecture of grid computing systems.

A grid system is designed to complete a set of programs/applications, so that to complete certain tasks. Executing those programs need use some resources in the grid. These programs and resources are distributed on the virtual nodes as in Fig. 5.2. A virtual link between two virtual nodes (*i* and *j*), denoted by $L(i, j)$, is defined as a direct communication channel between the two nodes *i* and *j* without passing through other virtual nodes.

Let $\vec{U}_n$ represent the set of resources shared by the *n*:th virtual node and $\vec{V}_n$ represent the set of programs executed by the *n*:th virtual node, (n=1,2,…,N). We also assume that *M* programs denoted by $P_1, P_2, ..., P_M$ are running in the grid system. The required processing time for each program is denoted by $t(1), t(2), ..., t(M)$, respectively. The programs may use some necessary resources during their execution, which is in fact to exchange information between them through the network. These resources are denoted by $R_1, R_2, ..., R_H$ which is registered in a resource management system of the grid.

When a program requests certain remote resources, the resource management system receives these requests, matches the registered resources to the requests, and then "tells" the program the sites of those matched resources. After the programs "know" the sites of their required resources, they begin to "access to" them through the network.

In an early stage, the grid reliability is mainly determined by the reliability of the resource management system, while in a later stage, the grid reliability is mostly affected by the reliability of the network for communicating or processing. The grid reliability model related to the two stages will be studied respectively in the following two sections. Then, section 5.4 further integrates other components such as software and resources etc into the grid reliability analysis.

5.2. Grid Reliability of the Resource Management System

Before the programs begin to access to their required resources in the grid, they have to know the sites of those resources, which is managed by the resource management system. The resource management system of the grid, see e.g. Livny and Raman (1998), is to receive the resource requests from application programs, and then to match the requests with the registered resources.

The resource management system is not perfect. It is possible to assign wrong resources to a program. Although the failures of resource management system might be sporadic compared to those of programs, resources, networks or processors, to estimate the grid reliability the resource management system have to be considered because it is one of the most important factors in affecting the quality of service of the grid.

5.2.1. Introduction of resource management system

For grid computing, the resource management system that manages its pool of shared resources is very important. This is especially the case for Open Grid Service Architecture, see e.g. Foster *et al.* (2002), that allows individual virtual organizations to aggregate their own resources on the grid.

The resource management system provides resource management services, which can be divided into four general layers as depicted by Fig. 5.3. They are program layer, request layer, management layer and resource layer.

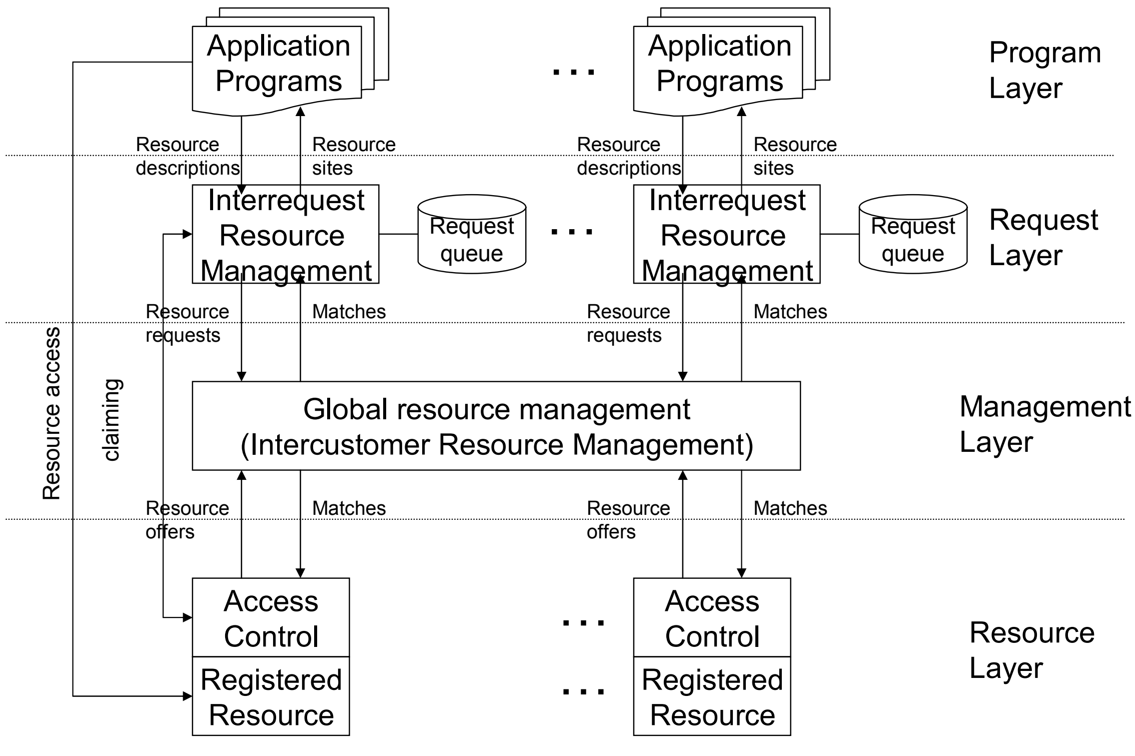


Fig. 5.3. Layers of resource management system.

1) *Program layer*: The program layer represents the programs (or tasks) of the customer's applications. The programs describe their required resources and constraint requirements (such as deadline, budget, function etc).
2) *Request layer*: The request layer represents the program's requirement for the resources. This layer provides the abstraction of "program requirements" as a queue of resource requests.
3) *Management layer*: The management layer may be thought of as the global resource allocation layer and its principal function is to match the resource requests and resource offers so that the constraints of both are satisfied.
4) *Resource layer*: The resource layer represents the registered resources from different sites including the requirements and conditions.

In the grid computing, failures may occur at any of the layers in the resource management system. For example,

1) In the program layer, the resource described by the program may be unclear or translated into wrong resource requests;
2) In the request layer, the request queue may be too long to be waited by the program (generating so called time-out failures), or some requests may be lost due to certain management faults;
3) In the management layer, the request may be matched to a wrong resource because of misunderstanding or faulty matching;

4) In the resource layer, the virtual organization may register wrong information of their resources or remove its registered resources without notifying/updating the resource management system.

If a grid program experiences the above resource management system failures, the program cannot be achieved successfully. The grid reliability should be computed by considering not only the reliability of physical networks or processing elements but also the resource management system reliability. In order to analyze the resource management system reliability, we construct a Markov model below.

5.2.2. Markov modeling

For the resource management system, if any failure that the program is matched to a wrong resource occurs, the program will send a failed feedback to it. It will remove the faults that cause the failures through an updating/debugging process. It is also possible for new faults to be generated in the resource management system such as some virtual organizations register wrong resources to it, etc. The assumptions for the resource management system reliability model are listed as follows:

1) The failures of resource management system follow an exponential distribution with failure rate function $\lambda(k)$ where k is the number of contained faults.
2) If any failure occurs, a fault that causes this failure is assumed to be removed immediately by an updating/debugging process, i.e. the time for removing the detected fault is not counted.

3) The resource management system may generate a new fault, and the occurrence of such event follows an exponential distribution with a constant rate v.

According to the above assumptions, the reliability model of resource management system can be built by a continuous time Markov chain (CTMC). This Markov model depicted in Fig. 5.4 is a typical birth-death Markov process with infinite number of states, where state *k* represents *k* faults contained in the resource management system.

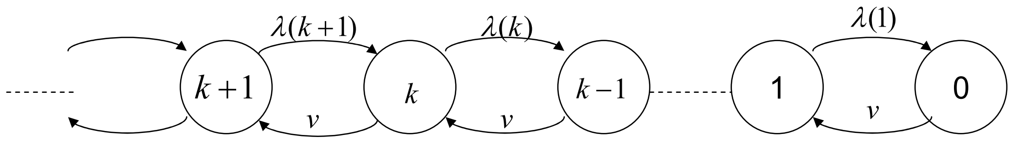


Fig. 5.4. CTMC for resource management system reliability model.

In this model, $\lambda(k)$ can be a general function to the number of remaining faults *k*. Usually, $\lambda(k)$ is an increasing function to the number of remaining faults *k*. The resource management system is desired to serve a long time, especially for the Open Grid Service Architecture (Foster *et al.*, 2002), so the birth-death process of failures can be viewed as a long-run Markov process (Trivedi, 1982). After running a long time, the expected death rate $\lambda(k)$ will trend to a steady value.

Therefore, after a long time run, the failure rate $\lambda(k)$ can be approximately viewed as a constant during a small enough time. An example is illustrated below.

Example 5.1. Consider a grid program denoted by P1 need access to remote resources. The time for resource management system to deal with its request is supposed *t*=15 seconds and the failure rate of resource management system at that time slot $\lambda = 0.0005$ per second. The reliability for the resource management system to deal with the request is computed as

$$R_{RMS}(P1) = \exp\{-\lambda \cdot t\}=0.992528$$

Based on the long-run birth-death Markov process, this approximation of constant failure rate indicates a way to reasonably and dynamically update the failure rate at different time slots. The resource management system can count the number of failures, say *n*, reported by the grid programs between a relatively small time interval, say Δt, and dynamically updates the value of failure rate by $\hat{\lambda} = n / \Delta t$.

Also, the fault birth rate v can be reduced through some information controls such as standardized resource registering, synchronic resource updating, consistent resource descriptions etc, so that to reduce the reliability of the resource management system.

5.3. Grid Reliability on Network

If the resource management system has informed the programs of the sites of their required resources in the grid after matchmaking, the running programs are able to access to those resources through the grid network as depicted by the previous Fig. 5.2. Then, the grid program/system/service reliability is mainly determined by the reliability of network, which will be studied in the following subsections.

5.3.1. Reliability model for the grid network

After the resource matching process, during the execution process of the programs, failures may occur on either virtual nodes or virtual links (Fig. 5.2). If a failure occurs on the virtual node when it is executing a program, the result out of the program is wrong. If a failure occurs on the link when some information is communicated through it, the communication is incorrect. To analyze the grid reliability, two assumptions about the model are given below:

1) The failures of virtual nodes and virtual links can be modeled by Poisson processes,
2) The failures of different elements (nodes and links) are independent from one another.

The first assumption can be justified as the operational phase without debugging process so that the failure rates can remain constant, see e.g. Yang and Xie (2000). The second assumption can be explained as that since the grid is a wide-area distributed system, the nodes and links should be allocated far away from one another so that the possibility of correlation among them can be viewed as very slight or even negligible.

Different programs can exchange information of different sizes with the same resources. Denote by D_{mh} the size of information exchanged between program P_m (m=1,2,…,M) and resource R_h (h=1,2,…,H). The communication time $T_c(i, j)$

between node *i* and node *j*, can be derived from

$$T_c(i,j) = \frac{D(i,j)}{S(i,j)} \tag{5.1}$$

where *D*(*i*,*j*) is the total size of information exchanged through the *L*(*i*,*j*), and *S*(*i*,*j*) is the expected bit rate of the link.

Denote the failure rate of the node *n* by λ_n and of the link *L*(*i*, *j*) by $\lambda_{i,j}$. If any failure occurs either on the link or on the connected two nodes during the communication, the communication process is viewed as a failed process. The reliability of communication between node *i* and node *j* through the link *L*(*i*,*j*) can be expressed as

$$R_c(i,j) = \exp\{-(\lambda_i + \lambda_j + \lambda_{i,j})T_c(i,j)\} \tag{5.2}$$

Similarly, during the execution of a program, any failure occurring on the virtual node that executes the program will also make the program failed. The reliability of the node *n* to run the program P_m, is then given by

$$R_p(m,n) = \exp\{-\lambda_n t(m)\} \tag{5.3}$$

This network reliability model is much more reasonable for the grid than that of conventional distributed systems (e.g. Kumar *et al.* 1986, Kumar and Agrawal, 1993; Chen *et al.*, 1997; Lin *et al.*, 1999, 2001; Dai *et al.*, 2003a). Those conventional models somehow inherit the assumptions of Kumar (1986) model. The most stringent assumption that is not suitable for the grid is that the operational probabilities of nodes or links are assumed constant, i.e. $R_c(i,j)$ and $R_p(m,n)$ in the above two equations are constant no matter how long or how different the $T_c(i,j)$ and $t(m)$ are.

Some concepts of grid reliability are defined as follows.

Definition 5.1. *Grid program reliability* (GPR) is defined as the probability of successful execution of a given program running on multiple virtual nodes and exchanging information through virtual links with the remote resources, under the environment of grid computing system.

Then, the grid system reliability (GSR) can be defined as the probability for all of the programs involved in the considered grid system to be executed successfully.

Furthermore, a grid service is to complete certain programs by using some resources distributed in the grid. The grid service reliability is similar to the grid system reliability by considering the programs of the given service, i.e. without taking other programs that are not used by the service into account. Thereby, the grid service reliability is defined as the probability that all the programs of a given service are achieved successfully.

5.3.2. Reliability of minimal resource spanning tree

Recall that the set of virtual nodes and virtual links involved in running the given programs and exchanging information with the resources form a resource spanning tree. The smallest dominating resource spanning tree (*RST*) is called *MRST* (Minimal Resource Spanning Tree). The reliability of an *MRST* is the probability for the *MRST* to be operational to execute the given program. The reliability of an *MRST* denoted by

R_{MRST} has three parts:

1. Reliability of all the links contained in the *MRST* during the communication;

2. Reliability of all the nodes contained in the *MRST* during the communication;

3. Reliability of the root node that executes the program during the processing time of the program.

The reliability of the link $L(i, j)$ for exchanging the information can be expressed by

$$R_L(i,j)=\exp\{-\lambda_{i,j}T_c(i,j)\} \tag{5.4}$$

The total communication time of the node G_j can be calculated by

$$T(j)=\sum_{i\in D_j}T_c(i,j) \tag{5.5}$$

where D_j represents the set of nodes that communicate with the node G_j in the *MRST*. The reliability function of the node G_j for communication is

$$R_c(j)=\exp\{-\lambda_j T(j)\} \tag{5.6}$$

Finally, the reliability for a program P_m to be executed successfully during the processing time $t(m)$ on the node *n* is $R_p(m,n)$.

The reliability of the *MRST* can be derived from the above equations as

$$\begin{aligned}R_{MRST}&=R_p(m,n)\prod_{L(i,j)\in MRST}R_L(i,j)\prod_{G_j\in MRST}R_c(j)\\&=\exp\{-\lambda_n t(m)\}\prod_{L(i,j)\in MRST}\exp\{-\lambda_{i,j}T_c(i,j)\}\prod_{G_j\in MRST}\exp\{-\lambda_j T(j)\}\\&=\exp\{-\lambda_n[t(m)+T(n)]\}\prod_{L(i,j)\in MRST}\exp\{-\lambda_{i,j}T_c(i,j)\}\prod_{\substack{G_j\in MRST\\ j\neq n}}\exp\{-\lambda_j T(j)\}\end{aligned}$$

(5.7)

In order to simplify the expression of the above equation, we generalize the term of "communication time" for the root node that contains not only the time of exchanging information with other elements but also the time of executing the given program, i.e. $t(m)+T(n)$.

The term of "*element*" is defined here to represent both the nodes and links of the *MRST*. Assume there are totally *K* elements in an *MRST*, so that $element_i$ (*i*=1,2…,*K*) denotes the *i*:th element in the *MRST*. Accordingly, the communication time of the *i*:th element is denoted by $T_w(element_i)$ and $\lambda(element_i)$ represents its failure rate. The reliability of the *MRST* of the above equation can be simply expressed as

$$R_{MRST}=\prod_{i=1}^{K}\exp\{T_w(element_i)\cdot\lambda(element_i)\} \tag{5.8}$$

With this equation, the reliability of an *MRST* can be computed if the communication time and failure rate of all the elements are given. Hence, finding all the *MRST*'s and determining the communication time of their elements are the first step in deriving the grid program reliability and grid system reliability.

The same program executed by different root nodes may cause different communication time on the same elements. Hence, the *MRST*'s should be treated distinctly for the same program executed by different nodes. An example is given below.

Example 5.2. As shown in Fig. 5.5, program P1 can run successfully when either computing node G1 or G4 is successfully working during the processing time and it is able to successfully exchange information with the required resources (say R1, R2 and R3) .

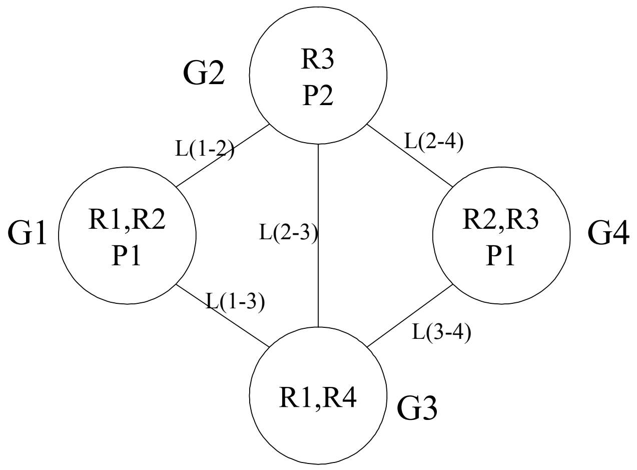


Fig. 5.5. A four-node computing system.

The *MRST*'s considering the communication time of the elements should be separated into two parts:

a) P1 being executed by G1 contains three *MRST*'s: 1) {G1, G2, L(1,2)}; 2) {G1,G2,G3, L(1,3); 3) {G1,G3,G4,L(1,3),L(3,4)}.
b) P1 being executed by G4 contains another three *MRST*'s: 4) {G3, G4, L(3,4)}; 5) {G2, G3, G4, L(2,4), L(2,3)}; 6) {G1,G2,G4,L(1,2),L(2,4)}

If not considering the difference of communication time, the first tree dominates the sixth tree and the fourth tree dominates the third tree. However, in the grid computing system, P1 executed by G1 needs to communicate with the remote resource R3 through the network while P1 executed by G4 needs to communicate with the another remote resource R1, which will differ the communication time of the same elements in the first *MRST* from the sixth *MRST* and the fourth *MRST* from the third *MRST*. Hence they should be treated as different *MRST*'s, i.e. one cannot dominate another.

An algorithm is presented in Dai *et al.* (2002) to search the *MRST*'s for a given program executed by one given virtual node. Repeatedly using this algorithm, all the *MRST*'s for different virtual nodes to execute this program can be found, respectively. This algorithm can be briefly described as follows:

Step 1. Start from the given node to search the required resources along the possible links, and record elements that compose the searching route and their communication times.

Step 2. Until all the required resources are reached, an *MRST* is found, and record this *MRST*.

Step 3. Then other routes are tried to search other *MRST*'s until all the *MRST*'s are searched.

An example of the algorithm to search the *MRST*'s is illustrated below.

Example 5.3. Continue to the above Example 5.2 and see Fig. 5.5 again. The program P_1 is assumed to exchange information with resources R1,R2,R3 (corresponding exchanged information size are: 500,400,300 Kbit). The bit rates of links L(1,2), L(1,3), L(2,3), L(2,4), L(3,4) are assumed 30, 20, 40, 50, 45 (Kbit/s). Then, search the *MRST*'s for P_1 executed by the node G1 and compute the communication time of each elements in those *MRST*'s, as shown by Fig. 5.6.

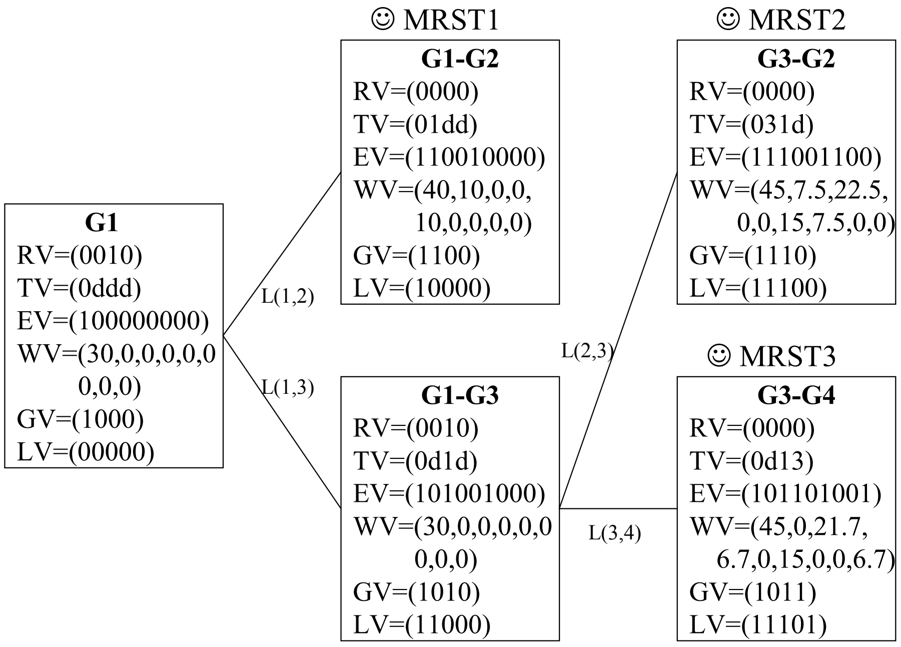


Fig. 5.6. Searching the MRST's of P1 executed by G1.

Three *MRST*'s are found by the algorithm marked by ☺ in the Fig. 5.6 where all the values in vector RV are 0. The corresponding elements contained in those *MRST*'s are recorded in vector EV with the value 1 and the corresponding communication time is saved in vector WV. Similarly, other three *MRST*'s for P_1 executed by the other node G4 can also be obtained as listed in the above Example 5.2.

5.3.3. Grid program and system reliability

Grid program reliability

Note that failures of all the *MRST*s will lead to the failure of the given program, and any one of the *MRST*'s can successfully complete the program only if all of its elements are reliable. The grid program reliability of a given program can be described as the probability of having at least one of the *MRST*'s working successfully,

$$R(P_m) = \Pr(\text{at least one MRST of a given program } P_m \text{ is reliable})$$

Let $N_t(P_m)$ be the total number of *MRST*'s for the given program of P_m and E_j be the event in which the $MRST_j$, *j*=1,2,…, $N_t(P_m)$, is able to successfully execute the given program. The grid program reliability of a given program P_m can be written as

$$R(P_m) = \Pr\left\{ \bigcup_{j=1}^{N_t(P_m)} E_j \right\} \tag{5.9}$$

By using the concept of conditional probability, the events considered in this equation can be decomposed into mutually exclusive events as

$$R(P_m) = \Pr(E_1) + \Pr(E_2)\Pr(\overline{E}_1 | E_2) \ldots + \Pr\{E_{N_t(P_m)}\}\Pr\{\overline{E}_1, \overline{E}_2, \cdots, \overline{E}_{N_t(P_m)-1} | E_{N_t(P_m)}\} \tag{5.10}$$

where $\Pr(\overline{E}_1 | E_2)$ denotes the conditional probability that $MRST_1$ is in the failure state given that $MRST_2$ is in the successful state.

Hence, the grid program reliability can be evaluated in terms of the probability of two distinct events. The first event indicates that the $MRST_i$ is in the operational state while the second indicates that all of its previous trees $MRST_j$ (*j*=1,2,…,*i*-1) are in the failure state given that $MRST_i$ is in the operational state. The probability of the first event, $\Pr(E_i)$ is straightforward, and it can be calculated through Eq. (5.8). The probability of the second event, $\Pr(\overline{E}_1, \overline{E}_2, \cdots, \overline{E}_{i-1} | E_i)$, can be computed using the algorithms presented by Dai *et al.* (2002).

The brief introduction of the algorithm is given here, which has two steps:

Step 1 identifies all the conditional elements that can lead to the failure of any

$MRST_j$ (j=1,2,…,$i-1$) while keeping $MRST_i$ to be operational.

Such a conditional element, say $element_k$ (contained in any $MRST_j$, j=1,2,…,$i-1$), has starting time and end time. If any failure occurs on the $element_k$ between its starting time and end time, it can lead the $MRST_j$ to fail. To keep $MRST_i$ operational, the starting time must be greater than the end time of the same $element_k$ in $MRST_i$.

Step 2 uses a binary search tree (Johnsonbaugh, 2001: pp. 349-354) to seek the possible combinations of these identified elements that can make all the $MRST_j$ (j=1,2,…,$i-1$) fail and computes the probabilities of those combinations.

The summation of the probabilities is the result of $\Pr(\overline{E}_1, \overline{E}_2, \cdots, \overline{E}_{i-1} | E_i)$. For detailed procedures of the two steps can be found in Dai *et al.* (2002). An example of this algorithm is illustrated below.

Example 5.4. Continue to the above Example 5.3 and revisit the results recorded in the vectors (*EV* and *WV*) as Fig. 5.6. We now use the above algorithm to evaluate $\Pr(\overline{E}_1, \overline{E}_2 | E_3)$, the probability that $MRST_1$ and $MRST_2$ fails given $MRST_3$ is operational.

Use step 1 to identify the conditional elements that can fail either $MRST_1$ or $MRST_2$ and keep $MRST_3$ to be operational. Table 5.1 shows the order for communication time T_w of different elements. As step 1, the starting time of any conditional element's failure must be greater than the end time of the same element in $MRST_3$ (shown by the second row). Then, the set of conditional elements can be obtained in Table 5.2 where *CEV*(m) denotes the vector of the m:th conditional element, $T_b(m)$ its starting time, $T_e(m)$ its end time, and its reliability is

$$R(m) = \exp\{\lambda(element_m)\cdot[T_e(m) - T_b(m)]\}.$$

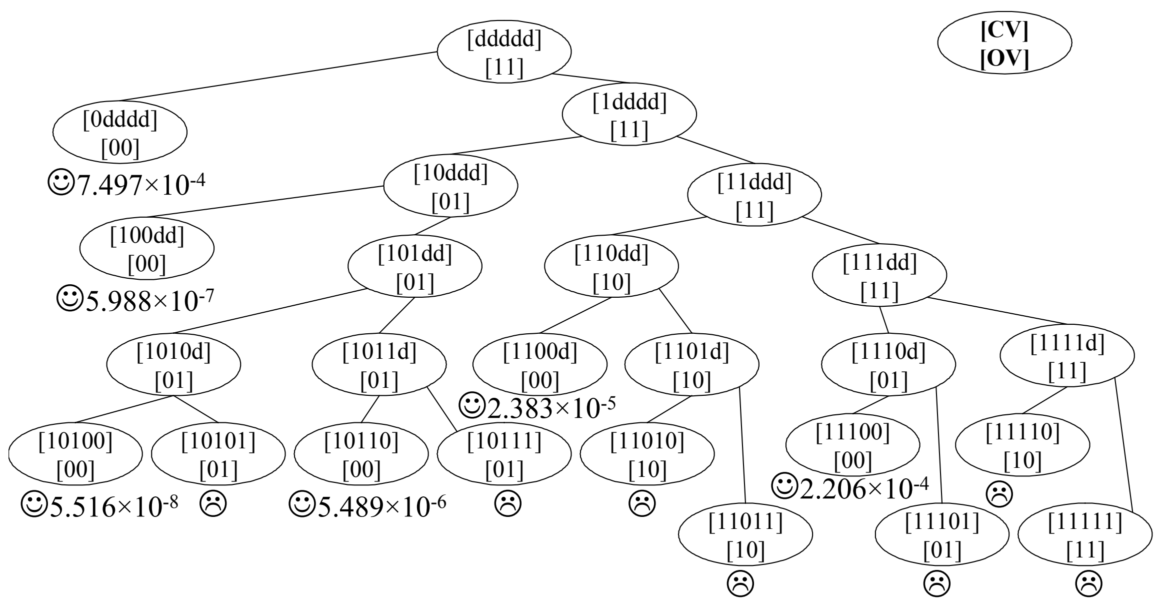
Table 5.1. The order of communication time of different elements.

Elements	G1	G2	G3	L(1,2)	L(1,3)	L(2,3)	L(3,4)
T_w **of** $MRST_3$	45	0	21.7	0	15	0	6.7
Others' T_w **>** **that of** $MRST_3$	None	7.5	22.5	10	None	7.5	None
		10					

Table 5.2. Result outputted by Step 1.

***CEV*(*m*)**	*CEV*(1)	*CEV*(2)	*CEV*(3)	*CEV*(4)	*CEV*(5)
Element	G2	G2	G3	L(1,2)	L(2,3)
$T_b(m)$	0	7.5	21.7	0	0
$T_e(m)$	7.5	10	22.5	10	7.5
$R(m)$	0.9993	0.9998	0.9976	0.9900	0.9778

As Step 2, the binary search tree for calculating $\Pr(\overline{E}_1, \overline{E}_2 | E_3)$ is depicted in the Fig. 5.7.



☺ **Denotes the available leaf for the calculation of probability**

☹ **Denotes the unavailable leaf without contribution to the probability**

Fig. 5.7. Binary search tree for calculating $\Pr(\overline{E}_1, \overline{E}_2 | E_3)$**.**

In Fig. 5.7, the leaves marked by ☺ represents the case that $MRST_1$ and $MRST_2$ fails while $MRST_3$ is operational, and the float value beside ☺ is the probability for the corresponding case to happen. Thus, summing up all the probabilities, we get that

$$\Pr(\overline{E}_1, \overline{E}_2 | E_3) = 0.001000.$$

After computing all the $\Pr(\overline{E}_1, \overline{E}_2, \cdots, \overline{E}_{i-1} | E_i)$ and $\Pr(E_i)$, i=1,2,…, $N_t(P_m)$, we can calculate the grid program reliability of the given program P_m by substituting them into the reliability expression.

Grid system reliability

For the grid system, it is important to obtain a global reliability measure that describes how reliable the grid system is. One way of measuring the reliability of the grid system is given by determining grid system reliability. The grid system reliability equation can be written as the probability of the intersection of the set of *MRST*'s of each program, which is

$$R_S = \Pr\left\{ \bigcap_{m=1}^{M} MRST(P_m) \right\} \tag{5.11}$$

where $MRST(P_m)$ denotes the set of all the *MRST*'s associated with the program P_m. The intersection of the trees of each $MRST(P_m)$ can be evaluated first by intersecting $MRST(P_1)$ and $MRST(P_2)$, i.e. intersect all the combinations of two *MRST*'s one of which is from $MRST(P_1)$ and the other is from $MRST(P_2)$, and so on until all the $MRST(P_m)$ (*m*=1,2,…,*M*) are intersected. The intersected tree of two *MRST*'s is generated by putting all the elements of the two *MRST*'s together, where the communication time of overlapped elements should be added together. An example of intersected *MRST* is illustrated below.

Example 5.5. Suppose one *MRST* related to program P_1 is {G1,G2,G3,L(1,3),L(2,3)} with the communication time {45, 7.5, 22.5, 15, 7.5} and one *MRST* related to program P_2 is {G1,G2,G3,L(1,2),L(1,3)} with the communication time {50, 70, 30, 20, 30}. Then, the intersected *MRST* of the above two *MRST*'s should be {G1,G2,G3,L(1,2),L(1,3),L(2,3)} with the communication time {95, 77.5, 52.5, 20, 45, 7.5}.

In fact, if any one of the intersected *MRST*'s of $MRST(P_m)$ (*m*=1,2,…,*M*) is reliable, all the programs required in the grid system can be successfully completed; If all the intersected *MRST*'s fail, the grid system cannot be successfully completed.

After generating all the intersected *MRST*'s, the grid system reliability can be written as

$$R_S = \Pr\left(\bigcup_{j=1}^{N_t} E_j\right) \tag{5.12}$$

where N_t is the total number of intersected *MRST*'s. This equation is similar to the prevous Eq. (5.9), so the above algorithms for deriving the grid program reliability can be similarly used in deriving the grid system reliability here.

Grid service reliability

The grid service reliability can be viewed as a special type of the grid system reliability if we consider the grid service in a way that the whole grid system is only providing this required service and other services are not considered now. With this classification, the concept of grid system reliability is generalized to include the reliability of different number of services.

All the above algorithms computing the grid program/system reliability are illustrated by a numerical example as below, and then the reliability of resource management system is also integrated into the grid reliability analysis.

Example 5.6. Suppose that a simple grid system is to provide a web service of "Stock Analysis" for different countries. Three different resources (R1,R2,R4) store the real-time stock price of different countries, and another resource (R3) is the database of a website that outputs and shows the results out of the “Stock Analysis”. The service procedure can be described as that two programs (P1 and P2) collect data from the three resources (R1,R2,R4) to analyze the stock market information for different countries, and then output the results into the database (R3) which can be loaded by a website service.

Revisit Fig. 5.5 that contains four virtual nodes and five virtual links and runs the two programs and prepare the four resources. Tables 7.3-7.4 show necessary input information.

Table 5.3. Failure rate and speed of elements (links and nodes).

Elements	**L(1,2)**	**L(1,3)**	**L(2,3)**	**L(2,4)**	**L(3,4)**	**G1**	**G2**	**G3**	**G4**
Failure rate	0.001	0.002	0.003	0.004	0.005	0.001	0.0001	0.003	0.004
Speed (Kbps)	30	20	40	50	45				

Table 5.4. Processing time and information exchanged with the resources.

Program	**Run Time (Sec)**	**Resources**	**Exchanged information (Kbit)**
P1	30	R1, R2, R3	500,400,300
P2	50	R3, R4	200,600

With the approaches presented above, Table 5.5 shows all *MRST*’s of the program P1 with the communication time of each element evaluated by the above Example 5.3 and its reliability $\Pr(E_i)$ calculated by Eq. (5.8). Table 5.5 also shows the conditional probability of $p=\Pr(\overline{E}_1, \overline{E}_2, \cdots, \overline{E}_{i-1} | E_i)$ evaluated similarly as the above Example 5.4.

Table 5.5. Evaluation for the grid program reliability of P1.

MRST$_i$	Elements	Communication time (T_w)	$\Pr(E_i)$	p
$MRST_1$	G1,G2,L(1,2)	40,10,10	0.950279	-----------
$MRST_2$	G1,G2,G3,L(1,3),L(2,3)	45,7.5,22.5,15,7.5	0.847258	0.010198
$MRST_3$	G1,G3,G4,L(1,3),L(3,4)	45,21.7,6.7,15,6.7	0.818403	0.001000
$MRST_4$	G3,G4,L(3,4)	11.1,41.1,11.1	0.776313	0.039890
$MRST_5$	G2,G3,G4,L(2,4),L(2,3)	22.5,12.5,40,10,12.5	0.755973	0.002314
$MRST_6$	G1,G2,G4,L(1,2),L(2,4)	16.7,26.7,40,16.7,10	0.789725	0.000810

Substituting the values of $\Pr(E_i)$ and $\Pr(\overline{E}_1,\overline{E}_2,\cdots,\overline{E}_{i-1}|E_i)$ of Table 5.5 into Eq. (5.10), the grid program reliability of P1 is

$$R(P1)=0.99309 \tag{5.13}$$

Similarly, the grid program reliability of P2 can be obtained as

$$R(P2)=0.773368+0.769665\times 0.122694+0.90801\times 0.0907=0.950158 \tag{5.14}$$

where three *MRST*'s are found for P2 to be executed by G2.

The grid system reliability can then be derived. The total number of intersected trees is $6\times 3=18$. Similar to grid program reliability, the grid system reliability is obtained as

$$R_S=0.926380 \tag{5.15}$$

Suppose total time for resource management system to deal with the program P1's requests is t=15 seconds and the failure rate at that time slot $\lambda=0.0005$ (sec^{-1}). The reliability for the request of the program P1 is then computed as

$$R_{RMS}(P1) = \exp(-\lambda t) = 0.992528 \tag{5.16}$$

The grid program reliability of P1 considering the reliability of resource management system can be calculated by multiplying the above R_{RMS} together with $R(P1)$ as

$$GPR(P1) = R_{RMS}(P1) \cdot R(P) = 0.992528 \times 0.99309 = 0.98567$$

For P2, if the total time for resource management system to deal with its resource requests is 10 seconds, a similar way can be used to obtain

$$R_{RMS}(P2) = \exp(-0.0005 \times 10) = 0.99501$$

Multiplying it with $R(P2)$, we get

$$GPR(P2) = R_{RMS}(P2) \cdot R(P2) = 0.99501 \times 0.950158 = 0.94542$$

For the grid system reliability that includes both P1 and P2, the reliability can be computed as

$$GSR = R_{RMS}(P1) \cdot R_{RMS}(P2) \cdot R_s = 0.992528 \times 0.99501 \times 0.926380 = 0.91487$$

5.4. Grid Reliability on Software and Resources

In the above section, the grid reliability is analyzed by considering only the network hardware failures, i.e. failures of processing nodes and communicating links. However, besides the hardware failures, the software program failures and the resource failures should also be integrated into the grid reliability analysis, which model is more practical and reasonable.

5.4.1. Failures of software programs and resources

Besides the hardware causes, the failures of a software program may also be caused by the faults in the program itself. Justified as in the operational phase (Yang and Xie, 2000), the software program failures can be assumed to follow the exponential distributions here. The software failure occurrence rate of program P_i running on processing node G_j is denoted by $\lambda_s(i,j)$, because a same program running on different processing nodes may have different failure rates. Also, the processing time of P_i on G_j is denoted by $t(i,j)$. Thus, the reliability of the software program P_i running on G_j can be simply computed by

$$R_{\text{prog}}(i,j) = \exp\{-\lambda_s(i,j)\cdot t(i,j)\} \tag{5.17}$$

For the resource reliability, previous section assumes that if the program uses the resource, the resource itself is perfect and the failures only occur when transferring the information through the communication network.

However, the resource may be a software, hardware, database, digital product, etc, that ought to contain faults too. In reality, the resource possibly risks failures when it is working (such as dealing with the information that the programs send or generating the results that the programs need). Suppose the time for resource *h* to work is determined by the program P_i by which the resource is requested and the node G_j on which the resource is integrated, denoted by $t(h,i,j)$. Also, considering the operational phase for the integrated resources, we denote the failure rate of the resource *h* on the node G_j by $\lambda_r(h,j)$, which follows the exponential distribution. Thus, the reliability of resource *h* requested by P_i and integrated on G_j can be simply expressed by

$$R_{\mathrm{res}}(h,i,j) = \exp\{-\lambda_r(h,j)\cdot t(h,i,j)\} \tag{5.18}$$

5.4.2. Approach to grid reliability integrating software and resource failures

In order to integrate the software program and resource failures into grid reliability analysis together with the hardware network reliability, we revise the model presented by section 5.3. For each virtual node, we abstract its programs and resources as its sub nodes, as shown by Fig. 5.8 where G_j is a virtual node on which $P_{x1},\ldots,P_{xm}$ are attached as the sub nodes representing programs and $R_{y1}\ldots R_{yk}$ corresponding to resources.

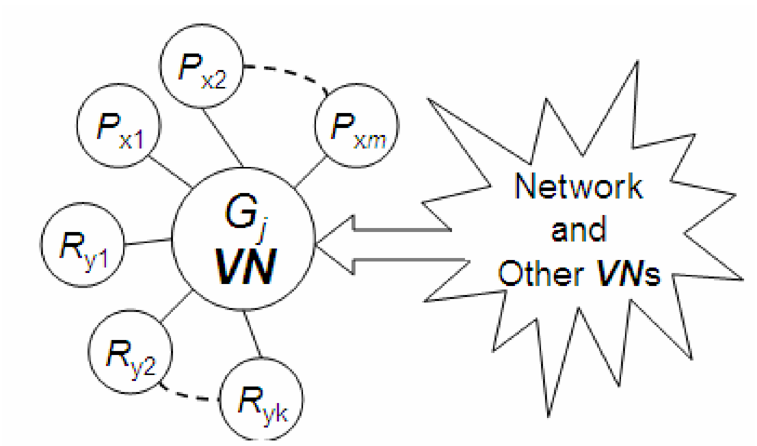


Fig. 5.8. Virtual node and its sub nodes of programs and resources.

Such abstraction of the Fig. 5.8 has the following advantages:

1) The reliability of different software programs and resources can be integrated into the whole grid reliability analysis given the failure rates of all the sub nodes and their communication time.
2) It also combines the hardware reliability into the grid reliability analysis and the common cause failures among those programs and resources are somewhat considered. For example, if G_j fails, all its sub nodes (corresponding to the programs or resources executed by or integrated on the same virtual node) are no longer available for running or using.
3) All the approaches presented in the Section 5.3 can be directly implemented to compute grid program/system/service reliability if each sub node is viewed as an element of failure rate and communication time of its corresponding program/resource, and the link between the virtual node and its sub node is viewed as perfect (i.e. failure rate is 0). Thus, this model integrating the software/resource failures is generalized to satisfy the common condition as Eq. (5.8) of the *MRST* approaches presented by the previous section 5.3.

Example 5.7. Revisit Fig. 5.5. Replace the nodes with those in the Fig. 5.8 that considers the software program and resource failures. Fig. 5.9 depicts the new network graph for the grid computing system containing the sub nodes of programs and resources. According to the third advantage as above, the approaches presented by Section 5.3 can be directly and similarly implemented in deriving the grid reliability of Fig. 5.9.

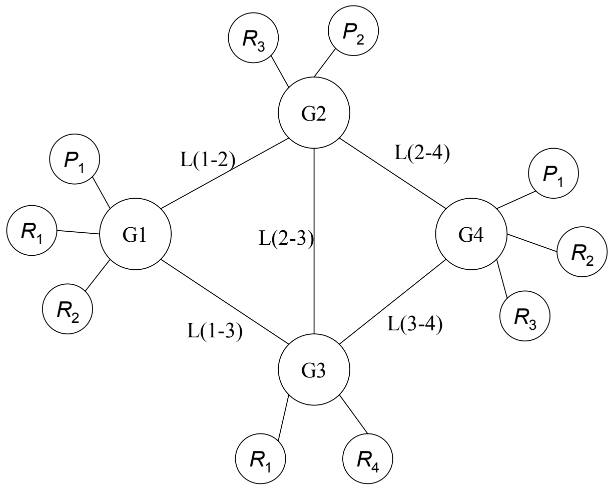


Fig. 5.9. Grid network containing the sub nodes of programs and resources.

CHAPTER 6

MULTI-TYPE FAILURE CORRELATION MODELS

Most software reliability models assume independence of successive software runs. It is a very strong assumption and it is usually not valid in reality. Goseva-Popstojanova and Trivedi (2000) presented an interesting study on failure correlation in software runs. In this chapter, by extending their results, a Markov renewal model is developed for such failure correlation, and further considers multiple types of failures. In addition, the cases of restarting with repair and without repair are both studied. Although such a model is more complex than the traditional approach based on reliability growth, it incorporates more information about the failures and system structure.

6.1. Introduction

Reliability and availability analysis of software systems is a very important issue today. There are many papers and books dealing with software reliability modeling and analysis, for some recent articles, see e.g. Zhang and Pham (2000), Xie (2000), Kuo *et al.* (2001) and Zhang and Horigome (2001). However, a common assumption is the independence of successive software runs, which is not realistic in practice. For example, distributed computing systems have become popular recently, and it is modeled as a collection of resources interconnected via an arbitrary communication network and controlled by a distributed operating system as described by Lin (1999). For this kind of systems, the successive software runs are dependent because the resources is interconnected with one another through the network so that the previous run affect the resources of the next run.

Recently, Goseva-Popstojanova and Trivedi (2000) formulated a Markov model for failure correlation and studied its effects on the software reliability measures. This model assumes a two-state Markov Renewal model. One state is a successful state and another is a failure state among successive runs. The result of next run can be affected by the outcome of the previous run.

In practice, failures can be classified in different types, see e.g. Bukowski and Goble (2001), and a common classification is according to the severity (Lyu, 1996). Other classification schemes are also possible. For example, Tukona (2000) introduced two types of failures. The first type is caused by the faults latent in the system before the testing; the second type is caused by the faults regenerated randomly during the testing phase. Different types of failures may also induce different conditional probability to the next run and there are some examples for failure correlation conditions as described in Kim *et al.* (1996).

In this chapter, as an extension of Goseva-Popstojanova and Trivedi (2000), we present a new Markov Renewal model that allows software failures to be in one of multiple states. Note that if the failures can be of *n* different types, the total number of possible states for every run will be n+1, in which there is a successful state. Thus the two states model in Goseva-Popstojanova (2000) is a special case of our new model.

This chapter is organized as follows. In Section 6.2, a brief overview of multiple failure states and Markov Renewal Model is provided and a model for two types of failure is developed and analyzed. Generalization of the model to n-type failure Markov Renewal Model is also given. In Section 6.3, formulas for probability computation are derived and a numerical example is illustrated. Although the model is complicated, it is still computationally tractable.

6.2. Markov Renewal Model for the Multi-Type Correlated Failures

Except the perfect working state, other states in the system can be viewed as different types of failure states. For example, according to the severity, Lyu (1996) divided the failures into four states: minor, marginal, critical and catastrophic.

Note that if the failures can be of n different types, the total number of possible states for the system will be n+1, in which there is a perfect state. The system state in the next run depends on its state in the current run. This is a kind of multi-type failure correlation among successive runs. Under such conditions, Markov model is also a good tool for analyzing the reliability as it is able to handle one-step dependence.

For the correlated system with n types of failures and a successful state, a general Markov process can be constructed as follows:

1) Build an n+1-state discrete time Markov chain with transition probability matrix

$$\mathbf{P}=\begin{bmatrix} P_{00} & P_{01} & \cdots & P_{0n} \\ P_{10} & P_{11} & \cdots & P_{1n} \\ \vdots & \vdots & \vdots & \vdots \\ P_{n0} & P_{n1} & \cdots & P_{nn} \end{bmatrix}$$

2) To overcome the dependence, introduce a process in continuous time by letting the time spent in a transition from state *k* to state *l* to have Cdf $F_{k,l}(t)$.

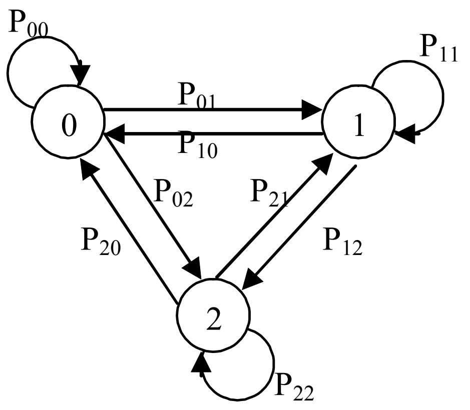
Such a process is attributed to a Semi-Markov Process.

6.2.1. Model for two failure states

When there are two failure states, there will be three states for the system after a run; a successful state, Type A failure state and Type B failure state. Type A failure could be a kind of serious failure such as Catastrophic or Critical failure. Type B failure could be less serious than Type A failure such as Minor or Marginal failure.

A common situation is that the system is not able to continue to perform its function when Type A failure occurs, but when Type B failure occurs, the system can still work, although it will have more chances to induce a Type A failure in the next run. For example, in a replicated file system, copies of the same file are kept in different servers so that failures of some servers can be tolerant in Chang (1999). If a minor failure does not destroy all the servers which contain the replicated files, the replicated file system is able to keep on working. Thus this kind of failure can be classified as Type B failure. Otherwise, if the replicated files of all the servers are corrupted by a failure to make the system down, this kind of failure is Type A failure.

The result from a run will affect the probable state in the next run as shown in Fig. 6.1. In this part, we consider the case when there is no debug except the resetting or restarting when Type A failure occurs. The transition probability will remain unchanged under this assumption.



State 0: Successful state after a run
State 1: Type A failure occurs after a run

State 2: Type B failure occurs after a run

Fig. 6.1. Markov interpretation of dependent runs.

Let Z_k be a random variable of the state after a run, and denote by

$$P_{mj} = P\{Z_{k+1} = j \mid Z_k = m\},\ m, j = 0,1,2$$

The transition matrix is

$$\mathbf{P}=\begin{bmatrix} P_{00} & P_{01} & P_{02} \\ P_{10} & P_{11} & P_{12} \\ P_{20} & P_{21} & P_{22} \end{bmatrix} \qquad (6.1)$$

in which

$$\sum_{j=0}^{2} P_{mj} = 1, \quad m\text{=0,1,2} \tag{6.2}$$

The unconditional probability of failure on run (i+1) is:

$$P\{Z_{i+1} = j\} = \sum_{m=0}^{2} P_{mj} P\{Z_i = m\}, \quad j\text{=0, 1, 2} \tag{6.3}$$

Substituting Eq. (6.3) into the above equation, we have that

$$P\{Z_{i+1} = j\} = (P_{2j} - P_{0j})P\{Z_i = 2\} + (P_{1j} - P_{0j})P\{Z_i = 1\} + P_{0j}, \quad j\text{=0,1,2} \tag{6.4}$$

we can see from this equation that if

$$P_{0j} = P_{1j} = P_{2j}, j\text{=0,1,2}$$

the results from two successive runs are independent. If the system does not satisfy this condition, it is dependent.

The next step is to develop a model in continuous time, considering the time that the system spends on running. Let $F_{k,l}(t)$ be a Cdf of the time spent in a transition from state k to state l of the DTMC in Fig. 6.1. Here, $F_{k,l}(t)$ is assumed to depend only on the state at the end of each interval in a system run, see e.g. Goseva-Popstojanova and Trivedi (2000) as:

$$F_{0,j}(t) = F_{1,j}(t) = F_{2,j}(t) = F_{\bullet j}(t), \quad j\text{=0,1,2}$$

With the addition of the $F_{\bullet j}(t)$ to the transitions of discrete time Markov chain, we obtain a Semi-Markov Process as the system reliability model in continuous time.

6.2.2. Model for two failure states with debugging

Furthermore, we assume that after a Type A failure, the system may be debugged and it is an instantaneous fault removing process. Hence, after removing the fault, the transition probability matrix will be changed. When the successive runs are successful or only cause the Type B failure, the system does not have to be debugged and it will continue running in the same way. The transition probability matrix is able to be assumed unchanged until a Type A failure happens. The Markov renewal model is modified as the Fig. 6.2.

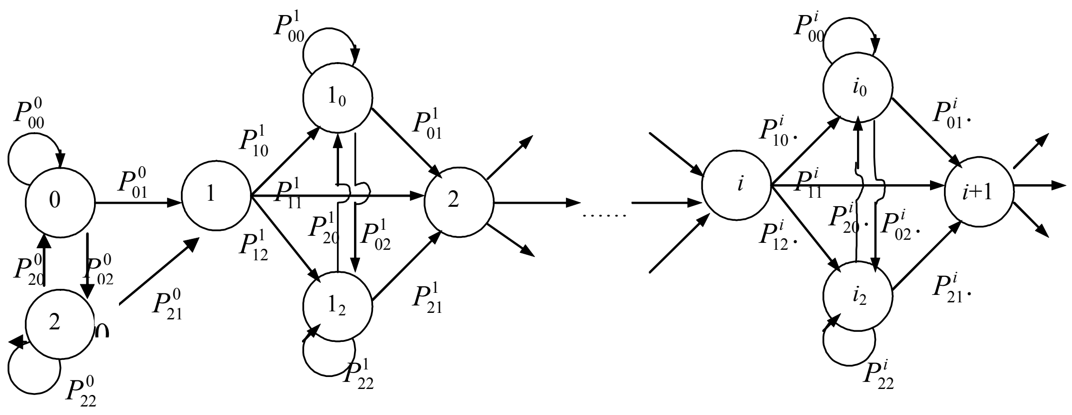


Fig. 6.2. Nonhomogeneous DTMC for system reliability model.

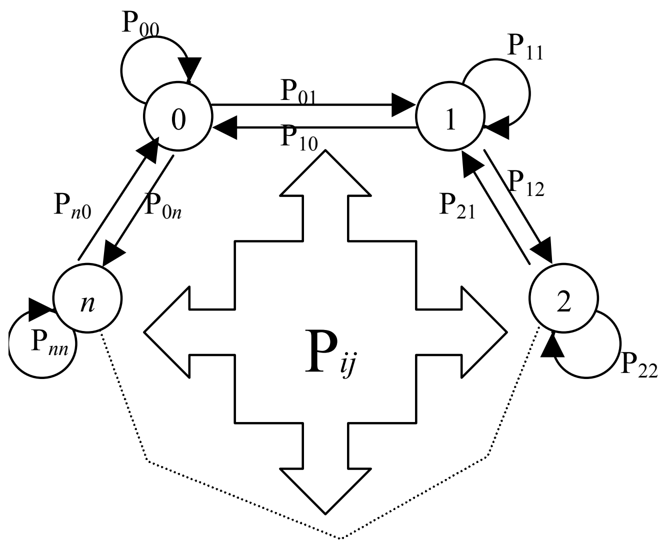
'*i*' is the number of Type A failures, which is already detected and removed. During the testing phase, system is subjected to a sequence of runs, making no changes if there is no Type A failure. When a Type A failure occurs on any run, then an attempt is made to fix the underlying fault, which causes the conditional probabilities of the state on the next run to change. The transition probability matrix for the period from the occurrence of the *i*:th Type A failure to the occurrence of the next (*i*+1):st Type A failure, is

$$\begin{bmatrix} P_{00}{}^{i} & P_{01}{}^{i} & P_{02}{}^{i} \\ P_{10}{}^{i} & P_{11}^{i} & P_{12}{}^{i} \\ P_{20}{}^{i} & P_{21}{}^{i} & P_{22}{}^{i} \end{bmatrix} \tag{6.5}$$

Assume S_m is the total number of Type A failures after *m* runs. The sequence S_m provides an alternate description of system reliability model with debugging process considered here. Thus, $\{S_m\}$ defines the DTMC presented in the above Fig. 6.2. All states, *i*, i_0 and i_2, represent that the Type A failure state has been occupied *i* times. State *i* represents the initial state for which $S_m = i$. State i_0 represents all the successful subsequent trials for which $S_m = i$, State i_2 represents all Type B failures subsequent trials for which $S_m = i$.

6.2.3. General model for *n* failure states

The above models can be extended to the case of general multi-state of failures. Assume that the failures can be divided into *n* states, so the system totally contains *n*+1 states including the perfect state. Denote again the critical failure type as Type A failure state. When this type of failures occurs, the system will completely stop working and action has to be taken. First we assume there are no changes in the system except resetting and restarting when Type A failure occurs. The transition probability matrix for the successive runs will remained unchanged. The Markov process can be expressed as the Fig. 6.3.



State 0: Successful state after a run
State 1: Type A failure occurs after a run

States 2 to *n*: The other *n*-1 types of failures occur after a run

Fig. 6.3. Markov interpretation for n-type correlated state transition.

Denote

$$P_{mj} = P\{Z_{k+1} = j \mid Z_k = m\},\ m, j = 0,1,2,\cdots,n$$

and the transition probability matrix is then

$$\mathbf{P}=\begin{bmatrix} P_{00} & P_{01} & P_{02}\cdots & P_{0n} \\ P_{10} & P_{11} & P_{12}\cdots & P_{1n} \\ P_{20} & P_{21} & P_{22}\cdots & P_{2n} \\ \vdots & \cdots & \cdots & \vdots \\ P_{n0} & P_{n1} & \cdots & \cdots P_{nn} \end{bmatrix}$$

and transition probabilities should satisfy

$$\sum_{j=0}^{n} P_{mj} = 1,\quad m=0,1,2,\ldots,n \tag{6.6}$$

The unconditional probability of failure on run (*i*+1) is:

$$P\{Z_{i+1} = j\} = \sum_{m=0}^{n} P_{mj} P\{Z_i = m\},\quad j=0, 1, 2,\ldots,n \tag{6.7}$$

Similar to the previous case of two types of failures, when there is a debugging after Type A failure, the transition probability matrix changes accordingly. The *n*-type failure states Markov renewal model can be constructed as depicted by Fig. 6.4.

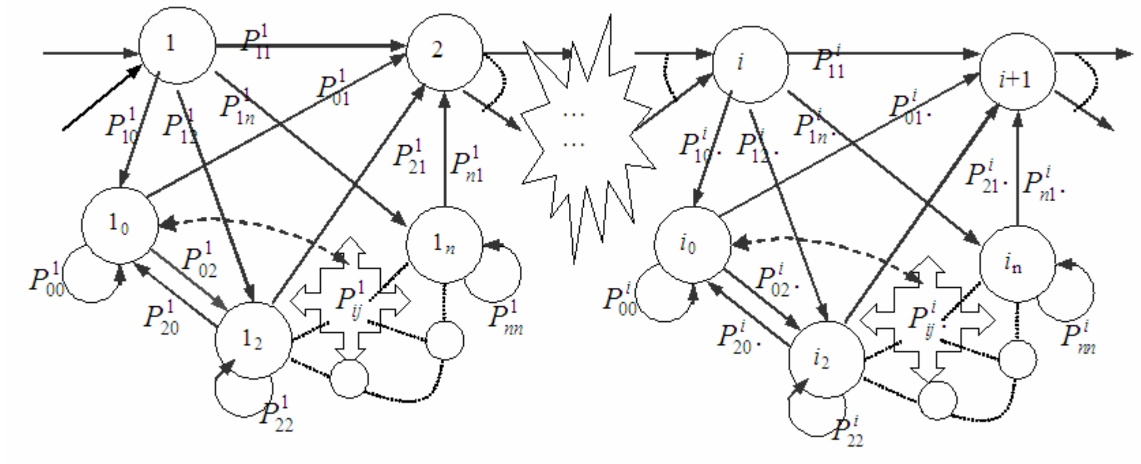


Fig. 6.4. Markov renewal model for *n*-type failure states.

As before, '*i*' in Fig. 6.4 is the number of Type A failure, which have already been detected and removed. The transition matrix for the period from the occurrence of the *i*:th Type A failure to the occurrence of the next (*i*+1)st Type A failure, is given as follow.

$$\begin{bmatrix} P_{00}{}^{i} & P_{01}{}^{i} & P_{02}{}^{i}\cdots & P_{0n}{}^{i} \\ P_{10}{}^{i} & P_{11}{}^{i} & P_{12}{}^{i}\cdots & P_{1n}{}^{i} \\ P_{20}{}^{i} & P_{21}{}^{i} & P_{22}{}^{i}\cdots & P_{2n}{}^{i} \\ \vdots & \cdots & \cdots & \vdots \\ P_{n0}{}^{i} & P_{n1}{}^{i} & \cdots & \cdots P_{nn}{}^{i} \end{bmatrix}$$

and the transition probability should satisfy

$$\sum_{j=0}^{n} P_{mj}{}^{i} = 1, \quad m=0,1,2,\ldots,n$$

Again $\{S_m\}$ defines the DTMC presented in above Fig. 6.4. All the states, *i*, i_0, i_2,…,

i_n, represent that the Type A failure state has been occupied *i* times. State *i* represents the first trial for which $S_m = i$. State i_0 represents all the successful subsequent trials for which $S_m = i$, State i_2 to i_n represents Type 2 to *n* failure states subsequent trials for which $S_m = i$.

6.3. Implementations of the model

The above Markov renewal model can be implemented to analyze the system behavior in both testing phase and validation phase. In testing phase, the system is debugged, so the transition probabilities should change after each Type A failure. However, between two Type A failures, the transition probabilities are constant, so the distribution of time between two successive Type A failures can be easily derived by using the Laplace-Stieltjes Transform. Thus, the conditional system reliability, which is defined as the survivor time distribution between two Type A failures, can also be obtained. Then, the mean time between failures can be easily computed by either integration or using the well known property of Laplace-Stieltjes transformation.

On the other hand, the probability transition matrix will be constant during the validation phase after the test, because no changes are made to the system during that phase. Thus, the system reliability can be easily calculated. Then, we propose a method to certify the system given certain confidence level based on the system reliability.

6.3.1. Some quantitative measures

From a reliability point of view, the time between failures or the number of failures over time is very important. Here, we derive the distribution of the discrete random variable X_{i+1}^j (j=0,2,3…n) defined as the number of runs visiting the j:th state between two successive visits from the i:th Type A failure to the (i+1)st Type A failure.

The probability of every possible number of X_{i+1}^j (j=0,2,3…,n) is given by

$$P\{X_{i+1}^j = K_j \big| j = 0,2,3...,n\} = \begin{cases} P_{11}^i & (\forall K_j = 0\big|_{j=0,2,3,...,n}) \\ g(K_0, K_2, K_3,..., K_n) & (\exists K_j \neq 0\big|_{j=0,2,3,...,n}) \end{cases} \tag{6.8}$$

in which $g(K_0, K_2, K_3,..., K_n)$ is the function of $K_0, K_2, K_3,..., K_n$, and K_j denotes the number of runs occupied on the j:th failure state (K_j=0,1,2…). The value of $g(K_0, K_2, K_3,..., K_n)$ can be obtained in principle.

Under the condition of that it visits the j:th state with K_j times (j=0,2,3,…,n) and that Type A failure occurs once between the i:th and (i+1):st Type A failures, the distribution of the time period used for this event can be derived as

$$F_{\bullet 0}^{K_0 *}(t) \otimes F_{\bullet 2}^{K_2 *}(t) \otimes F_{\bullet 3}^{K_3 *}(t) \cdots \otimes F_{\bullet n}^{K_n *}(t) \otimes F_{\bullet 1}(t) \tag{6.9}$$

in which $F_{\bullet j}^{K_j *}(t)$ is the K_j-fold convolution of $F_{\bullet j}(t)$ ($j = 0,2,3...,n$) and K_j can be 0,1,2…. Also, '$\otimes$' denotes the convolution of the two functions.

Define the distribution of time between the i:th and (i+1):st Type A failures as $F_{i+1}(t)$. Assume T_{i+1} is the random variable of time between the i:th and (i+1):st Type A failure runs. With the above two equations, it can be shown that the distribution of

$$\begin{aligned} F_{i+1}(t) &= P\{T_{i+1} \le t\} \\ &= \sum_{K_0=0}^{\infty}\sum_{K_2=0}^{\infty}\cdots\sum_{K_n=0}^{\infty} P\{X_{i+1}^{j} = K_j \,|\, j = 0,2,\ldots,n\} \cdot F_{\bullet 0}^{K_0 *}(t) \otimes F_{\bullet 2}^{K_2 *}(t) \cdots \otimes F_{\bullet n}^{K_n *}(t) \otimes F_{\bullet 1}(t) \end{aligned}$$

(6.10)

The Laplace-Stieltjes transform of $F_{i+1}(t)$ becomes

$$\widetilde{F}_{i+1}(s) = \sum_{K_0=0}^{\infty}\sum_{K_2=0}^{\infty}\cdots\sum_{K_n=0}^{\infty} P\{X_{i+1}^{j} = K_j \,|\, j = 0,2,\ldots,n\} \cdot \widetilde{F}_{\bullet 0}^{K_0}(s)\widetilde{F}_{\bullet 2}^{K_2}(s)\cdots\cdot\widetilde{F}_{\bullet n}^{K_n}(s)\widetilde{F}_{\bullet 1}(s)$$

(6.11)

The inversion of the above equation is straightforward, and reasonably simple closed-form results can be obtained when $F_{\bullet j}(t),(j = 1,2,...n)$ has a rational Laplace-Stieltjes transform.

The reliability of the system after *i*:th Type A failure is

$$R_{i+1}(t) = 1 - F_{i+1}(t) = P(T_{i+1} > t) \qquad (6.12)$$

Some general properties of the inter-failure time can be developed without making other assumptions. For example, the mean time between failures (*i* and *i*+1 Type A failures) is:

$$E[T_{i+1}] = \int_0^{\infty} R_{i+1}(t)dt \qquad (6.13)$$

or, see e.g. Goseva-Popstojanova and Trivedi (2000)

$$E[T_{i+1}] = -\frac{d\widetilde{F}_{i+1}(s)}{ds}\Big|_{s=0} \qquad (6.14)$$

6.3.2. Application to the validation phase

Now we discuss the system reliability in validation phases with two-type failure model.

After the testing (debugging) phase, the system enters a validation phase to show that it has a high reliability prior to actual use. In this phase, no changes are made to the system. Thus the probability transition matrix will not change as the Markov Process depicted by the previous Fig. 6.1.

Now we consider the independent condition, that is,

$$P_{0j} = P_{1j} = P_{2j} = P_{\bullet j}, j\text{=}0,1,2$$

If the state is not a Type A failure after a run, the system is reliable until the Type A failure occurs. The reliability in a run is $1 - P_{\bullet 1}$. The reliability for *m* successive runs is defined as the probability that *m* successive independent test runs are conducted without Type A failure, which can be derived as:

$$R(m) = (1 - P_{\bullet 1})^m = (P_{\bullet 0} + P_{\bullet 2})^m \tag{6.15}$$

Given a confidence level α, if $R(m) \geq \alpha$, we can say that the system is reliable in successive *m* runs without Type A failure with α confidence. In order to satisfy this condition, the value of $P_{\bullet 1}$ should make

$$(1 - P_{\bullet 1})^m \geq \alpha \tag{6.16}$$

Given a confidence level α, we can obtain an upper confidence bound on $P_{\bullet 1}$, which is denoted by P_1^*. Solving $(1 - P_{\bullet 1}^*)^m = \alpha$, we obtain the upper bound

$$P_{\bullet 1}^* = 1 - \alpha^{1/m} \tag{6.17}$$

This can help to test whether the system can be certified or not, i.e., if $P_{\bullet 1} \leq P_{\bullet 1}^*$, the system is certified with α confidence to say that the system is reliable in *n* successive runs without Type A failure. Otherwise, we cannot say that with α confidence.

Now consider a sequence of possibly dependent system runs. During the validation phase, the system is not changing, i.e., P_{ij} does not change. That is, the sequence of runs can be described by the homogeneous DTMC with the transition probability matrix. Assume that the DTMC is steady, i.e., each run has the same failure-probability:

$$P\{Z_{i+1}=j\}=P\{Z_i=j\}=\sum_{m=0}^{n}P_{mj}P\{Z_i=m\}\,,\quad j=0,1,2 \tag{6.18}$$

Let $P_j=P\{Z_i=j\}$ and substitute it into the above equation to get

$$P_j=\sum_{m=0}^{n}P_{mj}P_m\,,\quad j=0,1,2 \tag{6.19}$$

Solve the above equations to obtain unconditional probability of failure on run as

$$P_2=\frac{P_{01}P_{12}+P_{02}-P_{11}P_{02}}{(1-P_{11}+P_{01})(1-P_{22}+P_{02})-(P_{12}-P_{02})(P_{21}-P_{01})} \tag{6.20}$$

$$P_1=\frac{P_{02}P_{21}+P_{01}-P_{22}P_{01}}{(1-P_{22}+P_{02})(1-P_{11}+P_{01})-(P_{21}-P_{01})(P_{12}-P_{02})} \tag{6.21}$$

$$P_0=1-P_1-P_2 \tag{6.22}$$

The reliability for *m* successive runs after the system is steady will be

$$R(m)=(1-P_1)^m=(P_0+P_2)^m \tag{6.23}$$

Similarly, given the confidence level α, the largest value of P_1 so that

$$(1-P_1)^m\ge\alpha \tag{6.24}$$

is defined as the upper confidence bound, denoted by P_1^*. Solve the above equation for P_1^* given α, we have that

$$P_1^*=1-\alpha^{1/m} \tag{6.25}$$

Again, this can help to test whether the system is certified or not, i.e., if $P_1\le P_1^*$, the system is certified with confidence level α to say that the system is reliable in *m* successive runs without Type A failure. Otherwise, we cannot say that with confidence

Now we consider the system reliability in validation phases with *n*-type failure model as in Fig. 6.3. Assume that the DTMC is homogeneous, we have

$$P\{Z_{i+1} = j\} = P\{Z_i = j\} = \sum_{m=0}^{n} P_{mj} P\{Z_i = m\}, \quad j=0,1,2,\ldots,n \tag{6.26}$$

Let $P_j = P\{Z_i = j\}$ and substitute it into the above equation to get

$$P_j = \sum_{m=0}^{n} P_{mj} P_m, \quad j=0,1,2,\ldots,n$$

Solving the above equations, we can obtain unconditional probability of different states in principle. Then, we can analyze the properties with these values similarly as the two-type failure model analyzed above.

Although the analysis presented in the previous section seems to be complicated, it is numerically tractable. An example is given here to illustrate the procedure.

6.3.3. Illustrative example

Suppose the distribution of the execution time of each run is exponential so that

$$F_{\bullet j}(t) = 1 - \exp(-\mu_j t), j=0,1,2$$

It relates the MRP approach to the existing system reliability models. Its Laplace-Stieltjes transform is

$$\widetilde{F}_{\bullet j}(s) = \frac{\mu_j}{s + \mu_j}$$

Set the numerical values of $\mu_0 = 0.302, \mu_1 = 0.3, \mu_2 = 0.297$.

In the operational phase we can estimate the transition probability matrix from empirical data of successive runs. The following transition probability matrix is used as illustration.

$$\mathbf{P}=\begin{bmatrix} P_{00} & P_{01} & P_{02} \\ P_{10} & P_{11} & P_{12} \\ P_{20} & P_{21} & P_{22} \end{bmatrix}=\begin{bmatrix} 0.7 & 0.1 & 0.2 \\ 0.3 & 0.1 & 0.6 \\ 0.1 & 0.2 & 0.7 \end{bmatrix}$$

Substitute those values into Eq. (6.11), we can obtain the Laplace-Stieltjes transform equation and then invert it to get the Cdf of the time between failures as:

$$\begin{aligned} F(t) = 1 - 0.053e^{-0.3t} - 0.038e^{-0.302t} - 0.051e^{-0.297t} \\ -0.041e^{-0.55t} - 0.45e^{-0.05t} - 0.045e^{-0.51t} - 0.054e^{-0.342t} \\ -0.157e^{-0.259t} - 0.127te^{-0.09\cdot t} - 0.111e^{-0.09t} \end{aligned} \tag{6.27}$$

This equation implies that when successive runs are dependent, the Cdf of the time between failures is a mixture of exponential distributions. Fig. 6.5 displays the distribution of $F(t)$.

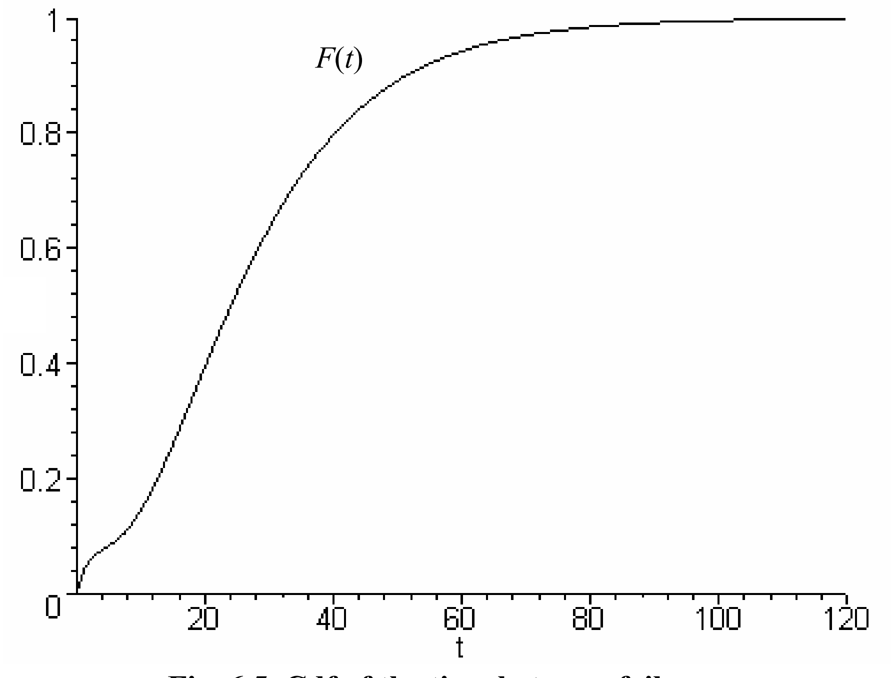


Fig. 6.5. Cdf of the time between failures.

The reliability is then calculated as

$$\begin{aligned} R(t) = {} & 0.053e^{-0.3t} + 0.038e^{-0.302t} + 0.051e^{-0.297t} \\ & +0.041e^{-0.55t} + 0.45e^{-0.05t} + 0.045e^{-0.51t} + 0.054e^{-0.342t} \\ & +0.157e^{-0.259t} + 0.127te^{-0.09\cdot t} + 0.111e^{-0.09t} \end{aligned} \tag{6.28}$$

Thereafter, the mean time to failure can be obtained as,

$$E[T] = \int_0^\infty R_{i+1}(t)dt = 27.31 \quad \text{(hours)} \tag{6.29}$$

When the system is in validation phase, no changes are made to the system. Hence, the probability transition matrix will not change. After some time, the system enters a steady-state. The unconditional probability of the three different states can be calculated through Eqs. (6.20-6.22)

$$P_2 = \frac{P_{01}P_{12} + P_{02} - P_{11}P_{02}}{(1 - P_{11} + P_{01})(1 - P_{22} + P_{02}) - (P_{12} - P_{02})(P_{21} - P_{01})} = 0.522$$

$$P_1 = \frac{P_{02}P_{21} + P_{01} - P_{22}P_{01}}{(1 - P_{22} + P_{02})(1 - P_{11} + P_{01}) - (P_{21} - P_{01})(P_{12} - P_{02})} = 0.152$$

$$P_0 = 1 - P_1 - P_2 = 0.326$$

These values reflect that with this transition probability matrix when the system is steady, the probability of successful state after a run is 0.326, the probability of a Type B failure occurs after a run is 0.552 and the probability of a Type A failure occurs after a run is 0.152. The steady probability for the system to be reliability is

$$P_0 + P_2 = 0.326 + 0.522 = 0.848$$

CHPATER 7

MULTI-STATE SYSTEMS WITH MULTI-LEVEL PROTECTIONS

In this chapter, we consider vulnerable systems which can have different states corresponding to different combinations of available elements composing the system. Each state can be characterized by a performance rate, which is the quantitative measure of a system's ability to perform its task. Both the impact of external factors (stress) and internal causes (failures) affect system survivability, which is determined as probability of meeting a given demand.

In order to increase the survivability of the system, a multilevel protection is applied to its subsystems. This means that a subsystem and its inner level of protection are in their turn protected by the protection of an outer level. This double-protected subsystem has its outer protection and so forth. In such systems, the protected subsystems can be destroyed only if all of the levels of their protection are destroyed. Each level of protection can be destroyed only if all of the outer levels of protection are destroyed.

We formulate the problem of finding the structure of series-parallel multi-state system (including choice of system elements, choice of structure of multilevel protection and choice of protection methods) in order to achieve a desired level of system survivability by the minimal cost. An algorithm based on the universal generating function method is used for determination of the system survivability. A multiprocessor version of genetic algorithm is used as optimization tool in order to solve the structure optimization problem. An application example is presented to illustrate the procedure presented in this chapter.

7.1. Introduction

Survivability is defined as the ability of a system to tolerate intentional attacks or accidental failures or errors. It is becoming important in the system performability (Smith *et al.*, 1988 and Haverkort *et al.*, 2001), especially when a system operates in battle conditions or is affected by a corrosive medium or other hostile environment. In this case both the impact of external factors (attack) and internal causes (failures) affect system survivability.

A system can have different states corresponding to different combinations of failed or damaged elements composing the system. Each state can be characterized by a system performance rate, which is the quantitative measure of a system's ability to perform its task. For example, in Malakhoff *et al.* (1998) and Parfenov (1989) each system state is characterized by an available ship propulsion power or by an available electric power respectively. Therefore a system should be considered a multi-state one when its survivability is analyzed, see e.g. Veeraraghavan and Trivedi (1994).

When applied to multi-state systems, mission success depends on a system's ability to meet the demand (required performance level). In this case, the outage effect will be essentially different for units with different nominal capacities and will also depend on demand. Therefore, the performance rates (productivity or capacity) of system elements should be taken into account as well as the level of demand when the survivability of the entire system is estimated.

To provide a required level of system availability, redundant elements can be included. Usually engineers try to achieve this level with minimal cost. The problem of total investment cost minimization, subject to reliability constraints, is well known as the redundancy optimization problem. The redundancy optimization problem for a system with different element performance rates may be considered as a problem of system structure optimization. The method for solving the structure optimization problem was suggested in Levitin (1998) and Levitin and Lisnianski (2001).

One of the ways to enhance system survivability is to separate elements with the same functionality (parallel elements). Adding more parallel elements will improve a MSS availability but will not be effective from a vulnerability standpoint without sufficient separation between elements Malakhoff *et al.* (1998). The separation can be performed by spatial dispersion, by encapsulating different elements into different protective casings etc.

Parallel elements not separated from one another are considered to belong to the same protection group. All elements belonging to the same protection group can be destroyed by the same impact while at least all the elements belonging to N-1 different protection groups out of N will survive a single impact. Obviously, separation has its price. Allocating all the parallel elements together (within a single protection group) is usually cheaper than separating them. The separation usually requires additional areas, constructions, communications etc. Moreover, each separated group can be intentionally protected against the external impact, which requires additional investments. There can be different methods of protection characterized by different vulnerability of the group and by different cost.

Since system elements with the same functionality can have different performance rates and different availability, the choice of elements to be included into the system strongly affects system survivability. Other factors that influence the survivability are the partitioning of elements into protection groups and the choice of protection method for each separated group.

The problem of finding structure of series-parallel MSS (including choice of system elements, their separation and protection) in order to achieve a desired level of system survivability by the minimal cost has been formulated in Levitin and Lisnianski (2003). To solve the problem an optimization procedure based on a Genetic Algorithm was suggested. In this problem each group of elements can have only one level of protection.

In real systems, a multilevel protection is often used (for example in defense-in-depth design methodology, Fleming and Silady 2002). The multilevel protection means that a subsystem and its inner level protection are in their turn protected by the protection of the outer level. This double-protected subsystem has its outer protection and so forth. In such systems, the protected subsystems can be destroyed only if all of the levels of their protection are destroyed. Each level of protection can be destroyed only if all of the outer levels of protection are destroyed. This creates the statistical dependence among events of destruction of different protection levels.

In systems consisting of nonidentical elements and having complex multilevel protection, different protections play different roles in providing for the system's survivability. Subject to cost limitations, one usually has to find a minimal cost configuration of protections that provides desired system survivability. The problem of choosing the optimal multilevel protection in series-parallel systems with a given structure and a given set of possible protections has been formulated in Levitin (2003) and solved using the GA.

In this chapter we generalise the structure optimization problem formulated in Levitin and Lisnianski (2003) to the case of multilevel protection. In this problem one has to

1) find the optimal system structure by choosing the appropriate product (version of a system element) from the list of available products for each type of equipment;

2) allocate the chosen elements among different protection groups and to define the hierarchy of the multilevel protection in the system;

3) choose the method of protection for each protection group.

The formulated problem is extremely complex combinatorial one. Even solving much simpler single-level protection problem (Levitin and Lisnianski, 2003) requires more than hour of computations for relatively small system. Therefore the use of the same computational approach for the multi-level protection case seems to be not realistic. In order to tackle the problem we developed a multiprocessor version of the GA. We also suggest the new solution encoding technique since the encodings presented in Levitin and Lisnianski (2003) or Levitin (2003) cannot define solutions of the formulated problem, which is more general.

In the following section, the system model is presented and the optimization problem is defined. An algorithm for evaluating the system survivability for arbitrary set of elements and protections is presented in section 7.3. Section 7.4 describes the multiple processor based optimization technique. In section 7.5 illustrative examples of the system survivability optimization problem are presented.

7.2. Model Description and Problem Formulation

Acronyms

PD	performance distribution
PG	protection group
MSS	multi-state system
GA	genetic algorithm
UGF	*universal generating function*

Nomenclature

C_{MSS}	total MSS cost
$d(z)$	double-u-function (d-function). Composition of u-functions $U(z)$ and $\widetilde{U}(z)$
E_m	maximal allowable number of elements within m-th component
$G_m(h)$	random performance rate of element of version h that can be included in component m
G_{MSS}	random output performance rate of the entire MSS
$g_{MSS,k}$	output performance rate of the entire MSS at state k

q_k	probability that the entire MSS is in state k
$g_{mk}(h)$	performance rate at state k of element of version h that can be included in component m
H_m	number of different versions of elements that can be included in m-th component
L_m	number of protection levels for m-th component
L_{comp}	number of protection levels for groups of serially connected components
$\boldsymbol{h_m}$	vector representing versions of elements belonging to component m
$K_m(h)$	number of different states of element of version h that can be included in component m
$p_{mk}(h)$	probability of state k of element of version h that can be included in component m
$\boldsymbol{r}^{\boldsymbol{e}}{}_{\mathrm{m}}$	vector representing e-level partition of PGs in component m
$\boldsymbol{r}^{\boldsymbol{e}}{}_{\mathrm{comp}}$	vector representing e-level partition of PGs consisting of serially connected components
S_{MSS}	MSS survivability index
S^*	desired MSS survivability
$U_j(z)$	u-function representing performance distribution of j-th subsystem belonging to some PG when the protection of this group is destroyed
$\tilde{U}_j(z)$	u-function representing performance distribution of j-th subsystem belonging to some PG when the protection of this group is not destroyed
$u_{mi}(z)$	u-function representing performance distribution of i-th element belonging to component m
w	system demand
$\gamma^e{}_{mj}$	method of e-level protection of j-th PG belonging to component m
$\gamma^e{}_{comp,j}$	method of e-level protection of j-th PG consisting of components connected in series
$\Gamma^e{}_m$	number of available methods for e-level protection in component m
$\Gamma^e{}_{comp}$	number of available methods for e-level protection of PGs consisting of components connected in series
$n^e{}_{mi}$	number of inner PGs in i-th e-level PG belonging to component m
$n^e{}_{comp,i}$	number of inner PGs in i-th e-level PG consisting of components connected in series
$\boldsymbol{\gamma^e{}_m}$	vector representing e-level protection methods of corresponding to groups of elements belonging to component m
$\boldsymbol{\gamma^e{}_{comp}}$	vector representing e-level protection methods of corresponding to groups of components connected in series
$v^e{}_m(\gamma)$	vulnerability of e-level protection of PG belonging to component m when protection method γ is used
$\varepsilon_m(h)$	cost of element of version h that can be included in component m

$c^{e}{}_{m}$	cost of *e*-level protection in component *m*
$c^{e}{}_{comp}$	cost of *e*-level protection of PG consisting of components connected in series
$\pi(d(z))$	operator incorporating protection vulnerability into d-function representing the performance distributions of PG
$\oplus$	composition operator over u-functions for parallel connection of elements
$\otimes$	composition operator over u-functions for series connection of elements

Consider a system consisting of *M* components connected in series. Each component contains elements connected in parallel. Different versions and number of elements may be chosen for any given system component. Elements are characterized by performance distributions and costs, according to their versions. The states of MSS elements are mutually statistically independent.

The MSS mission success is defined as its ability to meet a demand *W*. Therefore the system survivability is

$$S_{MSS}(w) = \Pr\{G_{MSS} \geq w\}, \tag{7.1}$$

where G_{MSS} is output performance of the MSS.

For MSS which have a finite number of states there can be *K* different levels of output performance: $G_{MSS} \in \{g_{MSS,k},\ 1 \leq k \leq K\}$ and system output performance distribution can be defined by two finite vectors $g=\{g_{MSS,k}\}$ and $q=\{q_k\}$, where $q_k=\Pr\{G=g_{MSS,k}\}$, $(1 \leq k \leq K)$. Therefore we can define MSS survivability as the probability that a system remains in those states in which $g_{MSS,k} \geq w$:

$$S_{MSS}(w) = \sum_{g_{MSS,k} \geq w} q_k. \tag{7.2}$$

For each component *m*, there exist a list of H_m different versions of available elements. A performance distribution $g_{mk}(h)$, $p_{mk}(h)$ $(1 \leq k \leq K_m(h))$ and cost $\varepsilon_m(h)$ can be specified for each version h of element of type m. The structure of system component m is defined, therefore, by a vector containing numbers of versions of elements chosen for the component $h_m=\{h_{m1},\ldots,h_{mEm}\}$, where $h_{mj}\in\{0,1,\ldots,H_m\}$. Note that including a dummy version 0 corresponding to the absence of elements allows one to represent a different number of elements included in component m by vectors h_m of the same length E_m. The total cost of elements chosen for the mth component is

$$C_m^{el}=\sum_{i=1}^{E_m}\varepsilon_m(h_{mi}). \tag{7.3}$$

The elements belonging to component m can be separated into E_m independent protection groups (some of these PGs can be empty and some can contain several elements). For example, the case when E_m-1 PGs are empty and one PG contains all the elements corresponds to gathering all of the elements within the same PG. The case when all E_m PGs are not empty corresponds to separation of all of the elements one from another. The partitioning among PGs can be represented by vector $r^1{}_{mj}$ ($1\le j\le E_m$), where $r^1{}_{mj}$ is a number of first-level PG to which element j belongs. The PGs are protected by first-level protection. For each protection group i within component m different methods of protection can be chosen $\gamma^1{}_{mi}\in\{0,\ldots,\Gamma^1{}_m\}$, where $\gamma^1{}_{mi}=0$ corresponds to absence of protection. The vector $\gamma^1{}_{mi}$ ($1\le i\le E_m$) defines the choice of the first-level protection methods in component m.

The E_m protected first-level PGs (that can be considered now as equivalent single elements) can be further separated into E_m second-level PGs and protected using methods $\gamma^2{}_{mj}\in\{0,\ldots,\Gamma^2{}_m\}$. These second-level protection groups can be further separated and protected by the third-level protection and so forth up to L_m protection level.

The vectors r^e_m: r^e_{mj} ($1\le j\le E_m$) and γ^e_m: γ^e_{mi} ($1\le i\le E_m$) determine the separation and protection of each level e. Having the vector r^e_m one can obtain the number of nonempty inner level PGs belonging to each PG i: n^e_{mi}.

Each e-level protection in component m that uses protection method γ can be destroyed with probability $v^e_m(\gamma)$. Therefore, the vulnerability of protection of e-level PG i is $v^e_m(\gamma^e_{mi})$. Observe that unprotected PG has vulnerability $v^e_m(0)=1$.

The protected subsystems can be destroyed only if all of the levels of their protection are destroyed. Each level of protection can be destroyed only if all of the outer levels of protection are destroyed.

The cost of protection of the e-level PG i depends on the type of elements protected (number of component m), on the number of nonempty inner PGs belonging to the given PG n^e_{mi} and on the chosen protection method γ^e_{mi}. This cost can be expressed as $c^e_m(n^e_{mi}, \gamma^e_{mi})$, where $c^e_m(0, \gamma^e_{mi})=0$ by definition.

Assume that component m has L_m protection levels. Each level has E_m PGs (some of these groups can be empty). The total cost of protections in the component m is:

$$C_m^{prot} = \sum_{e=1}^{L_m} \sum_{i=1}^{E_m} c^e{}_m(n^e{}_{mi}, \gamma^e{}_{mi}). \tag{7.4}$$

In a similar way one can define the cost protection of components connected in series C_{comp}^{prot}. Considering each component as equivalent element that can belong to any PG one can define the partition of components among e-level PGs using vector r^e_{comp} and the methods of protection for these PGs using vector γ^e_{comp} (note that the number of PGs on this level can be not greater than M). The vectors r^e_{comp} determine number of components in each PG i: $n^{e}_{comp,i}$.

Having the vectors r^{e}_{m} and γ^{e}_{m} for $1\le e\le L_m$ and $1\le m\le M$, the vectors r^{e}_{comp} and γ^{e}_{comp} for $1\le e\le L_{comp}$ and the vectors h_m for $1\le m\le M$ one can determine the structure of the entire system. The total MSS cost can be determined as

$$C_{MSS}(\boldsymbol{h},\boldsymbol{r},\boldsymbol{\gamma}) = \sum_{m=1}^{M}(C_m^{el}+C_m^{prot})+C_{comp}^{prot} =$$
$$\sum_{m=1}^{M}\Big[\sum_{j=1}^{E_m}\varepsilon_m(h_{mj})+\sum_{e=1}^{L_m}\sum_{i=1}^{E_m}c^{e}{}_{m}(n^{e}{}_{mi},\gamma^{e}{}_{mi})\Big]+\sum_{e=1}^{L_{comp}}\sum_{i=1}^{M}c^{e}{}_{comp}(n^{e}{}_{comp,i},\gamma^{e}{}_{comp,i})].$$

(7.5)

where

$$\boldsymbol{h}=\{\boldsymbol{h}_1,...,\boldsymbol{h}_{\mathrm{M}}\},\ \ \boldsymbol{r}=\{\boldsymbol{r}_1^{1},...,\boldsymbol{r}_1^{\mathrm{L}1},...,\boldsymbol{r}_{\mathrm{M}}^{1},...,\boldsymbol{r}_{\mathrm{M}}^{\mathrm{L}_\mathrm{M}},\boldsymbol{r}_{\mathrm{comp}}^{1},...,\boldsymbol{r}_{\mathrm{comp}}^{\mathrm{L}_\mathrm{comp}}\},$$
$$\boldsymbol{\gamma}=\{\gamma_1^{1},...,\gamma_1^{\mathrm{L}1},...,\gamma_{\mathrm{M}}^{1},...,\gamma_{\mathrm{M}}^{\mathrm{L}_\mathrm{M}},\gamma_{\mathrm{comp}}^{1},...,\gamma_{\mathrm{comp}}^{\mathrm{Lcomp}}\}.$$

Now we can formulate the optimal separation problem as follows.

Find sets h, r and γ that provide the desired MSS survivability S_{MSS} with the minimal cost:

$$(\boldsymbol{h},\boldsymbol{r},\boldsymbol{\gamma})=\arg\{C_{MSS}(\boldsymbol{h},\boldsymbol{r},\boldsymbol{\gamma})\to\min\mid S_{MSS}(\boldsymbol{h},\boldsymbol{r},\boldsymbol{\gamma},w)\ge S^{*}\}. \qquad (7.6)$$

7.3. Survivability Estimated by Universal Generating Function

To estimate the survivability of MSS with given structure, separation and protection of elements $S_{MSS}(h,r,\gamma)$ one has to apply a procedure which calculates the performance distribution of a given series-parallel structure. The procedure used in this chapter for the system survivability evaluation is based on the universal generating function (u-function) technique, which was introduced in Ushakov (1986) and which proved to be very effective for the reliability evaluation of different types of multi-state systems Lisnianski and Levitin (2003). The detailed description of the procedure can be found in Lisnianski and Levitin (2003). A brief description of this procedure is given in the following.

For the sake of simplicity, we consider only those MSS in which the performance measure is defined as productivity or capacity (continuous materials or energy transmission systems, manufacturing systems, power supply systems).

7.3.1. U-functions of individual elements and their parallel compositions

The u-function of a discrete random variable X is defined as a polynomial given by

$$u(z) = \sum_{k=1}^{K} a_k z^{x_k}, \tag{7.7}$$

where the variable X has K possible values and a_k is the probability that X is equal to x_k. To evaluate the probability that the random variable X is not less than the value w the coefficients of polynomial $u(z)$ should be summed for every term with $x_k \geq w$:

$$\Pr\{X \geq w\} = \sum_{x_k \geq w} a_k. \tag{7.8}$$

In our case, the polynomial $u_{mj}(z)$ can define performance distribution of j-th element belonging to component m, i.e. it represents all of the possible states of the element by relating the probabilities of each state to the performance of the element in that state. Note that the performance distribution of the chosen element (element of version h_{mj}) defined by the vectors $\{g_{mk}(h_{mj}), 1 \leq k \leq K_m(h_{mj})\}$ and $\{p_{mk}(h_{mj}), 1 \leq k \leq K_m(h_{mj})\}$ can now be represented as

$$u_{mj}(z) = \sum_{k=1}^{K_m(h_{mj})} p_{mk}(h_{mj}) z^{g_{mk}(h_{mj})}. \qquad (7.9)$$

Any subsystem consisting of two elements can be considered as a single equivalent element with a performance distribution equal to the performance distribution of the subsystem (represented by u-function obtained by the corresponding composition operator over u-functions of the two elements).

In MSS of a considered type, the total performance rate of a pair of elements connected in parallel is equal to the sum of the performance rates of the individual elements. To obtain the u-function $U(z)$ of a subsystem containing two parallel elements i and j, composition operator $\oplus$ is introduced:

$$U(z) = u_{mi}(z) \oplus u_{mj}(z) = \sum_{k=1}^{K_m(h_{mi})} p_{mk}(h_{mi}) z^{g_{mk}(h_{mi})} \oplus \sum_{f=1}^{K_m(h_{mj})} p_{mf}(h_{mj}) z^{g_{mf}(h_{mj})} =$$
$$= \sum_{k=1}^{K_m(h_{mi})} \sum_{f=1}^{K_m(h_{mj})} p_{mk}(h_{mi}) p_{mf}(h_{mj}) z^{g_{mk}(h_{mi}) + g_{mf}(h_{mj})}$$

(7.10)

7.3.2. U-functions of parallel PGs and their compositions

The probability of each state of a protected element (or subsystem) depends on the state of the protection. Therefore, each subsystem belonging to some PG is characterized by two conditional performance distributions: first corresponding to the case when the protection of the PG is destroyed and second corresponding to the case when the protection of the PG survives. In order to represent the performance distributions of an element j belonging to some PG within the component m we introduce the following double u-function (d-function) $d_{mj}(z)=<U_{mj}(z),\widetilde{U}_{mj}(z)>$, where $U_{mj}(z)$ and $\widetilde{U}_{mj}(z)$ represent performance distributions for the first and second cases respectively.

Note that if the protection of a basic single element is destroyed, this element is destroyed with probability 1 and has performance rate 0. Therefore for a basic single element j of component m that has a performance distribution represented by the u-function $u_{mj}(z)$

$$d_{mj}(z)=<z^0, u_{mj}(z)>. \tag{7.11}$$

It can be easily seen that any pair of parallel elements with d-functions $d_{mj}(z)$ and $d_{mi}(z)$ belonging to the same PG can be replaced by the equivalent element with d-function

$$\begin{aligned} d_{mj}(z)\oplus d_{mi}(z) &= \langle U_{mj}(z),\widetilde{U}_{mj}(z)\rangle \oplus \langle U_{mi}(z),\widetilde{U}_{mi}(z)\rangle \\ &=\langle U_{mj}(z)\oplus U_{mi}(z),\widetilde{U}_{mj}(z)\oplus\widetilde{U}_{mi}(z)\rangle \end{aligned} \tag{7.12}$$

Observe that any e-level PG x consists of a subsystem and its protection. The subsystem can always be replaced by its equivalent element with d-function $d_{mx}(z)=<U_{mx}(z),\widetilde{U}_{mx}(z)>$. The protection of the PG x has vulnerability $v^e{}_m(\gamma^e{}_{mx})$. If the protection survives (with probability $1-v^e{}_m(\gamma^e{}_{mx})$), the subsystem has its performance distribution represented by the u-function $\widetilde{U}_{mx}(z)$. If the protection is destroyed (with the probability $v^e{}_m(\gamma^e{}_{mx})$, the subsystem has its performance distribution represented by the u-function $U_{mx}(z)$, Therefore, the performance distribution of the entire PG (subsystem and its protection) can be obtained as

$$v^e{}_m(\gamma^e{}_{mx})U_{mx}(z)+(1-v^e{}_m(\gamma^e{}_{mx}))\widetilde{U}_{mx}(z). \tag{7.13}$$

If the PG x with its protection are in their turn protected by an e+1- level protection (which means that they belong to another PG), in the case when the outer protection survives, the inner protection also survives and the subsystem performance distribution is represented by the u-function $\tilde{U}_{mx}(z)$.

These considerations allow one to replace a subsystem x with d-function $d_{mx}(z)=<U_{mx}(z), \tilde{U}_{mx}(z)>$ and its protection with vulnerability $v^e_m(\gamma^e_{mx})$ by the equivalent unprotected element y with d-function obtained by applying the following operator π over $d_{mx}(z)$:

$$d_{my}(z)=\pi(d_{mx}(z))=\pi\langle U_{mx}(z),\tilde{U}_{mx}(z)\rangle$$

$$=\langle v^e_m(\gamma^e_{mx})U_{mx}(z)+(1-v^e_m(\gamma^e_{mx}))\tilde{U}_{mx}(z),\tilde{U}_{mx}(z)\rangle \quad (7.14)$$

7.3.3. U-functions of system consisting of components connected in series

Applying the operators (7.12) and (7.14) one can obtain d-functions $d_m(z)$ $1\le m\le M$ corresponding to entire components in the form

$$d_m(z)=(U_m(z),\tilde{U}_m(z))=(\sum_{k=1}^{K_m}a_{mk}z^{s_{mk}},\sum_{k=1}^{\tilde{K}_m}\tilde{a}_{mk}z^{\tilde{s}_{mk}}). \quad (7.15)$$

When the components are connected in series, the component with the lowest performance rate becomes the bottleneck of the subsystem. Therefore, for a pair of components connected in series the performance rate of the subsystem is equal to the minimum of the performance rates of the individual elements.

Therefore, the composition operators $\otimes$ defined for the series connection of a pair of components takes the form

$$U_m(z)\otimes U_l(z)=\sum_{k=1}^{K_m} a_{mk}z^{s_{mk}}\otimes\sum_{f=1}^{K_l} p_{lf}z^{s_{lf}}=\sum_{k=1}^{K_m}\sum_{f=1}^{K_l} a_{mk}a_{lf}z^{\min\{s_{mk},s_{lf}\}}\ .$$

(7.16)

Using the approach presented in section 3.2 and replacing in (7.12) operator ⊕ with operator ⊗ one can obtain the d-function of PGs consisting of components connected in series.

7.3.4. Algorithm for MSS survivability evaluation

Consecutively applying the operators presented in the previous section and replacing pairs of elements belonging to the same PG and a single element with protection by equivalent unprotected elements one can obtain the d-function representing the performance distribution of the entire system. The following recursive algorithm obtains the system survivability:

1. For each component m ($1\le m\le M$):

   1.1. According to the vector h_m determine d-functions of elements chosen for the component using (7.9) and (7.11).

   1.2. For each protection level e ($1\le e\le Lm$):

      1.2.1. According to the vectors r^e_m and γ^e_m determine protection groups and survivability of each protection.

      1.2.2. If the component contains a pair of unprotected elements belonging to the same PG, replace this pair with an equivalent element with d-function obtained by ⊕ operator (7.10) using Eq. (7.12).

      1.2.3. If some PG contains more than one element, return to step 1.2.2.

1.2.4. If the component contains PG consisting of a single element, replace this element and its protection with a single equivalent element with d-function obtained using Eq. (7.14).

Obtain the d-function of the entire component consecutively applying Eq. (7.12) over d-functions of PGs of highest level L_m.

2. For each protection level e ($1 \le e \le Lcomp$):

   2.1. According to the vectors $r^e{}_{comp}$ and $\gamma^e{}_{comp}$ determine protection groups and survivability of each protection.

   2.2. If the system contains a pair of unprotected components belonging to the same PG, replace this pair with an equivalent component with d-function obtained by substituting the $\otimes$ operator (7.16) into Eq. (7.12).

   2.3. If some PG contains more than one component, return to step 2.2.

   2.4. If the system contains PG consisting of a single component, replace this component and its protection with a single equivalent component with d-function obtained using Eq. (7.13).

3. Determine the d-function of the entire series-parallel system as the d-function of the remaining single equivalent element obtained by consecutively applying Eq. (7.12) over d-functions of PGs of highest level L_{comp}. The system performance distribution is represented by the first u-function of this d-function.

4. Obtain the system survivability for the given demand w by applying the Eq. (7.8) over the u-function representing the entire system performance

distribution.

7.4. Optimization Technique

Equation (7.6) formulates an extremely complicated combinatorial optimization problem. An exhaustive examination of all possible solutions is not realistic, considering reasonable time limitations. Even much simpler optimization problem (Levitin and Lisnianski, 2003) required considerable computational time for its solution. In order to solve the optimization problem (7.6) using the same Genetic Algorithm approach (Goldberg, 1989) that was applied for solving the problems in Levitin and Lisnianski (2003) and Levitin (2003), a multiprocessor version of the GA was developed.

7.4.1. Multiprocessor genetic algorithm

The basic structure of the multiprocessor version of GA based on GENITOR method (Whitley, 1989), is as follows:

Solutions in the GA are coded as a finite length strings. The main server maintains the population of encoded solutions and values of their fitness. The processors perform various procedures over individual solutions and their pairs and return the obtained solutions to the server.

An initial population of N_S solutions (strings) is generated at random by the processors. Each processor generates solutions, decodes them and evaluates their objective function (fitness). These values, which are a measure of quality, are used to compare different solutions. Then it sends the solution encoding strings and the corresponding values of fitness to the server.

During the genetic cycle the server chooses at random pairs of solutions and sends them to processors. In each processor, new solutions are obtained by using the crossover operator. This operator produces an offspring from the pair of parent solutions. The newly obtained offspring undergoes mutation with the probability P_{mut}. Each new solution is decoded and its fitness value is estimated. The new solution with its fitness value returns to the server.

The server accomplishes a selection procedure that determines which solution is better: the newly obtained solution or the worst solution in the population. The better solution joins the population, while the other is discarded. If the population contains equivalent solutions following selection, redundancies are eliminated and the population size decreases as a result.

After new solutions are produced N_{rep} times, new randomly constructed solutions are generated by the processors to replenish the shrunken population, and a new genetic cycle begins.

The GA is terminated after N_c genetic cycles. The final population contains the best solution achieved. It also contains different near-optimal solutions which may be of interest in the decision-making process.

To apply the genetic algorithm to a specific problem, a solution representation and decoding procedure must be defined as well as basic GA procedures and parameters.

7.4.2. Solution Representation

As described in the previous section, three things determine the structure of MSS:

the list of version numbers of elements chosen for each component $\boldsymbol{h} = \{\boldsymbol{h}_1,...,\boldsymbol{h}_\mathrm{M}\}$ partition of elements and lower level PGs between PGs of each level within each component and partition of components and lower level PGs between PGs consisting of serially connected components $\boldsymbol{r} = \{\boldsymbol{r}_1^1,...,\boldsymbol{r}_1^{\mathrm{L}1},...,\boldsymbol{r}_\mathrm{M}^1,...,\boldsymbol{r}_\mathrm{M}^{\mathrm{L}\mathrm{M}},\boldsymbol{r}_\mathrm{comp}^1,...,\boldsymbol{r}_\mathrm{comp}^{\mathrm{L}\mathrm{comp}}\}$ and methods of protection for each PG $\boldsymbol{\gamma} = \{\boldsymbol{\gamma}_1^1,...,\boldsymbol{\gamma}_1^{\mathrm{L}1},...,\boldsymbol{\gamma}_\mathrm{M}^1,...,\boldsymbol{\gamma}_\mathrm{M}^{\mathrm{L}\mathrm{M}},\boldsymbol{\gamma}_\mathrm{comp}^1,...,\boldsymbol{\gamma}_\mathrm{comp}^{\mathrm{Lcomp}}\}$.

The sets h, r and γ completely determine the system structure. This three sets can be combined within an integer string $\Theta=\{\Theta_1,\ldots,\Theta_M,\Theta_{comp}\}$, where Θ_m ($1\le m\le M$) is a concatenation of the vectors h_m, $r^e{}_m$ and $\gamma^e{}_m$ (for $1\le e\le L_m$) and Θ_{comp} is a concatenation of the vectors $r^e{}_{comp}$ and $\gamma^e{}_{comp}$ (for $1\le e\le L_{comp}$). Each one of the vectors h_m, $r^e{}_m$ and $\gamma^e{}_m$ has E_m elements, while each one of vectors $r^e{}_{comp}$ and $\gamma^e{}_{comp}$ has M elements. Therefore the total length of the string representing the solution is

$$\sum_{m=1}^{M}(E_m + 2L_m \cdot E_m) + 2M \cdot L_{comp}\,. \tag{7.17}$$

Note that all of the string values make sense only in the case when each component has maximal possible number of elements and all of these elements are separated one from another (all of the PGs are not empty). Otherwise some of values of the string should be ignored by the decoding procedure.

In the feasible solution the values of substring h_m should be distributed in the range (0, H_m), the values of substrings $r^e{}_m$ and $r^e{}_{comp}$ should be distributed in the range (1, E_m) and (1, M) respectively and the values of substrings $\gamma^e{}_m$ and $\gamma^e{}_{comp}$ should be distributed in the range (0,$\Gamma^e{}_m$) and (0,$\Gamma^e{}_{comp}$).

In order to allow all the string elements distributed within the same range to represent feasible solutions, we determine this range as

$$(0, \max\{M, \max_{1\le m\le M}\{H_m, E_m, \max_{1\le e\le L_m} \Gamma^e_m\}, \max_{1\le e\le L_{comp}} \Gamma^e_{comp}\}). \qquad (7.18)$$

When the string is decoded, we transform each string elements x, corresponding to substrings h_m, $r^e{}_m$, $r^e{}_{comp}$, $\gamma^e{}_m$ and $\gamma^e{}_{comp}$ respectively in the following way:

$$h_{mj} = \mathrm{mod}_{H_m+1}(x), r^e{}_{mj} = \mathrm{mod}_{E_m}(x)+1, r^e{}_{comp,j} = \mathrm{mod}_M(x)+1$$
$$\gamma^e{}_{mi} = \mathrm{mod}_{\Gamma^e{}_m+1}(x), \gamma^e{}_{mi} = \mathrm{mod}_{\Gamma^e{}_{comp}+1}(x). \qquad (7.19)$$

The unification of the distribution range of all the string elements simplifies the string generation procedure, as well as mutation and crossover operators.

Consider, for example, a series-parallel MSS with M=2, H_1=H_2=3, E_1=E_2=3, L_1=2, L_2=1, L_{comp}=2, $\Gamma^1{}_1$=3, $\Gamma^2{}_1$=2, $\Gamma^1{}_2$=3, $\Gamma^1{}_{\mathrm{comp}}$=$\Gamma^2{}_{\mathrm{comp}}$=2. In this example we will use the notation PG$n^e{}_m$ to designate nth PG of level e within mth component.

According to (7.18) the solution encoding string should consist of integer numbers distributed in the range (0, 3). Consider for example the following string obtained after transformation (7.19):

Θ: 1313322321221122021112312021122

In the first part of this string (Θ_1) the substrings 131, 332, 232, 122, and 112 represent h_1, $r^1{}_1$, $\gamma^1{}_1$, $r^2{}_1$ and $\gamma^2{}_1$ respectively. The versions of elements chosen to fill positions 1, 2 and 3 of the first component are according to h_1 1, 3 and 1 respectively. According to $r^1{}_1$ elements located at positions 1 and 2 belong to PG$3^1{}_1$, element located in position 3 belongs to PG$2^1{}_1$ (PG$1^1{}_1$ is empty). According to $\gamma^1{}_1$ the PG$2^1{}_1$ has protection method 3 and the PG$3^1{}_1$ has protection method 2 (the first number of $\gamma^1{}_1$ is ignored because PG$1^1{}_1$ is empty). Substring $r^2{}_1$ defines the distribution of first-level PGs among the PGs of the second level. According to $r^2{}_1$ PG$2^1{}_1$ and PG$3^1{}_1$ belong to PG$2^2{}_1$ (fist element of $r^2{}_1$ is ignored because PG$1^1{}_1$ is empty). PG$1^2{}_1$ and PG$3^2{}_1$ remain empty. The protection method for PG$2^2{}_1$ according to second element of $\gamma^2{}_1$ is 1 (the first and third elements of $\gamma^2{}_1$ are ignored).

In the second part of string (Θ_2) substrings 202, 111, 123 represent h_2, $r^1{}_2$, $\gamma^1{}_2$ respectively. Two elements of version 2 are chosen to fill positions 1 and 3 of the second component according to h_2 (0 corresponds to absence of any element). According to $r^1{}_2$ elements located at positions 1 and 3 belong to PG$1^2{}_2$ (PG$2^2{}_2$ and PG$3^2{}_2$ remain empty). According to $\gamma^1{}_2$ the PG$1^2{}_2$ has protection method 1 (the second and third numbers of $\gamma^1{}_2$ are ignored because PG$2^2{}_2$ and PG$3^2{}_2$ are empty).

In the last part of string (Θ_{comp}) substrings 12, 02, 11, 22 represent $r^1{}_{\text{comp}}$, $\gamma^1{}_{\text{comp}}$, $r^2{}_{\text{comp}}$ and $\gamma^2{}_{\text{comp}}$ respectively. According to $r^1{}_{\text{comp}}$ component 1 belongs to PG$1^1{}_{\text{comp}}$ and component 2 belongs to PG$2^1{}_{\text{comp}}$. According to $\gamma^1{}_{\text{comp}}$ the PG$1^1{}_{\text{comp}}$ has no protection (protection method 0) and the PG$2^1{}_{\text{comp}}$ has protection method 2. According to $r^2{}_{\text{comp}}$ both PG$1^1{}_{\text{comp}}$ and PG$2^1{}_{\text{comp}}$ belong to PG$1^2{}_{\text{comp}}$. According to $\gamma^2{}_{\text{comp}}$ this PG has protection method 2 (the second number of $\gamma^2{}_{\text{comp}}$ is ignored because PG$2^2{}_{\text{comp}}$ is empty).

One can see the structure of the system encoded by the given string in Fig. 1. (In this figure each system element is marked by its version number, protection denoted by ellipse numbered according to chosen protection method).

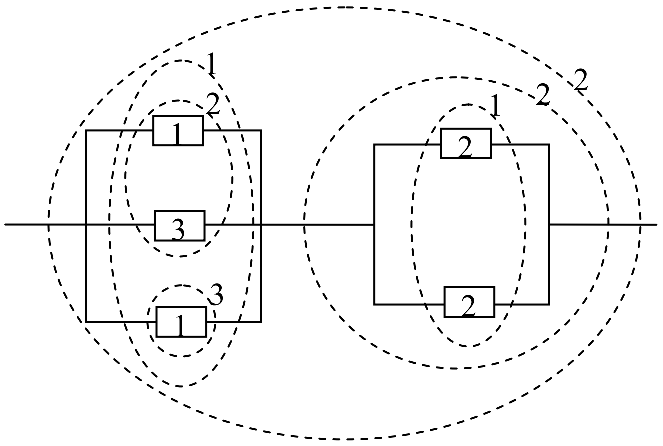


Fig. 7.1. MSS structure encoded by the integer string.

7.4.3. Solution decoding procedure

The following procedure determines the fitness value for an arbitrary solution defined by integer string Θ.

Assign 1 to the number of component m. Assign 0 to the total cost C_{MSS}.

Decode substring Θ_m and obtain versions of elements belonging to the component m, structure of protection groups ang corresponding protection methods at each protection level. Determine cost and performance distributions of elements in accordance with their versions and define d-functions of these elements using Eq. (7.9) and (7.11).

Calculate the cost of elements and protections in the component m using Eq. (7.3) and (7.4) and add this cost to C_{MSS}.

For each protection level e (from e=1 to e=L_m), obtain d-functions of PGs and replace them by equivalent elements using operators (7.10), (7.12) and (7.14).

Obtain the d-function of the m-th component using Eq. (7.12) with operator (7.10) over d-functions of nonempty PGs of L_m level.

Increment m and if $m \le M$ return to step 2.

Decode substring Θ_{comp} and obtain structure of protection groups ang corresponding protection methods at each protection level.

Calculate the cost of protections of serially connected components and add this cost to C_{MSS}.

For each protection level e (from e=1 to e=L_{comp}), obtain d-functions of PGs and replace them by equivalent elements using operators (7.16), (7.12) and (7.14).

Obtain the d-function of the entire MSS using Eq. (7.12) with operator (7.16) over d-functions of nonempty PGs of L_{comp} level.

Evaluate the system survivability S_{MSS} for the given demand w using Eq. (7.8) over the first u-function of d-function corresponding to the entire MSS.

So that the genetic algorithm will search for the solution with minimal total cost and with survivability not less than the required value S^*, evaluate the solution quality (fitness) Λ as follows:

$$\Lambda = a \cdot \exp\{-b \cdot [C_{MSS} + \lambda \cdot \min(S^* - S_{MSS}, 0)]\}, \qquad (7.20)$$

where λ is a sufficiently large penalty and a and b are positive constants. Note that for solutions with $S_{MSS} \ge S^*$, the fitness of the solution depends only on the system cost.

7.4.4. Crossover and mutation procedures

The crossover operator is aimed at producing a new solution (string) which inherits some properties of both parent solutions by combining parts of their strings.

String elements belonging to the fragment defined by two randomly chosen crossover sites are copied from the first parent and elements located out of the fragment, from the second parent.

The following example presents offspring string, obtained by the procedure:

First parent:	3 0 5 |1 4 7 2 0 1 | 1 0 0 0 2
Second parent:	*2 0 1 | 6 0 2 1 4 3| 4 9 1 3 0*
Crossover offspring:	*2 0 1* |1 4 7 2 0 1 |*4 9 1 3 0*

Mutation is aimed at maintaining a diversity of solutions by providing slight changes in the structure of the offsprings obtained by the crossover. This procedure avoids premature convergence to a local optimum and facilitates jumps in the solution space. The mutation operator replace a randomly chosen element of the solution encoding string by the random number belonging to the same range.

7.4.5. Choice of the GA parameters

In order to select the GA parameters that provide the fastest algorithm convergence to the best solutions the tests were performed on a set of 10 randomly generated problems with $M \le 5$, $\sum_{m=1}^{M} H_m \le 20$, $\sum_{m=1}^{M} L_m + L_{comp} \le 5$, $\sum_{m=1}^{M} \gamma_m + \gamma_{comp} \le 15$. According to the methodology suggested in Levitin and Lisnianski (2003), the fitness of the obtained solutions and the GA running time were compared. The chosen combination the GA parameters is N_s=200, N_{rep}=2000, N_c=100, P_{mut}=1.

7.5. Illustrative Example

In order to test the suggested algorithm it was compared with the procedure suggested in Levitin and Lisnianski (2003) for finding optimal structure of an MSS with single level protection. In illustrative example presented in Levitin and Lisnianski (2003) the series-parallel multi-state system (power substation) was considered. The parameters of this system are M=4, H_1=6, H_2=3, H_3=H_4=4, L_1=L_2=L_3=L_4=1, L_{comp}=0, $\Gamma^1{}_1$=3, $\Gamma^1{}_2$=1, $\Gamma^1{}_3$=3, $\Gamma^1{}_4$=2.

7.5.1. Minimal cost MSS with single-level protection

The cost, performance rate and availability of elements that can be included in each component are presented in Table 7.1. (All the elements have two states: normal functioning with performance g and total failure with performance 0. The probability of normal state is A, the failure probability is 1-A). For the sake of simplicity the cost of protection for each PG does not depend on the number of elements it contains, but does depend on the method of protection. The descriptions and costs of different protection levels available for each component and the vulnerabilities corresponding to these protection levels are presented in Table 7.2. The system should meet demand w=5.

Table 7.1. Parameters of available elements

No of Component	No of Version	g	A	ε
1	1	1.2	0.97	3.1
	2	1.6	0.92	4.2
	3	1.8	0.94	4.7
	4	2.0	0.93	5
	5	5.0	0.86	11
	6	5.0	0.91	14.5
2	1	1.8	0.98	3.1
	2	3.6	0.98	6
	3	5.4	0.96	8.8
3	1	1.4	0.9	6.6
	2	1.6	0.93	7
	3	1.8	0.91	7.9
	4	2.0	0.95	9.4
4	1	1.4	0.86	2.6
	2	2.6	0.91	6
	3	3.8	0.93	7.9
	4	5.0	0.85	9.4

Table 7.2. Characteristics of available protection for single-level protection example

No of Component	Protection method γ	Protection vulnerability v	Protection cost c
1	1	0.35	0.1
	2	0.15	4.1
	3	0.05	15.7
2	1	0.01	1.0
3	1	0.60	1.0
	2	0.35	5.5
	3	0.15	17.0
4	1	0.10	1.1
	2	0.03	4.2

The structure optimization problem for desired system survivability $S^*=0.99$ was solved in Levitin and Lisnianski (2003) for $E_1=E_2=E_3=E_4=10$ in about 75 minutes (on Pentium II PC). The obtained cost was $C_{MSS}=260.8$. The improvement of this solution can be achieved by increasing the possible number of elements in each component, but this increases the length of the solution encoding string and slows the GA convergence. The further increase of the string length was impossible given the reasonable time limitation. Solving the problem using the five-processor GA on Pentium IV PC took just 238 seconds for $E_1=E_2=E_3=E_4=12$. The total cost of the obtained solution (presented in Fig. 7.2) is $C_{MSS}=260.4$.

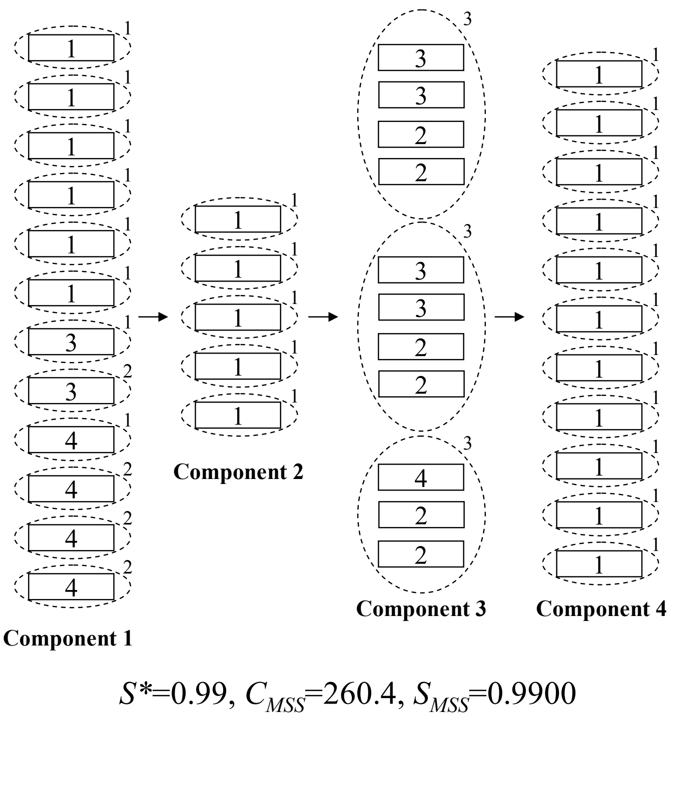


Fig. 7.2. Lowest cost MSS with single-level protection for $S^*=0.99$.

7.5.2. Minimal cost MSS with multi-level protection

Consider now the same example and assume that the elements within each component can have two level protections (different types of protection shields and casings). Assume also that the entire components can be allocated within protecting constructions and distributed among different protected sites. While the cost and vulnerability of the shields and the casings do not depend on the number of protected elements, the number of protected components strictly affects the protection cost and vulnerability. The parameters of the available protections within the components are presented in Table 7.3. The parameters of protections of the groups of entire components are presented in Table 7.4.

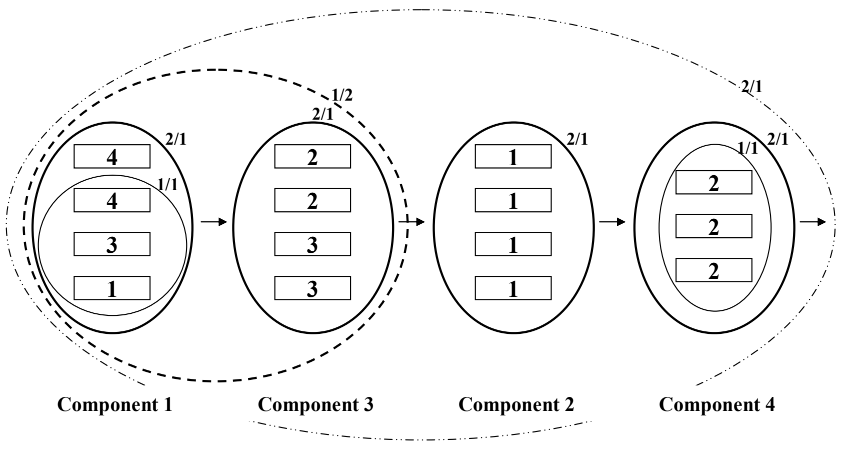
Table 7.3. Characteristics of protections available within components

No of Component	Protection level e	Protection method γ	Protection vulnerability v	Protection cost c
1	1	1	0.4	0.2
		2	0.3	2.1
		3	0.1	10.7
	2	1	0.35	0.1
		2	0.15	4.1
		3	0.05	15.7
2	1	1	0.1	1.2
	2	2	0.01	1.0
3	1	1	0.5	2.0
		2	0.37	4.5
		3	0.13	12.0
	2	1	0.60	1.0
		2	0.35	5.5
		3	0.15	17.0
4	1	1	0.2	1.5
		2	0.05	4.7
	2	1	0.10	1.1
		2	0.03	4.2

Table 7.4. Characteristics of protections available for entire components

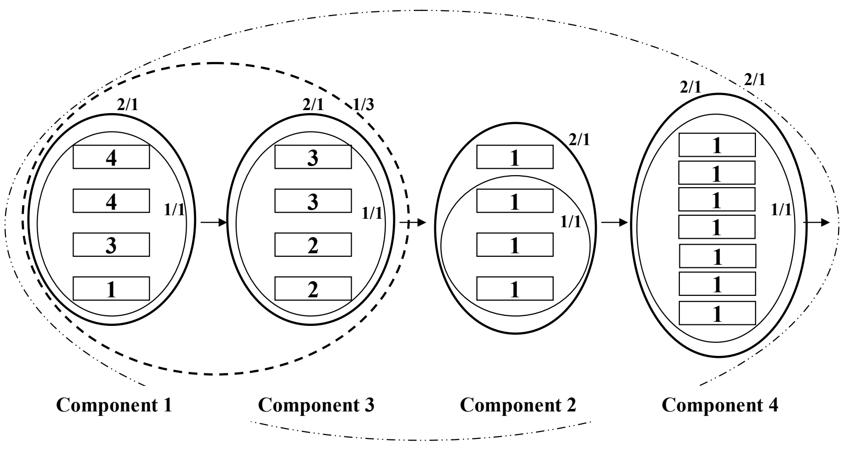
Protection level e	Protection method γ	No of protected components	Protection vulnerability v	Protection cost c
1	1	1	0.42	2.0
		2	0.44	2.2
		3	0.50	2.9
		4	0.55	3.5
	2	1	0.23	4.1
		2	0.25	4.6
		3	0.30	6.4
		4	0.35	7.3
	3	1	0.17	5.4
		2	0.19	6.0
		3	0.25	7.7
		4	0.30	8.5
2	1	1	0.38	4.0
		2	0.39	4.2
		3	0.40	4.3
		4	0.41	4.4
	2	1	0.33	8.1
		2	0.35	8.6
		3	0.38	9.4
		4	0.39	10.3

The structure optimization problem was solved for four different values of desired system survivability $S^* \in \{0.85, 0.90, 0.95, 0.99\}$. One can see the obtained solutions in Figs. 7.3-7.6, where the system elements are marked with their version numbers and each protection group is encased into an ellipse numbered in accordance with the level and the chosen method of protection (the marks has the form level/method). Ellipses corresponding to the different protection levels have different types of lines (solid lines represent the protections for elements within the components, and dashed lines represent the protections out of the entire components).



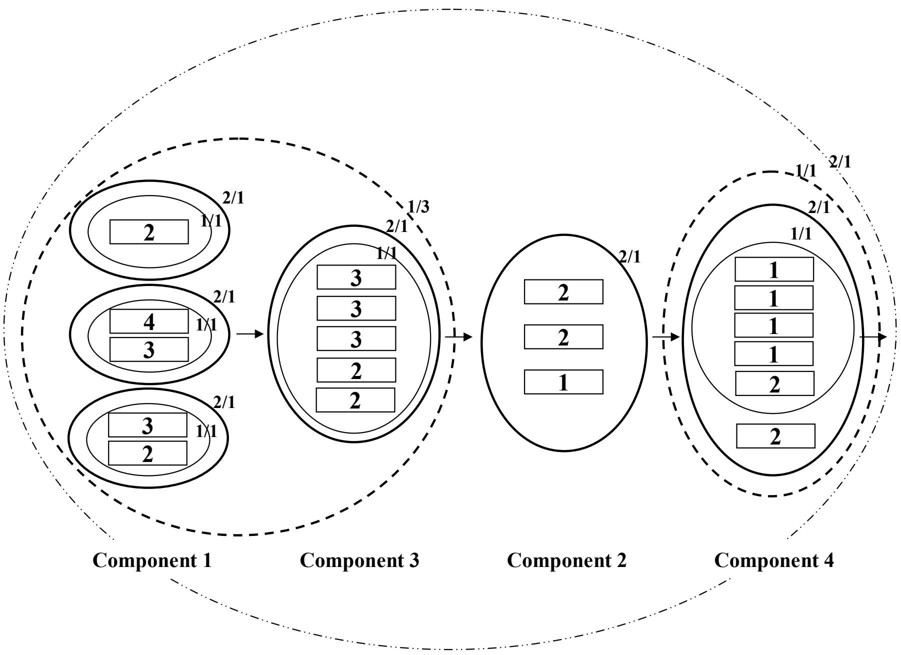
S^*=0.85, C_{MSS}=91.9, S_{MSS}=0.8519

Fig. 7.3. Lowest cost MSS with multilevel protection for *S**=0.85.



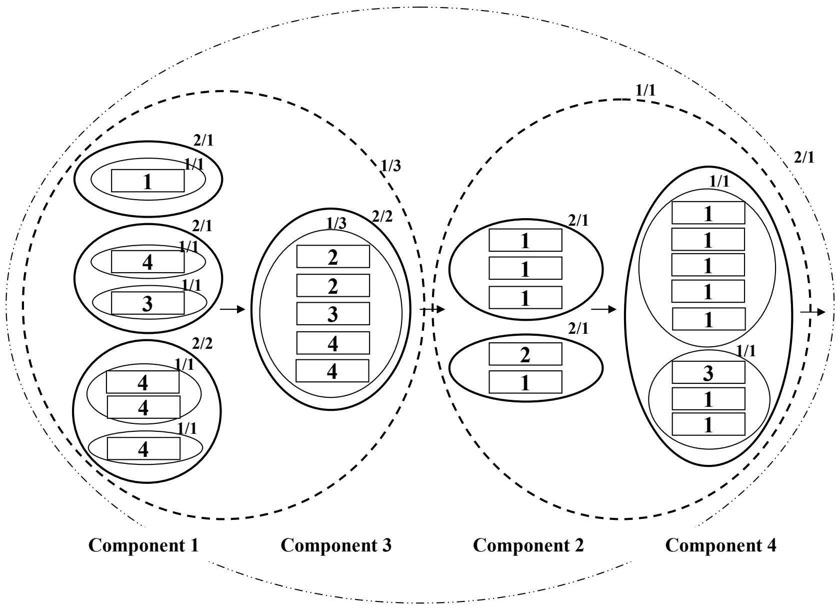
S^*=0.90, C_{MSS}=96.7, S_{MSS}=0.9006

Fig. 7.4. Lowest cost MSS with multilevel protection for *S**=0.90.



S^*=0.95, C_{MSS}=117.9, S_{MSS}=0.9501

Fig. 7.5. Lowest cost MSS with multilevel protection for *S=0.95.**



S^*=0.99, C_{MSS}=154.5, S_{MSS}=0.9901

Fig. 7.6. Lowest cost MSS with multilevel protection for *S=0.99.**

In this example, a five-processor GA was used on Pentium IV PCs with the following combination of the parameters: N_s=200, N_{rep}=2000, N_c=100, P_{mut}=1, a=500, b=0.01 and λ=10000. The running time for each one of the solved problems did not exceed 2000 seconds.

7.5.3. Computational effort and algorithm consistency

In order to show the impact of the number of processors on the running time of the multi-processor GA, we conducted an experiment on the problem presented in section 7.2 for $S^* = 0.99$. This problem was solved by Gas with diferent number of parallel processors *N*. The GA running times *T* (obtained as average of 10 GA runs for each number of processors) are presented in Fig. 7.7. This curve is compared with the function *T'*=5123/*N* corresponding to the perfect work sharing among the processors. The actual running time of *N*-processor GA takes more than *T'*. This can be explained by the fact that part of the algorithm is performed by the central server and the communication among the server and the processors take some additional time. The GA of running time curve trends to a certain constant value when the number of processors increases. This means that using more than 7-9 the processors cannot significantly improve the GA efficiency.

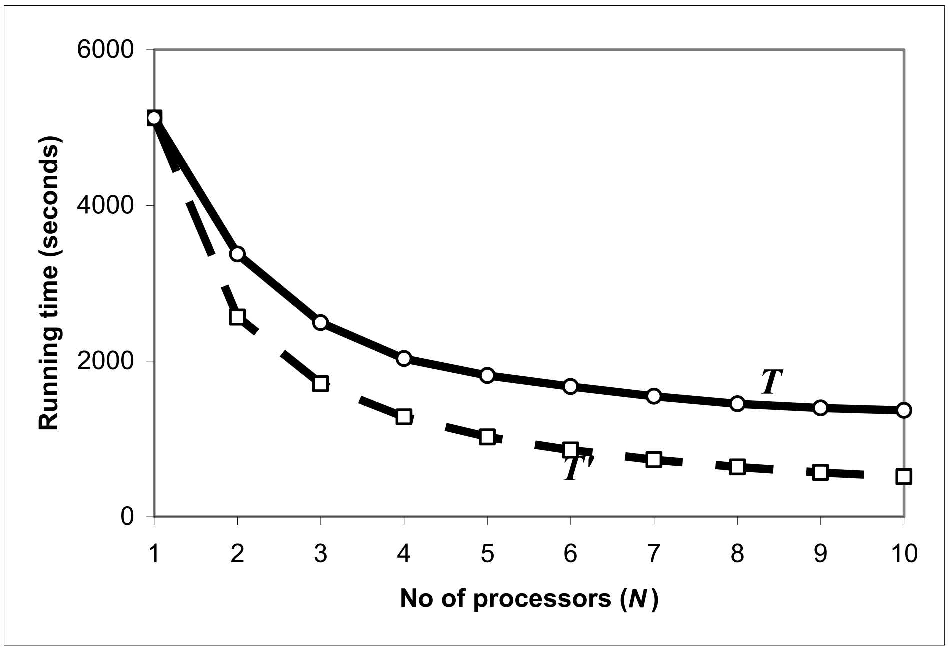


Fig. 7.7. Running time *vs.* number of processors.

To demonstrate the consistency of the suggested algorithm, GA was repeated 10 times with different starting solutions (initial population) for each one of four problems with different values of S^*. The coefficient of variation was calculated for fitness values of best-in-population solutions obtained during the genetic search by GA search processes. The variation of this index during the GA proceeding is presented in Fig. 7.8. One can see that the standard deviation of the final solution fitness does not exceed 4% of its average value.

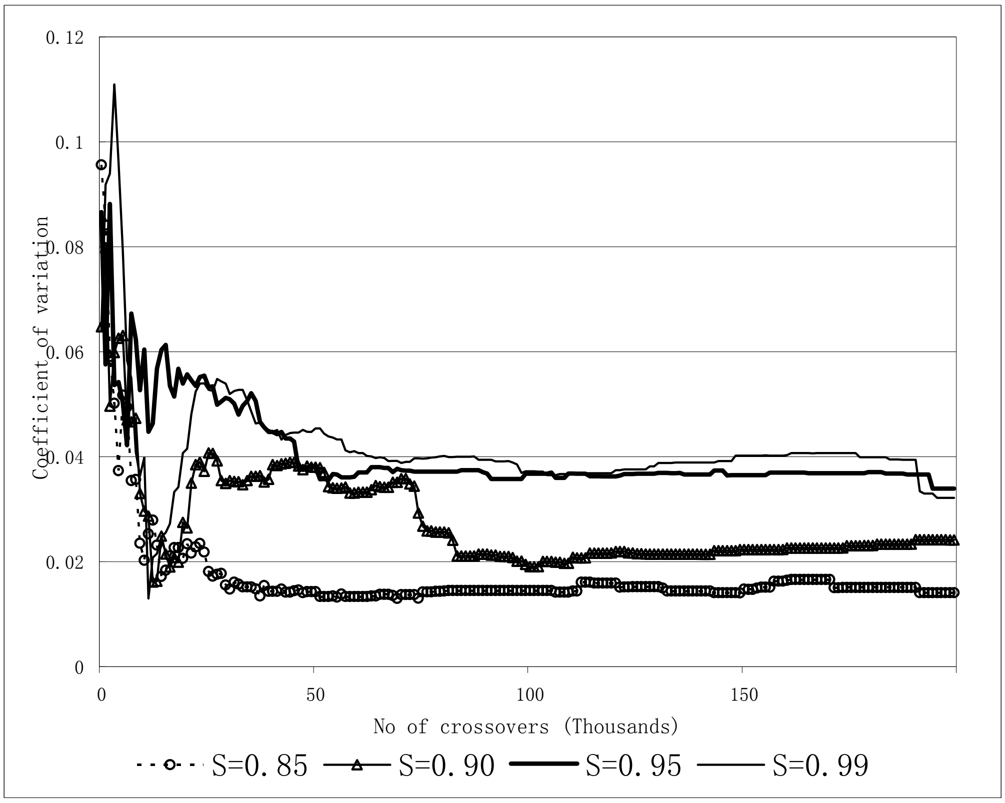


Fig. 7.8. Coefficient of variation *vs.* number of crossovers.

7.6. Conclusion

In this chapter, we formulated the problem of finding structure of series-parallel MSS with multi-level protections in order to achieve a desired level of system survivability by the minimal cost. This chapter also describes a straightforward recursive algorithm for evaluating the survivability of a series-parallel MSS with multi-level protections. Then, a multi-processor GA was developed for effectively solving the problems, where a new encoding scheme was proposed. Finally, two numerical examples were illustrated.

However, this chapter assumed that the failures of different groups of elements were independent from one another, which is not always true in practice. The correlation among different failures should be considered in future research, see e.g.

Levitin (2001).

Moreover, this chapter just studied the series-parallel structure. In fact, those elements may be connected or distributed according to other arbitrary topologies, such as bridge topology etc. Hence, the problems of other generally distributed systems are also worth further studying.

Anyway, this chapter extended the previous research (Levitin and Lisnianski, 2003) into a more general and practical case by introducing the multi-level protections. The general steps of our multi-processor GA can also be implemented in other similar problems so that the effectiveness of GA can be improved without reducing the quality of solutions.

CHAPTER 8

OPTIMAL TESTING RESOURCE ALLOCATION

8.1. Introduction

Today's software development is no longer an isolated task of a single programmer. Large systems are usually developed in a multi-language environment and run simultaneously on various platforms. Software development is a very complex process involving various factors. In Zhang *et al.* (2001), thirty-two environmental factors are defined and a survey was launched to investigate the impact of these factors on software reliability assessment. Tian (1999) describes how to establish predictive linkage between software reliability and other environmental factors which can be measured and controlled early in the development cycle, and using such predictive relations to drive continuous reliability improvement. The software-testing resource is a kind of entities, which can be measured and controlled early in the development cycle. Thus, for the development of large and complex systems, how to allocate the limited software-testing resource so that the overall reliability of the system is maximized is an important decision-making problem.

Testing-resource usually refers to the resource expenditures spent on software testing, e.g., man-power and available time, etc. During the testing stage, a project manager often faces various problems such as how to allocate available time (the time before deadline) among the modules and how to assign personnel, etc. In order to combine these two kinds of resources (man-power and available time) together, we define a term called total testing time that is calculated by multiplying the number of personnel with the available time. Each unit of the total testing time represents the resource of one person to work for one unit of time. For example, if ten persons are prepared to test the software and the deadline for testing is 1000 hours from now on (available time), the total testing time is calculated by $10 \times 1000 = 10000$ (hours). Here the testing-resource is referred to as total testing time and we use the term testing-resource as an exchangeable one with the term total testing time.

In this chapter, we study the testing resource allocation problems on both independent modules and dependent versions of software. Section 8.2 introduces the optimal testing resource allocation on independent modules with general parallel-series structures. Then, section 8.3 further extends the resource allocation problem into a more practical case by considering the failure correlation among different modules.

8.2. Testing Resource Allocation on Independent Modules

It is well known that in the development process of a computing system, the testing phase is a costly and time-consuming phase (Yamada *et al.*, 1995). During the unit-testing phase, all the testing activities of different modules are competing for the limited testing-resource. Thus, a critical problem is how to optimally allocate the total available testing-resource among software modules so that to achieve high reliability with low cost.

For the optimal testing-resource allocation problem on independent modules, the following assumptions are made, which are valid in this section:

(a) *n* modules in a software are independent during the unit-testing phase

(b) After T_i unit time of testing, the failure rate of module *i* is $\lambda_i(T_i)$.

The reliability of module *i* is

$$R_i(x \mid T_i) = \exp\{-\lambda_i(T_i)x\}, \quad x \geq 0 \tag{8.1}$$

where *x* is the operational time after testing. Note that in the above, we have used the operational reliability definition (Yang and Xie, 2000) as it is more common that after the release, there will be no reliability growth, and hence the failure rate will remain constant equal to $\lambda_i(T_i)$.

8.2.1. Optimization model of the parallel-series modular software

The Fig. 8.1 is the structure of a mixed Parallel-Series modular software system. There are *n* groups of parallel modules and *m* serial modules.

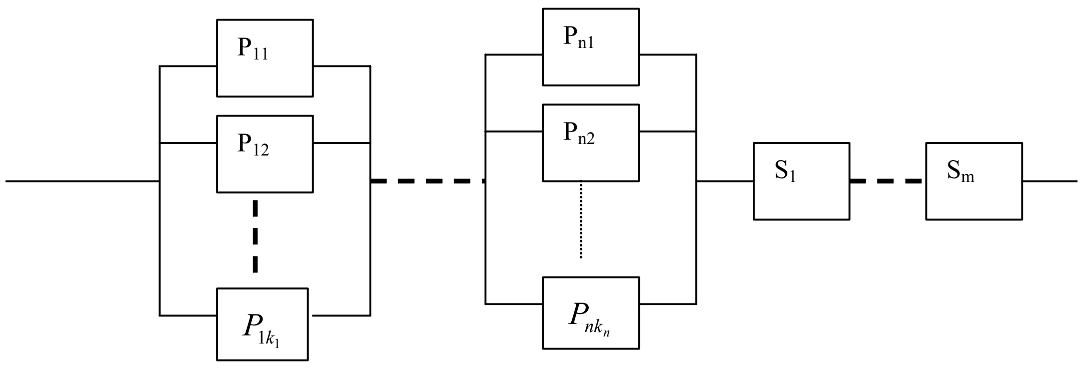


Fig. 8.1. The structure of a parallel-series modular software system.

8.2.2. Single objective of maximizing reliability

The reliability for this parallel-series modular software system is calculated as following equation

$$R(x \mid T) = \prod_{l=1}^{n} \left\{ 1 - \prod_{i=1}^{k_l} \left[1 - R_{li}(x \mid T_{li}) \right] \right\} \prod_{j=1}^{m} R_j(x \mid T_j) \tag{8.2}$$

where T_i is the testing time allocated to module *i*. Then, the following optimization model is to maximize system reliability:

$$\textbf{Maximize } R(x \mid T_{li}, T_j) = \prod_{l=1}^{n} \left\{ 1 - \prod_{i=1}^{k_l} \left[1 - R_{li}(x \mid T_{li}) \right] \right\} \prod_{j=1}^{m} R_j(x \mid T_j) \tag{8.3}$$

$$\textbf{Subject to } \sum_{l=1}^{n} \sum_{i=1}^{k_l} T_{li} + \sum_{j=1}^{m} T_j \le T \tag{8.4}$$

$$T_{li}, T_j \ge 0$$

in which *T* is the total resource of time consuming in every modules of parallel group (T_{li}) and serial modules (T_j).

8.2.3. Multiple objectives of maximizing reliability and minimizing cost

Also, note that in the development of a software system, we consider not only the system reliability but also the testing cost in the process. Hence, to minimize the cost in testing period, is another important objective to reach.

Assume that the cost function of Module *i* is $C_i(R_i)$ in which R_i is the reliability for the *i*-th module. The total cost in the parallel-series modular software system of Fig. 8.3 will be

$$C(R_{li},R_j)=\sum_{l=1}^{n}\sum_{i=1}^{k_l}C_{li}(R_{li})+\sum_{j=1}^{m}C_j(R_j) \tag{8.5}$$

where

$\sum_{i=1}^{k_l}C_{li}(R_{li})$ is the total cost of the *l*-th groups of parallel modules

$\sum_{l=1}^{n}\sum_{i=1}^{k_l}C_{li}(R_{li})$ is the total cost of all the *n* groups of parallel modules, and

$\sum_{j=1}^{m}C_j(R_j)$ is the total cost of all the series modules

Here, we adopt the cost function for individual module *i* proposed in Kumar and Malik (1991):

$$C_i(R_i)=H_i\exp(B_iR_i-D_i) \tag{8.6}$$

where H_i, B_i and D_i are constants and R_i is the individual module software reliability achieved at the end of testing. These parameters are explained in Kumar and Malik (1991). Briefly, the cost is exponentially increasing to the improved reliability of a single module.

The optimal testing-resource allocation problem can then be formulated with two objectives as

1) **Maximize** $R(x\mid T_{li},T_j)=\prod_{l=1}^{n}\left\{1-\prod_{i=1}^{k_l}\left[1-R_{li}(x\mid T_{li})\right]\right\}\prod_{j=1}^{m}R_j(x\mid T_j)$ (8.7)

2) **Minimize** $C(R_{li},R_j)=\sum_{l=1}^{n}\sum_{i=1}^{k_l}C_{li}(R_{li})+\sum_{j=1}^{m}C_j(R_j)$ (8.8)

Subject to $\sum_{l=1}^{n}\sum_{i=1}^{k_l}T_{li}+\sum_{j=1}^{m}T_j\le T$ (8.9)

$T_{li},T_j\ge 0$

in which T is the total resource of time consuming in every modules of parallel group (T_{li}) and serial modules (T_j).

For mixed parallel-series modular software, it is difficult to solve them, so the heuristic algorithms such as genetic algorithm, simulation annealing or Tabu search can be applied. Dai *et al.* (2003b) presented a genetic algorithm to solve the above multi-objective allocation problems. Here is an example of this type illustrated with that genetic algorithm.

Example 8.1. The structure of this 8 modules example is assumed as Fig. 8.2. We use the GO model for illustration. The mean value function is:

$$m_i(t) = a_i[1-\exp(-b_i t)], \quad i = 1,2,...8 \tag{8.10}$$

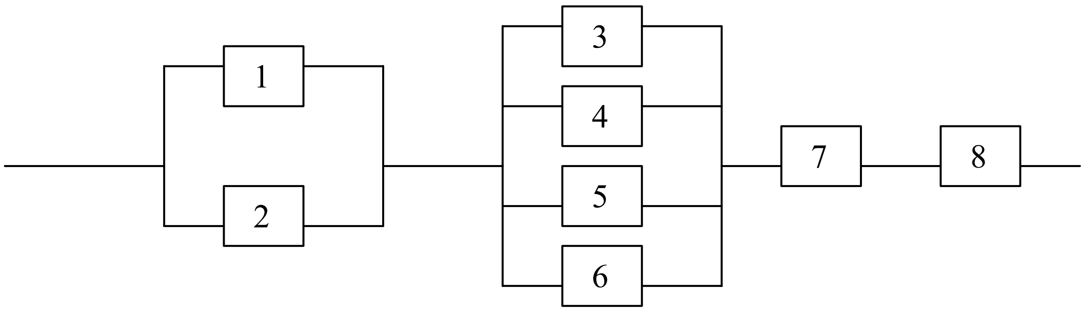


Fig. 8.2. The structure of a complex Parallel-Series modular software system.

It is assumed that the total testing time is 23000 hours and x is 200 hours to complete the given task. The values of parameters and optimal solution out of the genetic algorithm are given in the following Table 8.1 where T_i^* (i=1,2,…,8) is the optimal allocated testing time on different modules.

Table 8.1. The parameters of parallel-series modular software system.

Modules	a_i	b_i	H_i	B_i	D_i	T_i^*
1	210	0.00051	3.493	6.011	4.97	93.47
2	199	0.00059	3.503	6.12	4.93	10522
3	453	0.00048	3.498	6.012	4.995	0
4	345	0.00058	3.498	6.001	4.997	54.11
5	258	0.00063	3.499	6.002	4.995	60.48
6	221	0.00074	3.5015	6.15	4.97	8822.8
7	33.99	0.00579	3.495	6.01	4.98	2190.83
8	32.32	0.00593	3.500	6.005	4.01	1256.31

8.3. Testing Resource Allocation on Dependent N Versions

A method to increase the reliability of safety critical software is the *N*-version programming technique, e.g. Avizienis (1985). The *N*-version programming technique involves the execution of multiple, independently generated, versions of a single program. These versions receive identical input and each produces its own version of output. A voting scheme matches and tests the outputs, and then determines a final result. In the analysis of this type of systems, a common assumption is the independence of different versions, which was discussed as the independent parallel modules by the previous section.

However, failures of different versions are usually correlated, even though they could have been developed individually. It is shown in Knight and Leveson (1986) that there are factors that could affect the behavior of different versions in the same way. Different versions of software may have used some similar subroutines or functions and sometimes they may even contain the same part of codes, especially for object-oriented programming. Hence, certain external or internal reasons may cause part or all of the *N* versions to fail together due to the Common Cause Failures, see e.g. Kvam and Miller (2002).

Thus, when we allocate the testing resource/time on the *N*-version software, the correlated parts among different versions have to be considered because improving the reliability of one version by testing/debugging may also cause other correlated versions to change simultaneously. Such phenomenon often occurs when the faults that cause the common cause failures of those correlated versions are debugged, such as removing the faults in a subroutine simultaneously called by different versions.

In the following subsections, we first present a model to approach the reliability function of the dependent *N* versions of software. Then, based on the model, optimal allocation problem of the testing resource/time on the dependent *N* versions is modeled and solved.

8.3.1. Reliability analysis for dependent *N*-version programming

The *N*-version programming involves the execution of multiple versions of software, which are possibly dependent on one another by certain correlated failures. These versions receive identical input and each produces its own version of output. Some inputs may cause part or all of the *N* versions to fail together due to the so called Common Cause Failure.

A voting scheme matches and tests the outputs, and then determines a final result. There are various voting schemes. We implement here the voting scheme of "selecting the first qualified result", which is explained in details in Belli and Jedrzejowicz (1991). In this voting scheme, if any one version among the *N* versions of software passes a test, the voter will select it as the final result no matter whether the other versions are qualified or not. In other words, unless all of the *N* versions fail, the software is still reliable, which is in fact a 1-out-of-*N* voting system. However, unlike the case with parallel redundant system for which components can be assumed to be independent, the dependence among the *N* version software is an important issue here.

Decomposition by multi-component modeling

In the *N*-version software, any *j* versions may fail at the same time because of certain common cause failures. For example, if *j* versions of the *N*-version software share the same software packages or subroutines, these *j* versions will fail simultaneously when certain failure occurs in the common parts. We define a parameter for such failure, called dependence level, by the number of simultaneously failed versions caused by the failure.

We denote $M_{j,k}$ as the "components" that correspond to different common cause failures, where *j* (*j*=1,2,…,*N*) is the dependent level that correlates any *j* out of *N* versions and *k* (*k*=1,2,…, $K_{N,j}$) represents the *k*-th component among all the *j*-th dependent components ($M_{j\bullet}$), where

$$K_{N,j} = \binom{N}{j}$$

If all those failures with the *j*-th dependent level are numbered by *k* (*k*=1,2,…, $K_{N,j}$), $M_{j,k}$ can represent all the failures with different dependent levels, respectively. The total number of all "components" $M_{j,k}$ (*j*=1,2,…,*N*; *k*=1,2,…,$K_{N,j}$) is equal to

$$\sum_{j=1}^{N} K_{N,j} = \sum_{j=1}^{N} \binom{N}{j} = 2^N - 1$$

The *N* dependent versions of software can be decomposed into the mutually exclusive $2^N - 1$ components. Note that the *N* versions are not physically separated into the $2^N - 1$ components; the correlated parts objectively exist and our model merely divides them logically into exclusive components to consider their effects. An example of three-version programming is illustrated below.

Example 8.2. Three dependent versions are correlated as Fig. 8.3 and can be decomposed into 7 mutually exclusive components. $M_{1,k}$ (*k*=1,2,3) denote the failures that affect only the *k*-th version without influence on the other two versions; $M_{2,k}$ (*k*=1,2) denote the common cause failures that correlate the *k*:th and (*k*+1):st versions without influence on the other one version; $M_{2,3}$ represents the failure that correlates the first and the third versions; and $M_{3,1}$ denotes the failures that correlate all the three versions.

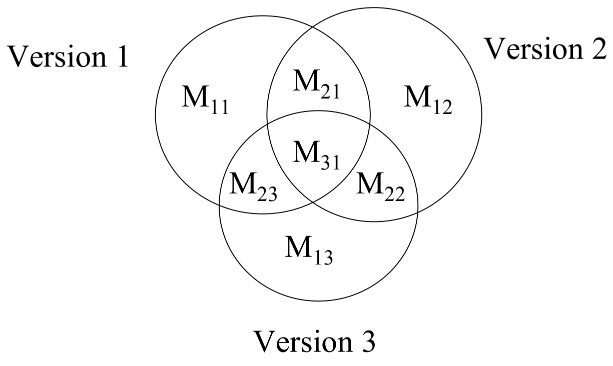


Fig. 8.3. Three dependent versions of software.

After separating the three dependent versions into 7 independent components, the reliability block diagram for those components can be built as shown in Fig. 8.4.

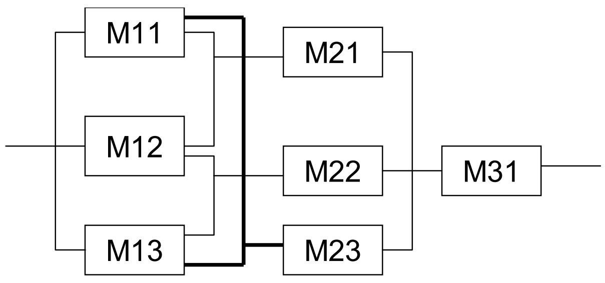


Fig. 8.4. Reliability block diagram of the decomposed components in Fig. 8.3.

The reliability block diagram is complex containing not only many parallel-series units but also some bridge structures. Moreover, the diagram will become much more complicated for four or more versions. Hence, the reliability estimation for dependent *N*-version programming is not straightforward. In order to analyze the system reliability based on our above model, a general approach is presented below.

System reliability function

The reliability of a component $M_{j,k}$ is defined as the probability for the corresponding common cause failure not to occur, which is denoted by $R_{j,k}(t)$. The software reliability of the dependent *N*-version programming is defined as the probability that at least one version of software can achieve the task successfully. The software reliability function at time *t* can be expressed as

$$R(t) = \text{Pr(at least one version of software is reliable at time } t) \qquad (8.11)$$

Let $E_i(t)$ represent the event in which the *i*-th version of software is reliable to successfully achieve the given task at time *t*, (*i*=1,2,…,*N*). The software reliability function for the dependent *N*-version programming can then be written as

$$R(t) = \Pr\{\bigcup_{i=1}^{N} E_i(t)\} \tag{8.12}$$

By using conditional probability, the events considered in the above equation can be decomposed into mutually exclusive events as

$$\begin{aligned} R(t) &= \Pr\{E_1(t)\} + \Pr\{E_2(t)\}\Pr\{\overline{E}_1(t)|E_2(t)\} + \\ &\cdots + \Pr\{E_N(t)\}\Pr\{\overline{E}_1(t), \overline{E}_2(t)\cdots \overline{E}_{N-1}(t)|E_N(t) \end{aligned} \tag{8.13}$$

where $\Pr\{\overline{E}_1(t)|E_2(t)\}$ denotes the conditional probability that the first version of the software fails given that the second version of the software is reliable at time *t*.

Hence, each term in the software reliability expression of the above equation can be evaluated in terms of the probability of two distinct events. The first event indicates that the *i*-th version of software V_i is reliable while the second event indicates that all of its previous versions V_m (*m*=1,2,…,*i*-1) fails given that V_i is reliable.

The probability of the first event, $\Pr\{E_i(t)\}$, is straightforward. It can be calculated by multiplying the reliability functions of all the components that will make the *i*-th version V_i fail as

$$\Pr\{E_i(t)\} = \prod_{V_i \in M_{j,k}} R_{j,k}(t) \tag{8.14}$$

where $V_i \in M_{j,k}$ means that the *i*-th version of software V_i will fail if the component $M_{j,k}$ fails.

The probability of the second event, $\Pr\{\overline{E}_1(t), \overline{E}_2(t)\cdots \overline{E}_{i-1}(t)|E_i(t)\}$, is not as straightforward to compute. It can be done in the following two steps:

Step 1. select out all those components that can make any version(s) among the $V_1, V_2, ..., V_{i-1}$ fail while V_i is still reliable;

Step 2. use binary search tree (Johnsonbaugh, 2001: pp. 349-354) to find out all the exclusive combinations, which can make all the *i*-1 versions $V_1, V_2, ..., V_{i-1}$ fail among those components selected by the previous step 1;

Step 3. sum up all the probabilities of those exclusive combinations to obtain the probability of $\Pr\{\overline{E}_1(t), \overline{E}_2(t) \cdots \overline{E}_{i-1}(t) | E_i(t)\}$.

After computing $\Pr\{\overline{E}_1(t), \overline{E}_2(t) \cdots \overline{E}_{i-1}(t) | E_i(t)\}$ and $\Pr\{E_i(t)\}$, *i*=1,2,…,*N*, we can obtain the software reliability function for the dependent *N*-version programming by substituting them into Eq. (8.13). For the whole approach in deriving the dependent *N*-version programming reliability, an example of aircraft landing is illustrated below.

Example 8.3. Suppose that three teams will compose three versions of a program to control the aircraft landing. If any one version is working, the aircraft can land successfully. These three versions may depend on one another through certain common cause failures. Those failures may occur on the common parts of some versions, such as using the same external electrical power, integrating the same software packages, sharing identical subroutines and so on.

As in the approaches presented above, the dependent three version software is first decomposed into its individual components. As shown in example 8.2, The dependent three versions can be decomposed into 7 components corresponding to different common cause failures as shown in Fig. 8.4. $R_{j,k}$ denotes the reliability function of $M_{j,k}$.

Then, the software reliability is derived by applying the approach presented above, and then substitute into Eq. (8.13) to get

$$\begin{aligned} R(t) = {} & R_{11}R_{21}R_{23}R_{31} + R_{12}R_{21}R_{22}R_{31}(1 - R_{23}R_{11}) \\ & + R_{13}R_{22}R_{23}R_{31}[1 - R_{21}(R_{11} + R_{12} - R_{11}R_{12})] \end{aligned} \tag{8.15}$$

8.3.2. Optimal testing resource allocation

Based on the above reliability model and approaches for the dependent *N*-version software, the optimal testing resource allocation problem on those dependent versions can be solved.

One optimization problem for testing resource allocation can be formulated to minimize the total cost for the *N* versions, when constrained by a fixed testing time budget *T* hours. Let t_i be the testing time allocated on the *i*-th version V_i (*i*=1,2,…,*N*), and the total testing time is less than *T* The allocation of testing time significantly affects the total cost. There are mainly two parts in the cost:

1) Test duration cost C_t: Here, the *N* versions of the software can be tested respectively given their allocated testing time t_i and their expected cost per unit of testing time c_i (*i*=1,2,…*N*). The test duration cost can be expressed as

$$C_t = \sum_{i=1}^{N} c_i t_i \tag{8.16}$$

where $c_i t_i$ is the expected cost in testing the *i*-th version.

2) Risk cost C_r: this is the cost incurred by an unreliable system, see e.g. Pham and Zhang (1999). This can be expressed as

$$C_r = d(1-R) \tag{8.17}$$

where d is the expected cost if the system fails and 1-*R* is the probability for the system to fail.

The total cost is the summation of the above two parts.

Denote $t_{j,k}$ the testing time for component $M_{j,k}$. During the testing period, the component $M_{j,k}$ continues running and risking the failures unless all the versions related to $M_{j,k}$ fail. Hence, the testing time of $t_{j,k}$ can be calculated by

$$t_{j,k} = \max_{m \in M_{jk}} (t_m) \tag{8.18}$$

where $m \in M_{jk}$ means version *m* is related to component $M_{j,k}$. Hence, the reliability function of the component $M_{j,k}$ can be written as $R_{j,k}(x \mid t_{j,k})$ where *x* is the operation time after the test. The whole software reliability function $R(x \mid \vec{t})$ can then be derived through our approach presented above, where $\vec{t} = \{t_i \mid i = 1,2,...N\}$. The optimization problem to minimize the total cost by finding a set of testing time allocations $\vec{t}$, can be formulated by

$$\textbf{Objective:} \quad \textbf{Minimize} \; C(\vec{t}) = C_t + C_r = \sum_{i=1}^{N} c_i t_i + d\left[1 - R(x \mid \vec{t})\right] \tag{8.19}$$

$$\textbf{Subject to:} \; \sum_{i=1}^{N} t_i \le T \tag{8.20}$$

$$t_i \ge 0 \qquad (i=1,2,\ldots,N) \tag{8.21}$$

Solving this problem is also difficult, so heuristic algorithms need be implemented. An example is illustrated where a genetic algorithm is used here to solve it.

Example 8.4. Continuing with Example 8.3 (the air-craft landing example), suppose that the testing resource budget is 2000 hours of testing time, i.e. *T*=2000, that the testing cost per hour on the three versions are $c_1 = 0.3$, $c_2 = 0.2$, $c_3 = 0.28$, and that the risk cost

$d = 10000$ if the aircraft cannot land successfully. The allocation problem becomes how to optimally allocate the 2000 hours on the three versions in order to minimize the total cost.

We assume common cause failures arriving on each component satisfy the classic NHPP model of Goel and Okumoto (1979). With this GO-model, the failure rate function for the components $M_{j,k}$ (*j*=1,2,3 and *k*=1,2,…, $K_{3,j}$) is modeled with:

$$\lambda_{j,k}(t) = a_{j,k} b_{j,k} \exp(-b_{j,k} t) \tag{8.22}$$

If the testing is stopped after t units of time, the reliability for a mission of duration *t* is given by (Yang and Xie, 2000)

$$R_{j,k}(x \mid t) = \exp\{-\lambda_{j,k}(t) \cdot x\} \tag{8.23}$$

The values of the parameters $a_{j,k}$ and $b_{j,k}$ in the GO-model are given in Table 8.2 for this example.

Table 8.2. parameters of GO-model for each component.

Component	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,1}$
$a_{j,k}$	16.91	95.52	21.56	15.80	22.45	26.23	6.25
$b_{j,k}$	0.0059	0.0006	0.0041	0.0028	0.0021	0.0022	0.0056

Then, the reliability for the dependent three-version software can be obtained through Eq. (8.15). Substitute the parameters of Table 8.2 into Eq. (8.23) to compute the reliability functions of all the components, and then substitute them into Eq. (8.15) to compute the software reliability by assuming *x*=5 (i.e. it will take 5 hours for the aircraft to land).

To solve the optimization problem as Eqs. (8.19-8.21), a genetic algorithm is used to get the solution $\vec{t} = \{638.2,\ 1361.8,\ 0\}$. Thus, the best allocation of the 2000 hours should be to test: the first version for 638.2 hours; the second for 1361.8 hours and the third for 0 hour. The total expected cost $C(\vec{t})$=579.48 and the software reliability $R(5\,|\,\vec{t})$=0.988434.

CHAPTER 9 CONCLUSIONS AND FUTURE WORK

This chapter concludes this dissertation with a summary and assessment of what this research has achieved in terms of its contributions to various related disciplines. Comments on the limitations of this work will be given and together with indications on how they may be addressed or resolved in future work.

This thesis mainly studied the reliability of various computing systems and some important issues related to the models and analysis. This work contributed much in the fields of parallel homogeneous distributed systems, centralized heterogeneous distributed systems, grid computing systems, multi-state systems, multi-type failures, failure correlations, multi-level protection, and testing resource allocation.

Chapter 3 developed a Markov model to analyze the reliability of the parallel homogeneous distributed systems, which help analyze the combined software/hardware system availability. Further, this work studied the imperfect debugging process of this type of system, where a more general Markov model was constructed. Although the model was more complex than that without considering imperfect debugging, the estimation of some important measures from this imperfect debugging model is easier and more reasonable than from the model with perfect debugging process. However, in those models, there are some unrealistic assumptions which should be relaxed in future research. The assumption of independent failures among different hosts and the uncorrelated faults between software and hardware of the same host might be the most critical assumption, because the failures may be dependent in practice, which has been revealed by the experiments of Knight and Leveson (1986). Another assumption of homogeneous property can also be relaxed to further study heterogeneous systems with various brands of hosts and different versions of software.

Thus, Chapter 4 studied the centralized heterogeneous distributed system (CHDS) in order to relax the above homogeneous assumption, where the concept of distributed service reliability was presented as the reliability from a service point of view. This measure faces directly to customers/users. A time-dependent Markov model was constructed to analyze the control center of the systems. Moreover, based on the model, many practical issues such as the determination of release time to achieve a service reliability requirement, and the sensitivity of model parameters were conducted. However, the sub-distributed systems of the CHDS inherited one assumption of the Kumar *et al.* (1986)'s model: "the operational probabilities of different nodes and links in the sub-distributed systems are of constant values." Therefore, it is worthwhile further to relax the assumption in future study.

Grid computing system is different from conventional distributed computing systems by its focus on large-scale resource sharing, where processing elements and information communication have significant influence on grid computing reliability. Chapter 5 described the property of the grid computing systems and presented algorithms to analyze the grid reliability on some important components of the grid, such as the resource management system, the network, the programs and the resources. However, this work assumed that the failures occurring on the network satisfying the homogeneous Poisson processes. This assumption may not always be true. For example, during the testing phase, the failure rate should decrease if the software or hardware faults in the system are being debugged. Other reliability analysis methods or models can be further studied in solving the problem of non-homogeneous Poisson process (NHPP) or other kinds of processes. If the grid becomes complicated with many nodes, links, programs and resources, the effectiveness of our algorithms to evaluate grid reliability should be concerned. Thus, improving the effectiveness of the algorithms for evaluating the grid reliability is also an interesting topic, which can be further studied in near future.

In reliability analysis of the computing systems, it is much possible for the successive runs to be correlated with one another and the failures may be of multiple types. In Chapter 6, a software reliability modeling framework based on Markov renewal processes was studied. It is capable of modeling the dependence among successive software runs and able to deal with multiple type of failures. With this model, the phenomena of failure clustering can be specified and analyzed. However, in this model, some parameters were difficult to precisely estimate, so the sensitivity analysis is necessary to be carried out in future work if the empirical/estimated values are used.

In Chapter 7, we formulated the problem of finding structure of series-parallel MSS with multi-level protections in order to achieve a desired level of system survivability by the minimal cost. This chapter also describes a straightforward recursive algorithm for evaluating the survivability of a series-parallel MSS with multi-level protections. Then, a multi-processor GA was developed for effectively solving the problems, in which a new encoding scheme was proposed. However, this chapter just studied the series-parallel structure, but the components of a system may be connected or distributed by other topological structure. Hence, the problems of other generally distributed systems are also worth further studying.

In software testing, an important issue is to allocate testing resources to achieve maximum reliability. Chapter 8 studied the optimization problems of testing resource allocation on both independent modules and dependent versions. At first, this chapter focused on allocating resources on the independent modules with the parallel-series structure, and multiple objectives are considered in this optimization model including maximizing the system reliability and minimizing the cost. However, this approach was restricted in the parallel-series architecture based modular system. If it could be generalized into a common model for arbitrary architectures of the modules, it should be more useful. Then, this chapter further considered the failure correlation among multiple versions of software and presented optimization models and algorithms in solving the testing resource allocation problems.

In addition, although we have tried to be theoretically rigorous, some of the theoretical topics are omitted. However, as many references as possible are provided to those interested in exploring and delving into details. Researchers and students may find many ideas useful in their academic work. Everyone who learns or uses reliability models in the computing systems should be aware of potential problems discussed in this work and also have the knowledge of at least the existence of techniques to solve these problems.

BIBLIOGRAPHY

Akhtar, S. (1994), Reliability of k-out-of-n:G systems with imperfect fault-coverage, *IEEE Transactions on Reliability*, **43**, 101-106.

Aki, S., Hirano, K. (1996), Lifetime distribution and estimation problems of consecutive-k-out-of-n:F systems, *The Annals of the Institute of Statistical Mathematics*, **48** (1), 185-199.

Avizienis, A. (1985), The N-version approach to fault tolerant software, *IEEE Transactions on Software Engineering,* **SE-11**, 1491-1501.

Becker, G., Camarinopoulos, L., Zioutas, G. (2000), A semi-Markovian model allowing for inhomogenities with respect to process time, *Reliability Engineering and System Safety,* **70** (1), 41-48.

Belli, F., Jedrzejowicz, P. (1991), An approach to the reliability optimization of software with redundancy, *IEEE Transactions on Software Engineering,* **17** (3), 310-312.

Block, H.W., Bueno, V., Savits, T.H., Shaked, M. (1987), Probability inequalities via negative dependence for random variables conditioned on order statistics, *Naval Research Logistics*, **34** (4), 547-554.

Brunelle, R.D., Kapur, K.C. (1999), Review and classification of reliability measures for multistate and continuum models, *IIE Transactions,* **31** (12), 1171-1180.

Buyya, R., Branson, K., Giddy, J., Abramson, D. (2003), The Virtual Laboratory: a toolset to enable distributed molecular modelling for drug design on the World-Wide grid, *Concurrency and Computation Practice and Experience,* **15** (1), 1-25.

Chang, C.W.J., Hsiao, M.F., Marek-Sadowska, M. (2003), A new reasoning scheme for efficient redundancy addition and removal, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **22** (7), 945-951.

Chang, H.K., Yuan, S.M. (1999), Minimal cost replicated file system with an availability constraint, *Information Sciences*, **119**, 107-23.

Chang, M.S., Chen, D.J., Lin, M.S., K.L. Ku (2000), The distributed program reliability analysis on star topologies, *Computers and Operations Research*, **27**, 129-142.

Chen, D., Chen, D., Li, L., Trivedi, K.S., Some, R.R., Nikora, A.P., Dharmaraja, S. (2002), Reliability and availability analysis for the JPL remote exploration and experimentation system, *Proceedings of the 2002 International Conference on Dependable Systems and Networks,* pp. 337-342.

Chen, D.J., Chen, R.S., Huang, T.H. (1997), A heuristic approach to generating file spanning trees for reliability analysis of distributed computing systems, *Computers and Mathematics with Application,* **34**, 115-131.

Chen, D.J., Huang, T.H. (1992), Reliability analysis of distributed systems based on a fast reliability algorithm, *IEEE Transactions on Parallel and Distributed Systems*, **3** (2), 139-154.

Chen, I.R., Bastani, F.B. (1994), Warm standby in hierarchically structured process-control programs, *IEEE Transactions on Software Engineering*, **20** (8), 658-663.

Cheung, R.C. (1980), A user-oriented software reliability model, *IEEE Transactions on Software Engineering,* **6** (2), 118–125.

Cinlar, E. (1975), *Introduction to Stochastic Processes*. Englewood Cliffs, NJ: Prentic Hall.

Clemen, R.T. (1995), *Making Hard Decisions: An Introduction to Decision Analysis,* NJ: Duxbury Press.

Dai, Y.S., Xie, M., Poh, K.L. (2002), Reliability analysis of grid computing systems, *Proceeding of Pacific Rim International Symposium on Dependable Computing,* IEEE Computer Press, pp. 97-103.

Dai, Y.S., Xie, M., Poh, K.L. (2003c), A fast algorithm for grid system reliability, *Regional Inter-University Electrical and Electronic Engineering Conference 2003*, Hong Kong.

Dai, Y.S., Xie, M., Poh, K.L., Liu, G.Q. (2003a), A study of service reliability and availability for distributed systems, *Reliability Engineering and System Safety*, **79** (1), 103-112.

Dai, Y.S., Xie, M., Poh, K.L., Yang, B. (2003b), Optimal testing-resource allocation with genetic algorithm for modular software systems, *Journal of Systems and Software,* **66** (1), 47-55.

Duane, J.T. (1964), Learning curve approach to reliability monitoring, *IEEE Transactions on Aerospace*, **AS-2**, 563-566.

Dugan, J.B., Lyu, M.R. (1994), System reliability analysis of an N-version programming application, *IEEE Transactions on Reliability,* **43** (4), 513-519.

Dugan, J.B., Lyu, M.R. (1995), System-level reliability and sensitivity analyses for three fault-tolerant system architectures, *Dependable Computing for Critical Applications*, **4**, 459-477.

Fakhre-Zakeri, I., Slud, E. (1995). Mixture models for reliability of software with imperfect debugging: identifiability of parameters, *IEEE Transactions on Reliability*, **44**, 104-113.

Findeisen, D. (2000), *System dynamics and mechanical vibrations*, Berlin; New York: Springer.

Fleming, K., Silady, F. (2002), A risk informed defense-in-depth framework for existing and advanced reactors, *Reliability Engineering and System Safety*, **78**, 205-225.

Foster, I., Kesselman, C. (1998), *The Grid: Blueprint for a New Computing Infrastructure*, San Francisco: Morgan-Kaufmann.

Foster, I., Kesselman, C., Nick, J.M., Tuecke, S. (2002), Grid services for distributed system integration, *Computer*, **35** (6), 37-46.

Foster, I., Kesselman, C., Tuecke, S. (2001), The anatomy of the Grid: Enabling scalable virtual organizations, *International Journal of High Performance Computing Applications,* **15** (3), 200-222.

Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S. (2002), Condor-G: a computation management agent for multi-institutional grids, *Cluster Computing,* **5** (3), 237-246.

Fricks, R., Puliafito, A., Telek, M., Trivedi, K.S. (1996), Markov renewal theory applied to performability evaluation, In: Bagchi, K.K.; Zobrist, G.: *Modeling and Simulation of Advanced Computer Systems: Applications and Systems*. Gordon and Breach Publishers, pp. 193-236.

Fricks, R.M., Puliafito, A., Trivedi, K.S. (1999), Performance analysis of distributed real-time databases, *Performance Evaluation*, **35**, 145-169.

Fryer, M.O. (1985), Risk assessment of computer controlled systems, *IEEE Transactions on Software Engineering*, **SE-11** (1), 125-129.

Fussell, J.B. (1975), A review of fault tree analysis with emphasis on limitations, *Proceedings of the 6th Triennial World Congress of the International Federation of Automatic Control),* **26** (4), 1-6.

Garg, S., Huang, Y., Kintala, C.M.R., Trivedi, K.S., Yajnik, S (1999), Performance and reliability evaluation of passive replication schemes in application level fault tolerance, *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, pp. 322-329.

Gaudoin, O., Lavergne, C., Soler, J.L. (1994), A generalized geometric de-eutrophication software-reliability model, *IEEE Transactions on Reliability,* **43** (4), 536-541.

Gaudoin, O., Soler, J.L. (1992), Statistical analysis of the geometric de-eutrophication software-reliability model, *IEEE Transactions on Reliability,* **41** (4), 518-524.

Gnedenko, B., Ushakov, I. (1995), *Probabilistic Reliability Engineering*, New York: John Wiley and Sons, Inc.

Goel, A.L. (1980), A summary of the discussion on “an analysis of competing software reliability models”, *IEEE Transactions on Software Engineering*, **SE-6**, 501-502.

Goel, A.L. (1985), Software reliability models: assumptions, limitations, and applicability, *IEEE Transactions on Software Engineering*, **SE-11**, 1411-1423.

Goel, A.L., Soenjoto, J. (1981), Models for hardware-software system operational performance evaluation, *IEEE Transactions on Reliability*, **30**, 232-239.

Goel, A.L., Okumoto, K. (1979), Time dependent error-detection rate model for software reliability and other performance measures, *IEEE Transactions on Reliability,* **R-28**, 206-211.

Gokhale, S.S., Marinos, P.N., Lyu, M.R., Trivedi, K.S. (1997), Effect of repair policies on software reliability, *COMPASS-Proceedings of the Annual Conference on Computer Assurance,* pp. 105-116.

Gokhale, S.S., Trivedi, K.S. (1999), A time/structure based software reliability model, *Annals of Software Engineering*, **8**, 85-121.

Goldberg, D.E. (1989), *Genetic Algorithms in Search of Optimization and Machine Learning*, NJ: Addison Wesley.

Goseva-Popstojanova, K., Trivedi, K.S. (2000), Failure correlation in software reliability model, *IEEE Transactions on Reliability*, **49**, 37-48.

Goseva-Popstojanova, K., Trivedi, K.S. (2001), Architecture-based approach to reliability assessment of software systems, *Performance Evaluation,* **45** (2-3), 179-204.

Goyal, A., Lavenberg, S.S (1987), Modelling and analysis of computer system availability, *IBM J. Research and Development*, **31**, 651-664.

Hariri, S., Mutlu, H. (1995), Hierarchical modeling of availability in distributed systems, *IEEE Transactions on Software Engineering*, **SE-21**, 50-56.

Hariri, S., Raghavendra, C.S., Kumar, V.K.P. (1985), Measures for distributed processing systems, *Proc. of Int. Directions in Comput.*, Trondheim, Norway.

Haverkort, B.R., Marie, R., Rubino, G., Trivedi, K.S. (2001), *Performability Modelling*, Chichester, England: Wiley.

Hecht, H., Hecht, M. (1986), Software reliability in the system context, *IEEE Transactions on Software Engineering*, **SE-12** (1), 51-58.

Helander, M.E., Zhao, M., Ohlsson, N. (1998), Planning models for software reliability and cost, *IEEE Transactions on Software Engineering,* **24** (6), 420-434.

Hillier, F.S., Lieberman, G.J. (1995), *Introduction to Operations Research*, New York: McGraw-Hill, Inc.

Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H., Stockinger, K. (2000), Data management in an international data grid project, In *Proc. 1st IEEE/ACM International Workshop on Grid Computing*, Springer Verlag Press.

Huang, C.Y., Lyu, M.R., Kuo, S.Y. (2003), A unified scheme of some Nonhomogenous

Poisson process models for software reliability estimation, *IEEE Transactions on Software Engineering*, **29** (3), 261-269.

Jelinski, Z., Moranda, P.B. (1972), Software reliability research. In: Freiberger W, (ed). *Statistical Computer Performance Evaluation*, New York: Academic Press, pp. 465-497.

Joe, H., Reid, N. (1985), On the software reliability models of Jelinski-Moranda and Littlewood, *IEEE Transactions on Reliability,* **R-34** (3), 216-218.

Johnsonbaugh, R. (2001), *Discrete Mathematics,* fifth edition, Upper Saddle River, New Jersey: Prentice Hapll.

Kaufman, G.M. (1996), Successive sampling and software reliability, *Journal of Statistical Planning and Inference*, **49** (3), 343-369.

Ke, W.J., Wang, S.D. (1997), Reliability evaluation for distributed computing networks with imperfect nodes, *IEEE Transactions on Reliability*, **46** (3), 342-349.

Keene, S., Lane, C. (1992), Combined hardware and software aspects of reliability, *Quality and Reliability Engineering International*, **8** (5), 419-426.

Kijima, M. (1997), *Markov processes for stochastic modeling*, London: New York: Chapman and Hall.

Kim, K.H., Welch, H.O. (1989), Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications, *IEEE Transactions on Computers*, **C-38**, 626-636.

Knight, J.C., Leveson, N.G. (1986), An experimental evaluation of the assumption of independence in multiversion programming, *IEEE Transactions on Software Engineering*, **SE-12**, 96-109.

Kremer, W. (1983), Birth-death and bug counting (software reliability), *IEEE Transactions on Reliability,* **R-32** (1), 37-47.

Kubat, P. (1989), Assessing reliability of modular software, *Operation Research Letters*, **8**, 35-41.

Kumar, A., Agrawal, D.P. (1993), A generalized algorithm for evaluating distributed-program reliability, *IEEE Transactions on Reliability*, **42**, 416-424.

Kumar, A., Agrawal, D.P. (1996), Parameters for system effectiveness evaluation of distributed systems, *IEEE Transactions on Computers*, **45** (6), 746-752.

Kumar, A., Malik, K. (1991), Voting mechanisms in distributed systems, *IEEE Transactions on Reliability,* **40** (5), 593-600.

Kumar, A., Rai, S., Agarwal, D.P. (1988), On computer communication network reliability under program execution constraints, *IEEE Journal of Selected Areas in Communications,* **6**, 1393-1400.

Kumar, V.K., Hariri, S., Raghavendra, C.S. (1986), Distributed program reliability analysis, *IEEE Transactions on Software Engineering*, **SE-12**, 42-50.

Kuo, S.Y., Huang, C.Y., Lyu, M.R. (2001), Framework for modeling software reliability, using various testing-efforts and fault-detection rates, *IEEE Transactions on Reliability*, **50**, 310-320.

Kuo, W., Prasad, V.R. (2000), An annotated overview of system-reliability optimization, *IEEE Transactions on Reliability,* **49**, 176-187.

Kuo, W., Zuo, M.J. (2003), *Optimal reliability modeling: principles and applications*, New York : John Wiley and Sons.

Kvam, P.H., Miller, J.G. (2002), Common cause failure prediction using data mapping, *Reliability Engineering and System Safety*, **76**, 273-278.

Lai, C.D., Xie, M., Poh, K.L., Dai, Y.S., Yang, P. (2002), A model for availability analysis of distributed software/hardware systems, *Information and Software Technology,* **44** (6), 343-350.

Laprie, J.C. (1984), Dependability evaluation of software systems in operation, *IEEE Transactions on Software Engineering*, **SE-10** (6), 701-714.

Laprie, J.C. (1990), On the assessment of safety-critical software systems, *12th International Conference on Software Engineering,* p. 222.

Laprie, J.C., Arlat, J., Biounes, C., Kanoun, K. (1990), Definition and analysis of hardware and software-fault-tolerant architectures, *IEEE Computer,* **23** (7), 39-51.

Laprie, J.C., Kanoun, K. (1992), X-ware reliability and availability modeling, *IEEE Transactions on Software Engineering*, **18**, 130-147.

Ledoux, J. (1999), Availability modeling of modular software, *IEEE Transactions on Reliability,* **48** (2), 159-168.

Levitin, G. (2001), Analysis and optimization of weighted voting systems consisting of voting units with limited availability, *Reliability Engineering and System Safety*, **73** (1), 91-100.

Levitin, G. (2002), Asymmetric weighted voting systems, *Reliability Engineering and System Safety*, **76** (2), 205-212.

Levitin, G. (2003), Optimal multilevel protection in series-parallel systems, to appear in *Reliability Engineering and System Safety*.

Levitin, G., Dai, Y.S., Xie M., Poh, K.L (2003), Optimizing survivability of multi-state systems with multi-level protection by multi-processor genetic algorithm, *Reliability Engineering and System Safety*, **82**, 93-104.

Levitin, G., Lisnianski, A. (2001), A new approach to solving problems of multi-state system reliability optimization, *Quality and Reliability Engineering International*, **17** (2), 93-104.

Levitin, G., Lisnianski, A. (2003), Optimizing survivability of vulnerable series-parallel multi-state systems, *Reliability Engineering and System Safety*, **79** (3), 319-331.

Levitin, G., Lisnianski, A., Beh-Haim, H., Elmakis, D. (1998), Redundancy optimization for series-parallel multi-state systems, *IEEE Transactions on Reliability*, **47**, 165-172.

Limnios, N. (1997), Dependability analysis of semi-Markov systems, *Reliability Engineering and System Safety,* **55** (3), 203-207.

Limnios, N., Oprisan G. (2000), *Semi-Markov processes and reliability*, Boston: Birkhauser.

Lin, M.S., Chang, M.S., Chen, D.J. (1999), Distributed-program reliability analysis: complexity and efficient algorithms, *IEEE Transactions on Reliability*, **R-48**, 87-95.

Lin, M.S., Chang, M.S., Chen, D.J. (1999), Efficient algorithms for reliability analysis of distributed computing systems, *Information Sciences*, **117** (1-2), 89-106.

Lin, M.S., Chang, M.S., Chen, D.J., Ku, K.L. (2001), The distributed program reliability analysis on ring-type topologies, *Computers and Operations Research*, **28**, 625-635.

Lin, M.S., Chen, D.J. (1997), The computational complexity of the reliability problem on distributed systems, *Information Processing Letters*, **64**, 143-147.

Lin, M.S., Chen, D.J., Hong, M.S. (1999), The reliability analysis of distributed computing systems with imperfect nodes, *The Computer Journal,* **42** (2), 129-141.

Lisnianski, A., Levitin, G. (2003), *Multi-state System Reliability*, Singapore: World Scientific.

Littlewood, B. (1975), A reliability model for systems with Markov structure, *Applied Statistics*, **24** (2), 172–177.

Littlewood, B. (1979), How to measure software reliability and how not to, *IEEE Transactions on Reliability,* **R-28** (2), 103-110.

Littlewood, B., Popov, P., Strigini, L. (2003), Assessing the reliability of diverse fault-tolerant software-based systems, *Safety Science*, **40** (9), 781-796.

Littlewood, B., Verrall, J.L. (1981), Likelihood function of a debugging model for computer software reliability, *IEEE Transactions on Reliability*, **R-30**, 145-148.

Liu, Y., Trivedi, K.S., Ma, Y., Han, J.J., Levendel, H. (2002), Modeling and analysis of software rejuvenation in cable modem termination systems, *Proceedings of the 13th International. Symposium on Software Reliability Engineering (ISSRE2002)*, pp. 159-170.

Livny, M., Raman, R. (1998), High-throughput resource management, *The Grid:*

Blueprint for a New Computing Infrastructure, San Francisco: Morgan-Kaufmann, pp. 311-338.

Lutfiyya, H.L., Bauer, M.A., Marshall, A.D., Stokes, D.K. (2000), Fault management in distributed systems: a policy-driven approach, *Journal of Network and Systems Management*, **8** (4), 499-525.

Lyu, M.R. (1996), *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, McGraw-Hill.

Lyu, M.R., Mendiratta, V.B. (1999), Software fault tolerance in a clustered architecture: techniques and reliability modeling, *Proc. of the 1999 IEEE Aerospace Conference*, **5**, 141 –150.

Malakhoff, A., Klinkhamer, D., McKesson, C. (1998), Analysis of the impact of reliability, availability and maintainability on ship survivability*, 6th International Conference on High Speed Marine Craft*, Norwegian Society of Chartered Engineers, (available at http://www.hurricane.net/~chrism/availability.htm)

Malhotra, M., Trivedi, K.S. (1994), Power-hierarchy of dependability-model types, *IEEE Transactions on Reliability*, **43** (3), 493-502.

Mendiratta, V.B. (1998), Reliability analysis of clustered computing systems, *Proc. of the Ninth International Symposium on Software Reliability Engineering*, pp. 268-272.

Miller, D.R. (1986), Exponential order statistic models of software reliability growth, *IEEE Transactions on Software Engineering*, **SE-12** (1), 12-24.

Moranda, P.B. (1979), Event-altered rate models for general reliability analysis, *IEEE Transactions on Reliability*, **R-28** (5), 376-381.

Musa, J.D. (1998), *Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, New York: McGraw-Hill.

Parfenov, U. (1989), *Reliability, Survivability and Effectiveness of Ship Electric Power Systems*, Russian naval academy, Leningrad, (in Russian).

Pham, H. (1992), Optimal design of k-out-of-n redundant systems, *Microelectronics and Reliability*, **32** (1-2), 119-126.

Pham, H. (1997), Reliability analysis of digital communication systems with imperfect voters, *Mathematical and Computer Modelling*, **26**, 103-112.

Pham, H. (2000), *Software Reliability*, Singapore: Springer-Verlag.

Pham, H. (2003), Software reliability and cost models: Perspectives, comparison, and practice, *European Journal of Operational Research*, **149** (3), 475-489.

Pham, H., Nordmann, L., Zhang, X.M. (1999), General imperfect-software-debugging model with S-shaped fault-detection rate, *IEEE Transactions on Reliability*, **48**, 169-175.

Pham, H., Suprasad, A., Misra, R.B. (1997), Availability and mean life time prediction of multistage degraded system with partial repairs, *Reliability Engineering and System Safety,* **56** (2), 169-173.

Pham, H., Zhang, X.M. (1999), A software cost model with warranty and risk cost, *IEEE Transactions on Computers,* **48** (1), 71-75.

Pourret, O., Collet, J., Bon, J.L. (1999), Evaluation of the unavailability of a multistate-component system using a binary model, *Reliability Engineering and System Safety,* **64** (1), 13-17.

Pukite, P., Pukite, J. (1998), *Modeling for reliability analysis : Markov modeling for reliability, maintainability, safety, and supportability analysis of complex systems*, New York : IEEE Press.

Rai, S., Veeraraghavan, M., Trivedi, K.S. (1995), A survey of efficient reliability computation using disjoint products approach, *Networks*, **25** (3), 147-163.

Rajgopal, J., Mazumdar, M. (2002), Modular operational test plans for inferences on software reliability based on a Markov model, *IEEE Transactions on Software Engineering,* **28** (4), 358-363.

Ross, S.M. (2000), *Introduction to Probability Models*, A Harcourt Science and Technology Company.

Sahner, R., Trivedi, K.S., Puliafito, A. (1995), *Performance and reliability analysis of computer systems: an example-based approach using the SHARPE software package*, Boston: Kluwer Academic Publishers.

Schick, G.J., Wolverton, R.W. (1978), An analysis of competing software reliability models, *IEEE Transactions on Software Engineering*, **SE-4**, 104-120.

Schneidewind, N.F. (1975), Analysis of error processes in computer software, *Sigplan Notices*, **10**, 337-346.

Shanthikumar, J.G. (1981), A general software reliability model for performance prediction, *Microelectronics and Reliability*, **23**, 903-943.

Siegrist, K. (1988), Reliability of systems with Markov transfer of control, II, *IEEE Transactions on Software Engineering,* **14** (10), 1478-1480.

Smith, R.M., Trivedi, K.S., Ramesh, A.V. (1988), Performability analysis: measures, an algorithm, and a case study, *IEEE Transactions on Computers,* **37** (4), 406-417.

Sols, A., Nachlas, J.A. (1995), Availability of multifunctional systems, *Reliability Engineering and System Safety,* **47** (2), 69-74.

Sridharan, V., Jayashree, P.R. (1998), Transient solutions of a software model with imperfect debugging and generation of errors by two servers, *Mathematical and Computer Modelling*, **27**, 103-108.

Srinivasan, S., Jha, N.K. (1999), Safety and reliability driven task allocation in distributed systems, *IEEE Transactions on Parallel and Distributed Systems*, **10** (3), 238-251.

Sumita, U., Masuda, Y. (1986), Analysis of software availability/reliability under the influence of hardware failures, *IEEE Transactions on Software Engineering*, **SE-12** (1), 32-41.

Tian, J. (1999), Measurement and continuous improvement of software reliability throughout software life-cycle, *Journal of Systems and Software,* **47** (2-3), 189-195.

Tokuno, K., Yamada, S. (2000), An imperfect debugging model with two types of hazard rates for software reliability measurement and assessment, *Mathematical and Computer Modelling,* **31** (10-12), 343-352

Tokuno, K., Yamada, S. (2000), Markovian software availability measurement based on the number of restoration actions, *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, **E83-A** (5), 835-841.

Tokuno, K., Yamada, S. (2001), Markovian modeling for software availability Analysis under intermittent use, *International Journal of Reliability, Quality and Safety Engineering,* **8** (3), 249-258

Trivedi, K.S. (1982), *Probability and Statistics with Reliability, Queueing, and Computer Applications*, Englewood, NJ: Prentice-Hall.

Trivedi, K.S. (2001), *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, New York: John Wiley and Sons.

Trivedi, K.S. (2002), SREPT: A tool for software reliability estimation and prediction, *Proc. of the International Conference on Dependable Systems and Networks*, p. 546.

Ushakov, I. (1986), Universal generating function, *Sov. J. Computing System Science*, **24**, 118-129.

Veeraraghavan, M., Trivedi, K.S. (1994), A combinatorial algorithm for performance and reliability analysis using multistate models, *IEEE Transactions on Computers,* **43** (2), 229-234.

Welke, S.R., Johnson, B.W., Aylor, J.H. (1995), Reliability modeling of hardware/software systems, *IEEE Transactions on Reliability*, **44** (3), 413-418.

Whitley, D. (1989), The GENITOR algorithm and selective pressure: why rank-based allocation of reproductive trials is best, *Proc. 3th International Conf. on Genetic Algorithms.*, pp. 116-121.

Wu, S.M., Chan, L.Y. (2003), Performance utility-analysis of multi-state systems, *IEEE Transactions on Reliability*, **52** (1), 14-21.

Xie, M. (1987), A shock model for software failures, *Microelectronics and Reliability*, **27**, 717-724.

Xie, M. (1991), *Software Reliability Modelling*. Singapore: World Scientific.

Xie, M. (2000), Software reliability models - past, present and future, In *Recent Advances in Reliability Theory: Methodology, Practice, and Inference*, Eds. N. Limnios, M. Nikulin, Boston: Birkhäuser, pp. 325-340.

Xie, M., Hong, G.Y., Wohlin, C. (1997), A study of the exponential smoothing technique in software reliability growth prediction, *Quality and Reliability Engineering International,* **13** (6), 347-353.

Xie, M., Zhao, M. (1993), On some reliability growth models with graphical interpretations, *Microelectronics and Reliability*, **33 (**2), 149-167.

Xue, J., Yang, K. (1995), Dynamic reliability analysis of coherent multistate systems, *IEEE Transactions on Reliability*, **44** (4), 683-688.

Yamada, S., Ohba, M., Osaki, S. (1984), S-shaped software reliability growth models and their applications, *IEEE Transactions on Reliability,* **R-33** (4), 289-292.

Yamada, S., Ohtera, H. (1990), Software reliability growth models for testing-effort control, *European Journal Operations Research*, **46**, 343-349.

Yamada, S., Osaki, S. (1985), Cost-reliability optimal release policies for software systems, *IEEE Transactions on Reliability*, **R-34**, 422-424.

Yamada, S., Osaki, S. (1985b), Software reliability growth modeling: models and applications, *IEEE Transactions on Software Engineering*, **SE-11**, 1431-1437.

Yamada, S.T., Nishiwaki, I.M., (1995), Optimal allocation policies for testing-resource based on a software reliability growth model, *Mathematical and Computer Modelling*, **22** (10-12), 295-301.

Yang, B. Xie, M. (2000), A study of operational and testing reliability in software reliability analysis, *Reliability Engineering and System Safety*, **70**, 323-329.

Yang, B., Xie, M. (2001), Optimal testing-time allocation for modular systems, *International Journal of Quality and Reliability Management*, **18 (**8), 854-863

Zang, X., Sun, N., Trivedi, K.S. (1999), A BDD-based algorithm for reliability analysis of phased-mission systems, *IEEE Transactions on Reliability*, **48** (1), 50-60.

Zang, X., Wang, D., Sun, H., Trivedi, K.S. (2003), A BDD-based algorithm for analysis of multistate systems with multistate components, *IEEE Transactions on Computers*, **52** (12), 1608-1618.

Zhang, T.L., Horigome, M. (2001), Availability and reliability of system with dependent components and time-varying failure and repair rates, *IEEE Transactions on Reliability*, **50**, 151-158.

Zhang, X.M., Shin, M.Y., Pham, H. (2001), Exploratory analysis of environmental factors for enhancing the software reliability assessment, *Journal of Systems and Software*, **57** (1), 73-78.