THE OPTIMUM SOFTWARE RELEASE PROBLEM

by

M. Ebru Angün

BS. In I.E., Istanbul Technical University, 1995

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Industrial Engineering

Boğaziçi University

1998

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my thesis supervisor, Prof.Dr. Süleyman Özekici, for his guidance, motivation and encouragement throughout the thesis.

I want to express my special thanks to Assoc. Prof.Dr. Kuban Altınel and Prof. Oğuz Tosun for being kind enough to take part in my thesis committee and for their precious suggestions.

I am also grateful to my family for their support and patience throughout the study.

ABSTRACT

The determination of the optimum software release time is an important problem in software engineering. There is an apparent trade-off between the reliability of the software and its cost. Thus, in many commercial software applications, the optimum release time is determined by minimizing its cost function.

The aim of this study is to build a new model to determine the optimum testing strategy when the software has an operational profile. The operational profile shows how users employ the system in the operational phase, and it consists of a set of distinct operations with their associated occurrence probabilities. The usage of the operational profile to guide testing makes this phase faster and more efficient.

Furthermore, an extensive literature review for the reliability models and optimum software release policies are presented as well as the detailed description of the operational profile and other related concepts.

ÖZET

Yazılımın piyasaya sürülme zamanının belirlenmesi yazılım mühendisliğinde önemli bir problem teşkil etmektedir. Eğer yazılım güvenilirliğini eniyileyecek şekilde denemeye tabi tutulursa deneme maliyeti artacak, aksi takdirdeyse bakım maliyetleri yükselecektir. Eniyi sürüm zamanının hesaplanabilmesi için toplam maliyet fonksiyonunu minimize eden modeller geliştirilmiştir.

Bu çalışmanın amacı, operasyonel profile sahip bir yazılımın eniyi deneme sürelerinin maliyet fonksiyonu minimize edilerek belirlenmesidir. Operasyonel profil, kullanıcının bir sistemi nasıl kullanacağını belirler ve operasyon seti ve bu operasyonların gerçekleşme olasılıklarından oluşmuştur. Operasyonel profilin kullanılması deneme süresinin daha verimli kullanılmasını sağlamaktadır.

Ayrıca bu çalışmada güvenilirlik modelleri ve eniyi sürüm zamanı metotları hakkında detaylı bir kaynak taraması verilmektedir. Operasyonel profil ve diğer ilgili terimler de detaylı olarak incelenmiştir.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

ABSTRACT iv

ÖZET v

LIST OF TABLES viii

LIST OF SYMBOLS ix

1. INTRODUCTION 1
2. SOFTWARE RELIABILITY AND OPERATIONAL PROFILE 2
   2.1. Importance of Measuring Software Reliabilty 2
   2.2. Basic Concepts 3
   2.3. Operational Profile 6
      2.3.1. Benefits and Costs 8
      2.3.2. Development 9
      2.3.3. Examples of Operational Profile 10
3. REVIEW OF THE SOFTWARE RELIABILITY MODELS 15
   3.1. Markovian Models 15
      3.1.1. General Poisson-Type Models 15
      3.1.2. Binomial-Type Models 17
      3.1.3. Poisson-Type Models (Finite Failures) 19
   3.2. Times Between Failures Models 20
      3.2.1. The Model of Jelinski and Moranda 21
      3.2.2. Littlewood and Verrall Model 21
      3.2.3. Goel and Okumoto's Imperfect Debugging Model 21
      3.2.4. Schick and Wolverton Model 22
   3.3. Fault Count Models 22
      3.3.1. Goel and Okumoto's Nonhomogeneous Poisson Process Model 22

3.3.2. Goel's Generalized Nonhomogeneous Poisson Process Model 23
3.3.3. Musa's Execution Time Model 23
3.3.4. Musa- Okumoto Logarithmic Poisson Execution Time Model 23
3.4. Fault Seeding Models 24
4. OPTIMUM SOFWARE RELEASE POLICIES 25
4.1. Static Policies 25
4.2. Dynamic Policy 30
5. THE OPTIMUM SOFTWARE RELEASE PROBLEM 32
5.1. Notation 32
5.2. Assumptions 32
5.3. The General Model 34
5.4. Determination of the Optimum Testing Policy for the Operational Profile 42
5.5. A Heuristic Algorithm 51
5.6. The Exponential Model 54
5.7. Interpretations 56
6. ILLUSTRATIONS 58
7. CONCLUSION 68
REFERENCES 69

LIST OF TABLES

		Page
TABLE 2.3.1.	Operational Profile.	7
TABLE 2.3.3.1.1.	Part of the Operational Profile for System Mode.	12
TABLE 2.3.3.1.2.	Expanded Operational Profile.	13
TABLE 2.3.4.2.1.	Operational Profile For Data Driven System.	14
TABLE 6.1.	Parameters for the First Illustration.	61
TABLE 6.2.	Iteration 1 of the First Illustration.	61
TABLE 6.3.	Iteration 2 of the First Illustration.	61
TABLE 6.4.	Iteration 3 of the First Illustration.	61
TABLE 6.5.	Iteration 4 of the First Illustration.	62
TABLE 6.6.	Iteration 5 of the First Illustration.	62
TABLE 6.7.	Parameters for the Second Illustration.	63
TABLE 6.8.	Iteration 1 of the Second Illustration.	63
TABLE 6.9.	Iteration 2 of the Second Illustration.	63
TABLE 6.10.	Iteration 3 of the Second Illustration.	64
TABLE 6.11.	Iteration 4 of the Second Illustration.	64
TABLE 6.12.	Iteration 5 of the Second Illustration.	64
TABLE 6.13.	Iteration 6 of the Second Illustration.	64
TABLE 6.14.	Iteration 7 of the Second Illustration.	65
TABLE 6.15.	Iteration 8 of the Second Illustration.	65
TABLE 6.16.	Iteration 9 of the Second Illustration.	65
TABLE 6.17.	Iteration 10 of the Second Illustration.	65
TABLE 6.18.	Iteration 11 of the Second Illustration.	66
TABLE 6.19.	Iteration 12 of the Second Illustration.	66
TABLE 6.20.	Iteration 13 of the Second Illustration.	66
TABLE 6.21.	Iteration 14 of the Second Illustration.	66
TABLE 6.22.	Iteration 15 of the Second Illustration.	67
TABLE 6.23.	Iteration 16 of the Second Illustration.	67

LIST OF SYMBOLS

a_i	Cost of finding a fault in operation i
b_i	Benefit of making all the testing in operation i
c_i	Testing cost per unit time in operation i
d_i	Debugging cost in operation i
e_i	Debugging cost after release in operation i
f_i	Benefit of removing a fault during testing in operation i rather than removing after release
$F_i(.)$	Time-to-failure distribution in operation i
H^L	Hessian matrix of L
K	Total number of operations in the operational profile
$L(x)$	The representation of the objective function of the model
p_i	The probability of removing a fault in operation i
R	A closed, bounded region in E
R^1	One-dimensional region
R^n	n-dimensional region
T_i	The time at which a fault is removed in operation i
t_i	Duration of testing for operation i
x_i	Transformed variable
z_i	Hazard rate function of operation i
λ	Exponential distribution failure rate
μ	The expected number of faults in the software before testing
π_i	The probability that the software will perform operation i after release

1. INTRODUCTION

An important problem in software engineering is the determination of the optimal release time. In life-critical software applications, the most important attribute of the software system is its reliability. However, in most other cases the objective is to minimize an expected total cost function which includes the costs incurred during testing as well as the costs incurred during operational phase. The testing phase is time consuming and costly. On the other hand, if the software is released too early without sufficient testing this will result in high maintenance cost because of the failures that will be caused by the remaining faults. This fact constitutes a tradeoff between the costs incurred during testing and the costs incurred during operational phase.

There are models in literature that are developed to determine the optimal release time of the software as if the software is working under the same conditions all time. An undesirable restriction in almost all of the testing procedure is that the parameters of the software failure process as well as the costs parameters of the software are assumed to be independent of the operation that the software performs. This assumes that the software is used for a single operation or that there are no differences between the model parameters under different working conditions. In [1] and [2], Musa indicates that the operational profile reduces the system risk by making it more realistic and it also makes the testing procedure faster and more efficient.

The aim of this study is to present a software reliability model with imperfect debugging when the time-to-failure has an arbitrary distribution. The decision to release the software or not is based on the minimization of a nonlinear cost function.

In Section 2, we will review some basic concepts of the software reliability literature and also the important concept on "Operational Profile." In Section 3, software reliability models will be presented and in Section 4, the optimum software release policies will be reviewed. Our model will be presented in Section 5 and some numerical illustrations will be given in Section 6.

2. SOFTWARE RELIABILITY AND OPERATIONAL PROFILE

2.1. Importance of Measuring Software Reliability

In [3], Musa et al. define three of the most important needs of software customers in regard to software products as quality, delivery time and cost. The development and operational costs of software have increased substantially. At the same time, time period available for the introduction of a new software product before being surpassed in capability or cost by another, has shortened. With these cost and delivery time pressures, it is becoming increasingly impossible to create a software product which provides high quality, rapid delivery and low cost simultaneously. Consequently, the need for trade-off is pressing.

Quantitative measures exist for cost and delivery time of software products, but the quantification of the quality has been more difficult. However, the absence of such a concrete measure for software quality means that quality will suffer when it competes for attention against cost and delivery time.

According to [4], the elements of software quality are:

1. Correctness: extent to which a program satisfies its specifications and fulfills the user's objectives;
2. Reliability: extent to which a program can be expected to perform its intended function with required precision;
3. Efficiency: the amount of computing resources and code required by the program to perform a function;
4. Integrity: extent to which access to software or data can be controlled;
5. Usability: effort required to learn, operate, prepare the input and interpret the output of the program;
6. Maintainability: effort required to locate and fix an error in an operational program;
7. Testability: effort required to test a program to ensure that it performs its intended function;

8. Flexibility: effort required to modify an operational program;
9. Portability: effort required to transfer a program from one hardware configuration and software system environment to another;
10. Interoperability: effort required to couple one system with another;
11. Reusability: extent to which a program can be used in other applications related to the packaging and the scopes of the functions that the program performs.

Among all these elements of software quality, software reliability has proved to be the most readily quantifiable so that it can be used as a quantitative measure for software quality.

2.2. Basic Concepts

The generally accepted definition of software reliability is that it is the probability of failure-free operation of a computer program in a specified environment for a specified time [3].

We will now examine several terms in this definition to gain greater insight. A computer program is defined as a set of complete machine instructions that executes within a single computer and accomplishes a specific function. For the purpose of software reliability modeling, the program is assumed to be stable (not changing in size or content with time). We should note that distributed systems and networks are considered to have separate programs executing in each of their computers.

The execution of a program can be viewed as a single entity, lasting for months or even years for real-time systems. However, it will be easier to characterize the operational profile if the execution is divided into a set of runs. Runs that are identical repetitions of each other form a run type. A run type is ordinarily associated with the accomplishment of a user function. A run type is specified by its input state or set of values for its input variables that it receives. We judge the reliability of a program by the output states (sets of values of output variables created) of its runs. Thus, the run type represents a transformation between an input state and an output state. Multiple input states may map to

the same output state, but a given input state can have only one output state. In other words, the input state uniquely determines the instructions that will be executed and the values of their operands. Whether a particular fault will cause a failure for a specific run type is predictable in theory. However, the analysis required to determine this might be impractical to pursue [3].

A failure represents a difference between the actual value of an output variable, resulting from a run, and the value prescribed by the requirements. The program has to be executing for a failure to occur. Thus, the term failure relates to the behavior of the program. A failure type is characterized by both input state or run type and discrepancy. Hence, two failures have the same type if they occur for the same run type and are characterized by the same discrepancy. If effective repair action is not taken, repeated failures of the same type will occur when the same run type recurs.

A fault can be defined as a defective, missing or extra instruction or set of related instructions that is the cause of one or more actual or potential failure types. By definition, there cannot be multiple faults causing a failure so that the entire set of defective instructions that is causing the failure is considered to be the fault. The set of input states that produces failures for a fault at any point in time is called its fail set. A fault is considered to be removed when the fail set is null. A fault is a property of the program rather than a property of its execution or behavior. Since in some of the reliability models the time at which a fault is introduced into the program is important, it is necessary to make the distinction between two types of faults:

1. Inherent faults: the faults that are associated with a software product as originally written or modified;
2. Seeded faults: the faults that are introduced to the program through fault correction or during testing.

Reliability quantities have usually been defined with respect to time. We are concerned with three kinds of time:

1. Execution time: the accumulated processor (CPU) time;
2. Calendar time: the actual chronological period;
3. Clock time: the execution time spent plus wait time and the execution time of other programs.

As indicated in [3], execution time is important because it is generally accepted that models based on execution time are superior. However, quantities should ultimately be related back to calendar time to be meaningful to engineers and managers.

A software reliability model describes software failures as a random process. There are two equivalent ways of describing the failure random process: the times of failures or the number of failures in a given period. Failure behavior is affected by two principal factors:

1. The number of faults in the software being executed;
2. The operational profile of execution.

Both the human error process which causes the introduction of faults into the program and the run selection process are dependent on an enormous number of variables so that the use of a random process model is appropriate. One of the important measure in the process is the mean value function, which represents the expected number of failures at time t. It is defined as

$$\mu(t) = E[M(t)] \tag{2.1}$$

where $M(t)$ is the number of failures experienced by time t. The function $\mu(t)$ is nondecreasing and is assumed to be a continuous and differentiable function of time t. Another important measure of the process is the failure intensity function. The failure intensity function is the instantaneous rate of change of the expected number of failures with respect to time and is defined by

$$\lambda(t) = \frac{d\mu(t)}{dt}. \tag{2.2}$$

The following relationships generally apply to reliability [3]. Reliability, denoted $R(t)$, is related to failure probability $F(t)$ by

$$R(t) = 1 - F(t). \tag{2.3}$$

If $F(t)$ is differentiable, then the failure density $f(t)$ is the first derivative of $F(t)$ with respect to time t. The hazard rate $z(t)$ is the conditional failure density, given that no failure has occurred in the interval between 0 and t. It is defined as

$$z(t) = \frac{f(t)}{R(t)}. \tag{2.4}$$

Then, it can be also related to the reliability by

$$R(t) = \exp\left[- \int_0^t z(x)dx \right]. \tag{2.5}$$

2.3. Operational Profile

A software-based product's reliability depends on just how a customer will use it. Making a good reliability estimate depends on testing the product as if it were in the field. The operational profile is thus essential in software reliability engineering. In addition, the operational profile shows you how to increase productivity and reliability and speed development by allocating development resources to functions on the basis of use [1].

Using an operational profile to guide testing ensures that if testing is terminated and the software is released because of imperative schedule constraints, the most-used operations will have received the most testing and the reliability level will be the maximum that is practically achievable for the given test time.

An operation is an externally-initiated task performed by a system "as built." It is different from a function, which is an externally-initiated task to be performed by a system, as viewed by users. When the idea or the need for the task arises in the minds of users or developers, it is a function. As the system is designed, functions evolve into and are implemented as operations.

The operational profile is now simply a set of operations and their probabilities of occurrence. Thus, it is a quantitative and probabilistic characterization of how a system will be used. For example, suppose a system that receives various alarms and processes them, taking actions that depend on the particular alarms. Table 2.3.1 shows a possible operational profile for such a system.

Table 2.3.1. Operational Profile

Operation	Occurrence Probability
Alarm 1 processing	0.20
Alarm 2 processing	0.15
Alarm 3 processing	0.10
Alarm 4 processing	0.08
.	.
.	.
Alarm M processing	0.01
Total	1

Let $X:\{X_n; n \geq 0\}$ denote the operational process with the state space E where X_n is the *nth* operation performed by the system. If the stochastic process X is ergodic with the limiting distribution:

$$\pi(i) = \lim_{n\to\infty} P[X_n = i] \tag{2.6}$$

then $\pi(i)$ is simply the proportion of i operations performed in the long run. Thus, the pair (E, π) is the operational profile that consists of the set of all possible operations and their occurrence probabilities.

We don't have to worry about the ergodicity of the model if X is a sequence of independent and identically distributed random variables with some distribution π since it is obvious then that the operational process is ergodic with limiting probabilities given by π.

2.3.1. Benefits and Costs

The benefit-to-cost ratio in developing and applying the operational profile is typically 10 per cent or greater.

The benefits of operational profiles can be enumerated as in [2]:

1. Increase user satisfaction by capturing their needs more precisely,
2. Satisfy important user needs faster with operational development,
3. Reduce costs with reduced operation software,
4. Speed up development and improve productivity by allocating resources in relation to use and criticality,
5. Guide distribution of review efforts,
6. Reduce system risk with more realistic testing,
7. Make testing faster (faster reliability growth) and more efficient,
8. Make performance evaluation and management more precise, and
9. Guide development of better manuals and training.

The cost of developing an operational profile varies. The effort to construct the operational profile for an "average" project -about 10 developers, 100,000 source lines, and a development interval of 18 months- is about one staff month. Large projects can cost more, but the increase is clearly less than linear with project size [1].

2.3.2. Development

Developing an operational profile to guide testing involves as many as five steps [1]:

1. Find the customer profile.
2. Establish the user profile.
3. Define the system-mode profile.
4. Determine the functional profile.
5. Determine the operational profile itself.

A customer is the person, group or institution that is acquiring the system. A customer group is a set of customers that will use the system in the same way. Thus, the customer profile is the complete set of customer groups and their associated occurrence probabilities. A measure for each customer group's probability is the proportion of use it represents. If this is not available, a simple approximation is to use the proportion of total deliveries that are expected to be made to that customer group.

A user is a person, group or institution that employs, not acquires, the system. A user group is a set of users that will employ the system in the same way. Thus, the user profile is the set of user groups and their occurrence probabilities. The user profile is derived from the customer profile by looking at each customer group and determining what user groups exist. If similar user groups are found among different customer groups, they should be combined. Then, the probability of each user group should be multiplied by its customer group probability to obtain its overall probability. When user groups are combined across customer groups, their overall user group probabilities should be added to yield the total user group probability.

A system mode is a set of operations that are grouped for convenience in analyzing execution behavior. A system-mode profile is the set of system modes and their associated occurrence probabilities. For each system mode, an operational profile must be determined.

The next step is to break each system mode down into the functions it needs and determine each function's occurrence probability. The functional profile provides a quantitative picture of the relative use of different functions. At this point, a choice must be made between an explicit and implicit profile because it will determine the type of the operational profile (explicit or implicit) that will be developed later. An explicit profile consists of one enumerated set of all variables taken together, with their associated occurrence probabilities. An implicit profile, on the other hand, consists of sets of the key input variables' values, with their associated occurrence probabilities. An implicit profile can be used only when the key input variables are independent (at least approximately) of each other with regard to the occurrence probabilities of their values. If they interact, an explicit profile must be developed since such occurrence probabilities must be measured or estimated directly.

Functions evolve into operations as the way the user will employ operations to accomplish functions is developed. A function may evolve into one or more operations, or a set of functions may be restructured into a different set of operations. Thus, the mapping from functions to operations is not necessarily straightforward.

The first three steps in determining the operational profile are to divide the execution into runs, identify the input space, and partition the input space into operations. A program's input space is the set of input states that can occur during its operation. Many programs are expected to take action for environmental variables such as data-entry errors, heavy traffic, or data degradation. These considerations expand the required input space that a program must respond to and be tested for. Thus, the design input space that the program has been developed to work with is not the same as the required input space, and moreover, the areas of the required input space that do not fall in the design input space will contain input states with a higher likelihood of failure. In defining the input space, the most important thing is to develop a complete list of input variables. The input space can be partitioned by grouping run types into operations. The run types that are grouped should share the same input variables so that a common and efficient testing procedure can be set up. If one or more input variables have values or value ranges with different occurrence

probabilities, multiple corresponding operations should be defined even if they share the same input variables.

2.3.3. Examples of Operational Profile

We now examine two common types of systems, command-driven and data-driven. The examples are taken from [1].

2.3.3.1. Command Driven System. The PBX is a command-driven system. In implementing the system-administration mode, this command set was developed:

reloc <old location><new location>
add -s<service grade><location>
remove <location>
dirup

Relocation is being handled by removing old location and adding a new one.

As you consider the parameters, you note that location does not affect the nature of the processing. However, the type of user does because the features provided are substantially different for staff, secretaries and managers. Therefore the add command is expanded into three operations:

reloc <old location><new location>
add -s staff<location>
add -s secretary<location>
add -s manager<location>
remove <location>
dirup

All these commands, except dirup, account for 0.019 of the occurrence probability; dirup accounts for 0.001. Suppose that the expected 80 additions of service per month break down into 70 staff, five secretaries, and five managers. There will be 780 relocations

and 20 changes of service each month, the latter representing promotions to managers. There will be 70 removals proper and 20 removals created as the result of "change" functions, yielding a total of 90 removals. The part of the operational profile for the system-administration mode is shown in Table 2.3.3.1.1.

Table 2.3.3.1.1. Part of the Operational Profile for System Mode

Command	Transactions per Month	Occurrence Probability
reloc	780	(780/970)*0.019
remove	90	(90/970)*0.019
add -s staff	70	(70/970)*0.019
dirup		0.0010
add -s manager	25	(25/970)*0.019
add -s secretary	5	(5/970)*0.019
total	970	

There will usually be environmental variables that affect the processing sufficiently to require testing based on some of their values. In the sample system, the environmental variable is telephone type: The system must handle both analog and digital telephones. Under this environmental variable, the operational profile will expand into 11 operations since directory update is not affected by telephone type. We exclude directory update and consider the part for analog telephones. Analog phones take 80 per cent of phone calls.

Assume that system load is such an environmental variable. If system-administration functions are performed when the system is in a overload condition, processing may be affected (for example, administrative requests might be queued). Assume that this occurs 0.1 per cent of the time.

To generate the expanded operational profile shown in Table 2.3.3.1.2, first we multiply the occurrence probabilities in Table 2.3.3.1.1 by 0.8 to give the occurrence probabilities for analog telephones. Then we multiply by 0.999 to obtain the occurrence

probabilities for normal load, or by 0.001 to obtain the occurrence probabilities for overload.

Table 2.3.3.1.2. Expanded Operational Profile.

Command	Environment	Occurrence Probability ($*10^{-6}$)
reloc	Normal load	12,228.00
remove	Normal load	1,359.00
add -s staff	Normal load	1,119.00
add -s manager	Normal load	400.00
add -s secretary	Normal load	79.90
reloc	Overload	12.24
remove	Overload	1.36
add -s staff	Overload	1.12
add -s manager	Overload	0.40
add -s secretary	Overload	0.08

It may not be necessary for an operation to be tested if it occurs very infrequently. As an example, "add -s secretary under overload conditions" in Table 2.3.3.1.2 can be eliminated.

2.3.4.2. Data Driven System. Financial and billing systems are commonly data-driven. Suppose a telephone billing system was designed as two subsystems. The first receives call transactions and sorts them by billing period and account number, grouping all the items for one account and the current billing period. The second processes the charge entries for each account for the current billing period and generates bills.

The reliability to be evaluated is the probability of generating a correct bill. This involves determining the reliabilities of each subsystem over the period required to process the bill or the entries associated with the bill. Then, we multiply reliabilities of the

subsystems to obtain system reliability. An operational profile must be developed for each subsystem.

The sort subsystem will have relatively few operations and thus a simple operational profile. The operation for processing correct charge items has an occurrence probability greater than 0.99; other operations handle missing data, data with recognizable errors, and so on.

The account-processing subsystem has an operational profile that relates to account attributes. Its operations are classified by service (residential or business), use of a discount calling plan (none, national or international), and payment status (paid or delinquent).

Assume that 80 per cent of the service is residential and 20 per cent is for business. A national discount calling plan is used by 20 per cent of subscribers; international, five per cent. Only one per cent of accounts are delinquent. The operational profile for the data-driven system is as follows.

Table 2.3.4.2.1. Operational Profile for Data Driven System

Operation	Occurrence Probability
Residential, no calling plan, paid	0.5940
Residential, national calling plan, paid	0.1580
Business, no calling plan, paid	0.1485
Business, national calling plan, paid	0.0396
Residential, international calling plan, paid	0.0396
Business, international calling plan, paid	0.0099
Residential, no calling plan, delinquent	0.0060
Residential, national calling plan, delinquent	0.0016
Business, no calling plan, delinquent	0.0015
Business, national calling plan, delinquent	0.0004
Residential, international calling plan, delinquent	0.0004
Business, international calling plan, delinquent	0.0001

3. REVIEW OF THE SOFTWARE RELIABILITY MODELS

3.1. Markovian Models

There are different classifications on the software reliability models. The first one is based on the Markovian property. In all the Markovian models we will be interested in the following quantities:

1. Number of failures experienced
2. Number of failures remaining
3. Failure times
4. Times between the failures.

3.1.1. General Poisson-Type Models

In these models, Musa et al. study the software failure process using a nonhomogeneous Poisson process (NHPP) with failure intensity $\lambda(t)$ [3]. Let $M(t)$ denote failures experienced by time t. $M(t)$ is distributed as a Poisson random variable, that is,

$$P[M(t)=m]=\frac{[\mu(t)]^m}{m!}\exp[-\mu(t)] \tag{3.1}$$

where

$$\mu(t)=\int_0^t \lambda(x)dx. \tag{3.2}$$

Suppose that we have experienced m_e failures by time t_e, then for $t \geq t_e$ and $m \geq m_e$

$$P[M(t)=m|M(t_e)=m_e]=\frac{[\mu(t)-\mu(t_e)]^{m-m_e}}{(m-m_e)!}\exp-[\mu(t)-\mu(t_e)] \tag{3.3}$$

which is also equal to the probability of experiencing $(m - m_e)$ failures during $(t_e, t]$.

Let $Q(t)$ be a random variable representing failures remaining at time t, that is

$$Q(t) = M(\infty) - M(t). \tag{3.4}$$

Then,

$$P[Q(t) = q] = P[M(\infty) - M(t) = q] = \frac{[\upsilon(t)]^q}{q!}\exp[-\upsilon(t)] \tag{3.5}$$

for q = 0,1,.... where

$$\upsilon(t) = \mu(\infty) - \mu(t). \tag{3.6}$$

The other two interesting random quantities are:

1. How long does it take to experience a certain number of failures?
2. What is the probability of failure-free operation during a certain amount of time (that is reliability)?

Let T_i' be a random variable representing the *ith* failure interval and define T_i as a random variable representing the *ith* failure time. Therefore,

$$T_i = \sum_{j=1}^{i} T_j' \tag{3.7}$$

where T_0 = 0. It now follows from the relationship

$$[M(t) \geq i] \Leftrightarrow [T_i \leq t] \tag{3.8}$$

that

$$P[T_i \le t] = \sum_{j=i}^{\infty} \frac{[\mu(t)]^j}{j!} \exp[-\mu(t)]. \tag{3.9}$$

This also implies that

$$P[T_i' > t_i' | T_{i-1} = t_{i-1}] = R(t_i' | t_{i-1}) = \exp\{-[\mu(t_{i-1} + t_i') - \mu(t_{i-1})]\}. \tag{3.10}$$

The conditional density function can be obtained from (3.10) as:

$$-\frac{dP[T_i' > t_i' | T_{i-1} = t_{i-1}]}{dt_i'} = f(t_i' | t_{i-1}) = \lambda(t_{i-1} + t_i') \exp\{-[\mu(t_{i-1} + t_i') - \mu(t_{i-1})]\} \tag{3.11}$$

so that the hazard rate can be expressed as

$$z(t_i' | t_{i-1}) = \lambda(t_{i-1} + t_i'). \tag{3.12}$$

3.1.2. Binomial-Type Models

The model is first presented by [3]. The assumptions are:

1. Whenever a software failure occurs, the fault that caused it will be removed instantaneously.
2. There are u_0 inherent faults in the program.
3. Each failure, caused by a fault, occurs independently and randomly in time, according to the per-fault hazard rate.

Note that u_0 will vary with the changes in the operational profile. Each failure is not in reality totally independent of every other failure but we assume that they are

independent. Let $Q(t)$ denote the number of faults remaining in the program at time t. Then

$$Q(t) = u_0 - M(t) \tag{3.13}$$

where $M(t)$ represents the number of experienced failures up to time t.

The probability of the number of failures experienced up to time t is

$$P[M(t) = m] = \begin{bmatrix} u_0 \\ m \end{bmatrix} [F_a(t)]^m [1 - F_a(t)]^{u_0 - m}. \tag{3.14}$$

The expected number of experienced failures up to time t is the mean of the binomial distribution above

$$\mu(t) = u_0 F(t). \tag{3.15}$$

The distribution of remaining faults at time t can be derived as:

$$P[Q(t) = q] = \begin{bmatrix} u_0 \\ q \end{bmatrix} [1 - F_a(t)]^q [F_a(t)]^{u_0 - q}. \tag{3.16}$$

Suppose that we have observed m_e failures up to time t_e. We can find the conditional distribution of $M(t)$ and $Q(t)$ for $t > t_e$ as follows:

$$P[M(t) = m | M(t_e) = m_e] = \begin{bmatrix} u_0 - m_e \\ m - m_e \end{bmatrix} F_a(t|t_e)^{m - m_e} [1 - F_a(t|t_e)]^{u_0 - m} \tag{3.17}$$

where

$$F_a(t|t_e) = \frac{F_a(t) - F_a(t_e)}{1 - F_a(t_e)} \tag{3.18}$$

and

$$P[Q(t) = q | M(t_e) = m_e] = \begin{bmatrix} u_0 - m_e \\ q \end{bmatrix} [1 - F_a(t|t_e)]^q [F_a(t|t_e)]^{u_0 - m_e - q}. \tag{3.19}$$

The distribution of the times between failures can be obtained as

$$P[T_i \le t] = P[M(t) \ge i] = \sum_{j=i}^{u_0} P[M(t) = j] = \sum_{j=i}^{u_0} \begin{bmatrix} u_0 \\ j \end{bmatrix} [F_a(t)]^j [1 - F_a(t)]^{u_0 - j}. \tag{3.20}$$

The conditional reliability can be obtained as

$$R(t_i'|t_{i-1}) = P[T_i - T_{i-1} > t_i' | T_{i-1} = t_{i-1}] = \exp\left\{ -(u_0 - i + 1) \int_{t_{i-1}}^{t_{i-1} + t_i'} z_a(x) dx \right\} \tag{3.21}$$

and the conditional hazard rate function can be found as

$$z(t_i'|t_{i-1}) = (u_0 - i + 1) z_a(t_{i-1} + t_i'). \tag{3.22}$$

3.1.3. Poisson-Type Models (Finite Failures)

Assumption (3) of the binomial models also applies here. The total number of remaining faults in the program at $t = 0$ is now a random variable with a mean μ as denoted in [3].

The probability that the number of failures experienced up to time t is equal to m is:

$$P[M(t)=m]=\sum_{x=0}^{\infty}P[M(t)=m|U_0=x]P[U_0=x]$$

$$=\sum_{x=0}^{\infty}\begin{bmatrix}x\\m\end{bmatrix}[F_a(t)]^m[1-F_a(t)]^{x-m}\frac{\mu^x}{x!}\exp[-\mu]$$

$$=\frac{[\mu F_a(t)]^m}{m!}\exp[-\mu F_a(t)]. \tag{3.23}$$

Therefore the expected number of failures experienced up to time t is a Poisson random variable with mean

$$\mu(t)=\mu F(t). \tag{3.24}$$

Let $\upsilon(t)$ denote the rate of the remaining failures, we obtain that from the previous equation:

$$\upsilon(t)=\mu(\infty)-\mu(t)$$

$$=\mu-\mu F(t). \tag{3.25}$$

The conditional reliability can be obtained as

$$R(t_i'|t_{i-1})=\exp\{-\mu[F_a(t_{i-1}+t_i')-F_a(t_{i-1})]\} \tag{3.26}$$

and the program hazard rate is

$$z(t_i'|t_{i-1})=\mu f_a(t_{i-1}+t_i'). \tag{3.27}$$

3.2. Times Between Failures Models

This type of models assume that the failure rate depends on the number of the remaining faults in the software after the most recent failure.

3.2.1. The Model of Jelinski and Moranda

The model of Jelinski and Moranda assumes a fixed number N of faults at the start of testing, each is independent of others and is equally likely to cause a failure during testing [5]. There is perfect debugging and the hazard rate, at any time is assumed to be proportional to the current number of faults in the program. For the time interval between the ith and the $(i-1)th$ failures the hazard rate is given by

$$z(t_i') = \lambda[N-(i-1)] \tag{3.28}$$

where λ is the failure rate per fault and t_i' is the passed time since the ith failure.

3.2.2. Littlewood and Verrall Model

In the model of Littlewood and Verrall, the times between failures are assumed to follow an exponential distribution with a parameter that is gamma distributed [6]. Now,

$$r(t_i') = \frac{\alpha}{t_i' + \Psi(i)} \tag{3.29}$$

where $\Psi(i)$ describes the quality of the programmer and the difficulty of the programming task.

3.2.3. Goel and Okumoto's Imperfect Debugging Model

In [7], Goel and Okumoto propose a model which is basically an extension of the model proposed in [5]. In this model, the number of faults in the program at time t is treated as a Markov process. The hazard rate during the interval between $(i-1)th$ and the ith failures is

$$z(t_i') = \lambda[N-p(i-1)] \tag{3.30}$$

where N is the initial number of faults, p is the probability of imperfect debugging, and λ is the failure rate per fault.

3.2.4. Schick and Wolverton Model

In [8], Schick and Wolverton propose a model which is based on the same assumptions as the model proposed in [5], except that the hazard rate is assumed to be proportional to the current number of faults in the program as well as to the time elapsed since the last failure. The hazard rate is given by

$$z(t_i') = \lambda[N - (i-1)]t_i'. \tag{3.31}$$

3.3. Fault Count Models

This class of models is concerned with the number of failures seen in a testing interval. The basic process behind most of these models is Poisson process.

3.3.1. Goel and Okumoto's Nonhomogeneous Poisson Process Model

In this model, the number of failures observed until time t is modeled as a nonhomogeneous Poisson process with the mean value function of

$$\mu(t) = a(1 - e^{-bt}) \tag{3.32}$$

where a represents the expected number of failures to be observed eventually and b represents the fault detection rate per fault [9].

3.3.2. Goel Generalized Nonhomogeneous Poisson Process Model

In [10], Goel proposes a generalized model based on the nonhomogeneous Poisson process model in [9], to allow for modeling the observation that the failure rate first increases and then decreases. The mean value function of the model is given by

$$\mu(t) = a\left(1 - e^{-bt^c}\right) \tag{3.33}$$

where a represents the expected number of faults to be eventually detected, and b and c are constants that reflect the quality of testing.

3.3.3. Musa Execution Time Model

In [11], Musa proposes a model which is based on the model in [5], but there is dependence between failure rate and the execution time of the model. The hazard rate for the model is given by

$$z(\tau) = \phi f\left(N - n_C\right) \tag{3.34}$$

where τ is the execution time in executing the program up to the present, f is the average instruction execution rate divided by the number of instructions in the program, ϕ is the fault exposure ratio, and n_C is the number of faults removed during $(0, \tau)$.

3.3.4. Musa-Okumoto Logarithmic Poisson Execution Time Model

In this model, the observed number of failures by some time τ is modeled as a nonhomogeneous Poisson process with a mean value function which is a function of τ [12]. Thus, it is given as

$$\mu(\tau) = \frac{1}{\theta}\ln\left(\lambda_0 \theta \tau + 1\right) \tag{3.35}$$

where λ_0 and θ represent the initial failure intensity and the rate of the reduction in the normalized failure intensity per failure, respectively.

3.4. Fault Seeding Models

The approach to assessing the reliability is to seed a known number of faults in the program. After testing, the number of detected seeded and inherent faults are counted. Using combinatorics and maximum likelihood estimation, the reliability of the software and the number of inherent faults in the program can be estimated.

In the hypergeometric model of Mills, the number of inherent faults is estimated from the number of inherent and seeded faults detected during the test by using the hypergeometric distribution [13].

In the model of Lipow, the approach of the above model is modified to include the probabilities of finding a fault of either kind during any test of the software so that for statistically independent tests, the probability of finding given numbers of inherent and seeded faults can be calculated [14].

In the model of Basin, it is suggested to use two programmers to develop a two stage procedure of testing without actually seeding faults. The number of faults detected by each programmer is used to estimate the software reliability [15].

4. OPTIMUM SOFTWARE RELEASE POLICIES

An important issue in software engineering is the determination of the optimal release time of a software system. Before a software is released for operational use, it is tested to decrease the number of inherent faults and increase its reliability. However, the testing procedure is time consuming and costly. On the other hand, releasing a software without sufficient testing causes high maintenance costs after release. Hence, there must be a trade-off between the testing cost and the maintenance cost after release. A consequence of this trade-off is the development of stopping rules for the testing process which are based on the reliability of the software after testing and the comparative costs of a fault occurrence during testing and operation.

Software release policies can be classified as

1. Static policies;
2. Dynamic policies.

In static policies, a release time is determined at the beginning of the testing phase, and once testing is started, the release time remains unchanged. Thus, in static policies the information about the true state of the software system gathered during testing is ignored. On the other hand, in dynamic policies the release time is determined by using the observed history of the failure process. The decision to release the software or not is reevaluated after each failure observation and the consequent debugging activity.

4.1. Static Policies

In [16], Forman and Singpurwalla suggest a procedure for the estimation of the model parameters and a stopping rule for debugging the program. The failure process of software is modeled by [5]. The proposed stopping rule is based on the maximum likelihood estimator of inherent faults in the software. The steps of the procedure are as follows:

1. Compute $\hat{N}$, the maximum likelihood estimator of inherent faults N.

2. If $\hat{N} \approx n$ (the number of the observed failures), go to step 3, else wait for another failure and go to step 1.

3. Plot the relative likelihood function of N, $R(N)$, and the normal relative likelihood function of N, $R_{normal}(N)$, for various values of N. If the two functions are in agreement, then $\hat{N}$ is a good estimator of N. Otherwise, wait for another failure and go to step 1.

Since the maximum likelihood estimate of N is asymptotically efficient, it can be approximated by the normal distribution for a large number of observations. Thus a comparison of the actual relative likelihood $R(N)$ and the approximate normal relative likelihood $R_{normal}(N)$ gives some indication of the appropriateness of the large sample theory for estimating N.

After applying the procedure above to estimate N and the failure rate per fault λ, Forman and Singpurwalla use the hypothesis testing to determine the optimal stopping time [17].

$$H_0: N^* = 0 \qquad H_1: N^* = r \quad r = 1,2,... \tag{4.1}$$

where N^* represents the remaining faults in the software. In order to test the above hypothesis, the software will be exercised for an additional time t_a, and H_0 will be rejected if a failure is encountered. For such a test, the Type I error (the probability of rejecting H_0 when it is true) is zero, since if N^*=0, we will not encounter any failures. The additional time t_a can be obtained from the predetermined Type II error β by setting $r = 1$:

$$\beta = e^{-\lambda r t_a}. \tag{4.2}$$

Another way to determine t_a is to minimize the following cost function:

$$E[C] = \int_0^{t_a} c_1(t) t \lambda e^{-\lambda t} dt + \int_{t_a}^{t_a+t_m} \left(c_1(t) t_a + c_2\right) \lambda e^{-\lambda t} dt + \int_{t_a+t_m}^{\infty} c_1(t) t_a \lambda e^{-\lambda t} dt \qquad (4.3)$$

where c_1 is the cost of testing per unit time, c_2 is the cost of failure in the operational phase, and t_m is the life cycle length of the software.

Koch and Kubat suggest a model to determine the optimal release time of a software by considering the cost-benefit function of the entire company [18]. The model is based on the model in [5]. There are two ways that the software can be used:

1. The company which developed it can use the software itself
2. The company which is going to use the software is a second party.

The cost function is as follows:

$$c(t) = \psi_1(t) I_{[0,D)}(t) + \psi_2(t) I_{[D,\infty)}(t) \qquad (4.4)$$

with

$$\begin{aligned} \psi_1(t) &= c_1 M e^{-\lambda t} + \left(c_2 + c_4 - c_3 + b_1 + b_2\right) t + \left(c_3 - b_1 - b_2\right) D \\ \psi_2(t) &= c_1 M e^{-\lambda t} + \left(c_2 + c_4\right) t + p(t) \end{aligned} \qquad (4.5)$$

where I is the indicator function. The cost parameters are as follows: c_1 is the cost of debugging in the operational phase, c_2 is the cost of the CPU time per unit testing time, c_3 is the cost of using a software per unit time, c_4 is the cost of manpower per unit time, b_1 is the benefit from the freed manpower after release, b_2 is the benefit of using the software, D is the deadline for system release, $p(t)$ is the penalty cost if the software is released after the deadline, and M is the remaining faults in the program. If the optimal release time is found to be after the deadline, it is advised to test the software until the deadline and then reconsider to continue further testing if the costs of future testing plus penalty are less than the debugging cost of the remaining faults in the operational phase.

Yamada and Osaki determine the optimal software release time by evaluating the following two criteria simultaneously [19]:

- Total average software cost;
- Software reliability.

They investigate the problem by using the nonhomogeneous Poisson process (NHPP) model in [9]. The optimal software release problem can be formulated as follows:

$$\begin{aligned} &\min c(t) \\ &s.t.\ R(x|T) \geq R_0 \end{aligned} \tag{4.6}$$

where R_0 denotes the desired reliability level. The optimal release time can be obtained from $T^* = \max\{T_0, T_1\}$ where T_0 is the optimal release time satisfying the cost criterion and T_1 is the optimal release time satisfying the reliability criterion. In other words,

$$T_0 = \begin{cases} \dfrac{1}{b}\ln\left[\dfrac{ab(c_2 - c_1)}{c_3}\right], & ab(c_2 - c_1) > c_3 \\ 0, & otherwise \end{cases}$$

$$T_1 = \begin{cases} \dfrac{1}{b}\left[\ln[m(x)] - \ln\left[\ln\left(\dfrac{1}{R_0}\right)\right]\right], & R(x/0) < R_0 \\ 0, & otherwise. \end{cases} \tag{4.7}$$

The parameters of the above formulas are as follows: a is the number of inherent faults, b is the fault detection rate per fault, c_1 is the cost of debugging during testing, c_2 is the cost of debugging during operational phase, c_3 is the cost of testing per unit time, and $m(x)$ is the expected number of faults removed up to time x.

Ross proposes a stopping rule by considering the estimation of the failure rate $\varepsilon(t)$ of software [20]. It is advised to discontinue testing at the first time t such that

$$\varepsilon(t) + 3\sqrt{E\left[\left(\varepsilon(t) - \Lambda(t)\right)^2\right]} < A \tag{4.8}$$

where A denotes the acceptable level of the failure rate and $\Lambda(t)$ denotes the failure rate of the software. The estimation of the above formula will be inaccurate for small values of t so that the testing should be continued for at least some fixed time T_0 which can be obtained from

$$\begin{aligned} e^{-AT_0} &= \alpha \\ T_0 &= -\frac{\ln \alpha}{A} \end{aligned} \tag{4.9}$$

where α represents the acceptable level of risk.

Singpurwalla describes a procedure for addressing the important problem of how long to test and debug software before it is released [21]. The procedure is based on the principles of decision making under uncertainty, and involves a maximization of expected utility. He uses two different types of utility function: one based on costs alone, and the other involving the realized reliability of the software.

Wohlin and Runeson describe methods that can be used to certify and measure the ability of software components to fulfil the reliability requirements placed on them [22]. A usage modelling to formulate usage models for components is presented. The usage model describes the usage from a structural point of view, complemented with a profile describing the expected usage in figures. The failure statistics from the usage test form the input of a hypothesis certification model, which makes it possible to decide whether the software component can be accepted with a given degree of confidence. The conclusion is that the proposed method makes it possible to certify software components, both when developing for and with reuse.

Hou et al. developed the hyper-geometric distribution software reliability growth model to estimate the number of remaining faults after completing the test/debug phase [23]. The optimum software release time is determined by the minimization of the total expected software costs which include penalty cost that should be paid in case of a late delivery. The underlying model is the hyper-geometric distribution model.

4.2. Dynamic Policy

Özekici and Çatkan propose a dynamic procedure to determine the optimal release time under imperfect debugging [24]. The procedure is based on a cost model constructed by considering testing cost, debugging cost during testing phase and debugging cost after release. The dynamic policy proposes to test the software for an additional amount of time after the occurrence of a failure and the following debugging activity. The length of the additional testing time can be found by minimizing the expected total cost using dynamic programming. If no failure occurs during the proposed additional testing time, the software is released for operational use. If, however, a failure is observed, then a new debugging activity is undertaken and a new additional testing time is computed by using the updated information on the expected number of faults remaining in the software.

The dynamic programming equation is given by

$$v(n) = \min_{u \geq 0} \left\{ c_1 E\left[X_n \wedge u\right] + c_2 h(n) \overline{F}_n(u) + \left[c_3 + v(n+1)\right] F_n(u) \right\} \tag{4.10}$$

$$E\left[X_n \wedge u\right] = u\overline{F}_n(u) + \int_{[0,u)} sF_n(ds). \tag{4.11}$$

where $v(n)$ represents the minimum expected total cost using the optimal policy after the nth failure. The first term represents the expected cost of testing at the nth stage since the software is tested until the release time or the occurrence of a failure. In other words, the cost $c_1 u$ is incurred if no failure is observed during u time units of testing, and the cost

$c_1 E[X_n]$ is incurred if the interfailure time turns out to be less than the specified additional testing time u at the *nth* stage of testing.

The second term represents the expected total cost of debugging which will be experienced in the operational phase if the software is released at the *nth* stage. It is assumed that the life cycle of the software is infinite so that all remaining faults will manifest themselves through failures.

The third term accounts for the expected cost incurred in the case of a failure observation before u. It is composed of the cost of debugging and the expected total cost which will be incurred from this stage onward using the optimal policy.

The model is based on the following assumptions:

1. The failure rate is decreasing in t for all $n \geq 0$.
2. The failure rate is decreasing in n for all $t \geq 0$.
3. The expected number of faults remaining after the *nth* failure is convex decreasing in $n \geq 0$.

The dynamic programming equation presented in (4.10) can be solved by a backward recursive approach for the u values minimizing the cost function at each stage until the optimal cost is obtained.

5. THE OPTIMUM SOFTWARE RELEASE PROBLEM

In this section, a new software reliability model with imperfect debugging will be presented. Our model is an extension of the model developed in [25] where one of the main assumptions is perfect debugging. Moreover, in our study the time-to-failure has an arbitrary distribution. Under these conditions, we determine the optimal testing strategy by using a nonlinear expected total cost function.

5.1. Notation

We now introduce the notation used throughout the thesis. Suppose that there are a total of K operations that can be performed by software. All parameters of the model depend on the operations as given by the following notation:

$F_i(.)$: time-to-failure distribution during operation i,

e_i: debugging cost after release during operation i,

c_i: testing cost per unit time during operation i,

d_i: debugging cost during operation i,

π_i: the probability that the software will perform operation i after release,

p_i: the probability of removing a fault during testing phase in operation i,

μ: the expected number of inherent faults in the software before testing,

t_i: duration of testing for operation i.

5.2. Assumptions

The model to determine the optimal testing strategy under imperfect debugging condition is based on the following assumptions:

Assumption 1 a) Whenever a software failure occurs, the fault that caused the failure will be removed instantaneously with a predetermined probability $0 \leq p_i \leq 1$ during operation i,

b) No new faults are introduced during debugging,

c) There are a finite number of faults in the program, the total number of faults in the program at time $t = 0$ is a random variable with mean μ,

d) Each failure, caused by a fault, occurs independently and randomly in time according to the time-to-failure distribution $F_i(.)$ during operation i,

e) $e_i, c_i, d_i \geq 0$,

f) A fixed sequence of testing is given.

The first assumption implies the imperfect debugging activity. Each fault is removed with probability p_i or remains in the program with probability $q_i = (1 - p_i)$ during operation i. Another implication of this assumption is that the number of failures experienced up to a certain time t is not equal to the number of faults removed by the same time.

The purpose of the second assumption is to ensure that the modeled failure process does have a monotonic pattern. In general, this may not be true if faults are debugged after each occurrence because other paths may be affected during debugging, leading to additional faults in the system. The only way to strictly satisfy this is to ensure that the correction process does not introduce new faults. If, however, the additional faults introduced constitute a very small fraction of the fault population, the practical effect on model results would be minimal [26].

The third assumption indicates that the total number of faults in the program is an extraordinarily complex function of many factors such as program size, complexity and human performance. Hence, it is reasonable to consider the total number of faults in the program as a random variable rather than assigning a fixed value at the beginning.

The fourth assumption implies that each failure is totally independent of every other failure. But in reality one failure can influence succeeding failures by its individual impact on the effectiveness of repair. Also, a failure may prevent another failure from occurring because it may prevent a particular code path from being executed. However the effects

are, on the average, secondary and will be neglected [3]. We can rewrite this assumption more formally as follows: Let T_i denote a random variable representing time to failure of a fault during operation i. Then, the random variables $\{T_i\}$ are independently and identically distributed as $F_i(.)$ for all remaining faults. We can obtain the cumulative distribution function $F_i(t)$ from the per-fault hazard rate function $z_i(t)$ by

$$F_i(t) = 1 - \exp\left[\int_0^t z_i(x)dx\right] \tag{5.1}$$

where

$$z_i(t) = \frac{dF_i(t)/dt}{1 - F_i(t)}. \tag{5.2}$$

A fixed sequence of testing indicates that testing is to be performed in a prescribed order 1,2,3,....,K of operations with durations $t_1, t_2, t_3, ..., t_K$.

5.3. The General Model

We will now derive some computational formulas on the expected number of faults removed during testing in an operation. This will be done in the context of an ordinary renewal process $N = \{N_t; t \geq 0\}$ with some interrenewal distribution F. The renewal process represents the failure process caused by a fault provided that the debugging activities are not successful. If, however, debugging after a failure is successful with probability p, then the renewal (failure) process is stopped since there can be no more failures. We will count the failures by the process $M = \{M_t; t \geq 0\}$. As soon as a failure is counted with probability p, that is the fault that caused the failure is removed, the process $M = \{M_t; t \geq 0\}$ is stopped. Therefore, the process $M = \{M_t; t \geq 0\}$ gives us the number of failures generated by a fault in the program up to time t.

First, we will determine the expected number of failures caused by a fault in the program up to time t. We derived the following conditional probability function of the number of failures counted given that n failures have been generated by $N = \{N_t; t \geq 0\}$: For $n \geq 1$,

$$\begin{aligned}
&P[M_t = 0 | N_t = n] = 0, \\
&P[M_t = 1 | N_t = n] = p, \\
&P[M_t = 2 | N_t = n] = qp, \\
&P[M_t = 3 | N_t = n] = q^2 p, \\
&\cdot \\
&\cdot \\
&P[M_t = n-1 | N_t = n] = q^{n-2} p, \\
&P[M_t = n | N_t = n] = q^{n-1}.
\end{aligned} \tag{5.3}$$

We don't consider the trivial case of $n = 0$ since we know that:

$$P[M_t = 0 | N_t = 0] = 1. \tag{5.4}$$

It can be shown that the conditional probability function given in (5.3) is a probability distribution function since

$$\begin{aligned}
\sum_{k=0}^{\infty} P[M_t = k | N_t = n] = \sum_{k=0}^{n} P[M_t = k | N_t = n] &= q^{n-1} + p \sum_{k=0}^{n-2} q^k \\
&= q^{n-1} + p \frac{(1 - q^{n-1})}{(1-q)} = 1
\end{aligned} \tag{5.5}$$

after recalling that $p = 1 - q$.

By using the conditional probability distribution function in (5.3), we can now determine the expected number of failures counted up to time t given the number of failures generated by $\{N_t; t \geq 0\}$:

$$
\begin{aligned}
E\left[M_t | N_t = n\right] &= \sum_{k=0}^{n} kP\left[M_t = k | N_t = n\right] \\
&= p\sum_{k=0}^{n-2}(k+1)q^k + nq^{n-1} = pq\sum_{k=0}^{n-2} kq^{k-1} + p\sum_{k=0}^{n-2} q^k + nq^{n-1} \\
&= pq\left[\frac{-(n-1)q^{n-2}(1-q)+(1-q^{n-1})}{(1-q)^2}\right] + p\frac{(1-q^{n-1})}{(1-q)} + nq^{n-1} \\
&= 1+\frac{q}{p}(1-q^{n-1}).
\end{aligned}
\tag{5.6}
$$

We can therefore write

$$
E\left[M_t | N_t\right] = 1+\frac{q}{p}(1-q^{N_t-1}) \tag{5.7}
$$

and the expression in (5.7) is true whenever $p \neq 0$. Furthermore, if $p = 1$, the expression in (5.7) will be equal to one when $N_t \geq 1$, and to zero when $N_t = 0$.

The expected number of failures caused by a fault in the program up to time t can be obtained from (5.7) as follows:

$$
\begin{aligned}
E\left[M_t\right] &= E\left[E\left[M_t | N_t\right]\right] \\
&= 1+\frac{q}{p}\left[1-\frac{1}{q}E\left[q^{N_t}\right]\right] = \frac{1}{p}\left[1-E\left[q^{N_t}\right]\right].
\end{aligned}
\tag{5.8}
$$

Let

$$
E\left[q^{N_t}\right] = G(t), \tag{5.9}
$$

we can then rewrite the expected number of failures caused by a fault up to time t as

$$
E\left[M_t\right] = \frac{1}{p}\left[1-G(t)\right]. \tag{5.10}
$$

Recall that the expected number of inherent faults in the program is μ. Thus, the expected number of failures experienced up to time t is given by

$$\frac{\mu}{p}[1-G(t)] \tag{5.11}$$

so that the expected number of failures experienced up to time t_i is

$$\frac{\mu}{p_i}[1-G_i(t_i)] \tag{5.12}$$

where G_i is the generating function (5.9) defined for a renewal process with interrenewal distribution F_i .

We assume that $\{N_t; t \geq 0\}$ is a recurrent renewal process. The implications of the assumption are:

$$F(\infty) = \lim_{t\to\infty} F(t) = 1 \tag{5.13}$$

and

$$N(\infty) = \lim_{t\to\infty} N_t = +\infty \tag{5.14}$$

with probability one. As a consequence, the limiting behavior of $G(t)$ can be stated as follows:

$$\lim_{t\to\infty} G(t) = \lim_{t\to\infty} E[q^{N_t}] = E[q^{N_\infty}] = E[q^{\infty}] = 0 \tag{5.15}$$

whenever q is in the interval $(0,1)$.

Another assumption for the process $\{N_t; t \geq 0\}$ is that no failure has occurred at time $t = 0$, i.e. $N_0 = 0$. Consequently, the value of the function $G(t)$ at $t = 0$ is

$$G(0) = E[q^{N_0}] = E[q^0] = 1. \tag{5.16}$$

whenever $q \neq 0$.

We need now to derive an expression for the expected number of faults removed up to time t. Let T denote the time at which a fault is removed. Then, the distribution of T is

$$P[T \leq t] = H(t) = p\sum_{k=1}^{\infty} q^{k-1} F^{*k}(t) \tag{5.17}$$

where F^{*k} represents the k-fold convolution of the time-to-failure distribution F with itself.

There exists a relation between the functions $H(t)$ and $G(t)$ such that:

$$P(T > t) = E[P(T > t / N_t)] = E[q^{N_t}]. \tag{5.18}$$

The above relation indicates that given that N_t failures have occurred, the N_t independent debugging activities have failed to remove the fault, each with a probability q, up to time t. We can rewrite the relation in (5.18) as follows:

$$H(t) = 1 - G(t). \tag{5.19}$$

Using (5.19), the expected number of faults removed up to time t is

$$\mu(1 - G(t)) \tag{5.20}$$

so that the expected number of faults removed up to time t_i is

$$\mu\left(1-G_i\left(t_i\right)\right). \tag{5.21}$$

Using the derivations above we see that the expected number of remaining faults before testing in operation i is

$$\mu\prod_{j=1}^{i-1}G_j\left(t_j\right). \tag{5.22}$$

The expected number of failures experienced during the *ith* operation is given by (using (5.22))

$$\frac{\mu}{p_i}\left[1-G_i\left(t_i\right)\right]\prod_{j=1}^{i-1}G_j\left(t_j\right). \tag{5.23}$$

The expected number of faults removed during the *ith* operation is given by (using (5.22))

$$\mu\left[1-G_i\left(t_i\right)\right]\prod_{j=1}^{i-1}G_j\left(t_j\right). \tag{5.24}$$

Using equation (5.23) we can now derive the expected total cost of debugging during the testing period as

$$\mu\sum_{i=1}^{K}\frac{d_i}{p_i}\left[1-G_i\left(t_i\right)\right]\prod_{j=1}^{i-1}G_j\left(t_j\right). \tag{5.25}$$

After the testing period, we know that the software will be released and will be used in environment i with probability π_i. The expected number of remaining faults after whole testing period is

$$\mu\left\{1-\sum_{i=1}^{K}\left[1-G_i(t_i)\right]\prod_{j=1}^{i-1}G_j(t_j)\right\} \tag{5.26}$$

and the expected total cost of debugging under operation n is given by

$$\mu e_n\left\{1-\sum_{i=1}^{K}\left[1-G_i(t_i)\right]\prod_{j=1}^{i-1}G_j(t_j)\right\}. \tag{5.27}$$

and the expected total cost of debugging after release using equation (5.27) is

$$\mu\sum_{n=1}^{K}\pi_n e_n\left\{1-\sum_{i=1}^{K}\left[1-G_i(t_i)\right]\prod_{j=1}^{i-1}G_j(t_j)\right\}. \tag{5.28}$$

The total testing cost is

$$\sum_{i=1}^{K}c_i t_i. \tag{5.29}$$

Combining equations (5.25), (5.28) and (5.29), the expected total cost is

$$\sum_{i=1}^{K}c_i t_i+\mu\sum_{n=1}^{K}\pi_n e_n\left\{1-\sum_{i=1}^{K}\left[1-G_i(t_i)\right]\prod_{j=1}^{i-1}G_j(t_j)\right\}+\mu\sum_{i=1}^{K}\frac{d_i}{p_i}\left[1-G_i(t_i)\right]\prod_{j=1}^{i-1}G_j(t_j). \tag{5.30}$$

Let

$$e=\sum_{n=1}^{K}\pi_n e_n. \tag{5.31}$$

Using equations (5.30) and (5.31), the objective function turns out to be

$$\sum_{i=1}^{K} c_i t_i - \mu \sum_{i=1}^{K} \left(e - \frac{d_i}{p_i} \right) \left[1 - G_i(t_i) \right] \prod_{j=1}^{i-1} G_j(t_j) + \mu e \tag{5.32}$$

which can be further written as

$$\sum_{i=1}^{K} \left[c_i t_i - \mu f_i \left[1 - G_i(t_i) \right] \prod_{j=1}^{i-1} G_j(t_j) \right] + \mu e \tag{5.33}$$

where

$$f_i = \left(e - \frac{d_i}{p_i} \right). \tag{5.34}$$

Setting

$$x_i = G_i(t_i) \tag{5.35}$$

and taking the inverse of the function $G_i(t_i)$ in (5.35), we get

$$t_i = G_i^{-1}(x_i). \tag{5.36}$$

Using equations (5.33) and (5.36), we now obtain

$$\sum_{i=1}^{K} \left[c_i G_i^{-1}(x_i) - \mu f_i (1 - x_i) \prod_{j=1}^{i-1} x_j \right] + \mu e. \tag{5.37}$$

By setting

$$h_i(x_i) = G_i^{-1}(x_i) \tag{5.38}$$

and defining

$$b_i = \mu f_i \tag{5.39}$$

equation (5.37) turns out to be

$$\sum_{i=1}^{K}\left[c_i h_i(x_i) - b_i(1-x_i)\prod_{j=1}^{i-1} x_j\right] + \mu e. \tag{5.40}$$

Since equation (5.40) represents the total expected cost of testing, it must be minimized. Our problem is

$$\min_{0\le x_i\le 1}\sum_{i=1}^{K}\left[c_i h_i(x_i) - b_i(1-x_i)\prod_{j=1}^{i-1} x_j\right] \tag{5.41}$$

which is also equivalent to

$$\max_{0\le x_i\le 1}\sum_{i=1}^{K}\left[-c_i h_i(x_i) + b_i(1-x_i)\prod_{j=1}^{i-1} x_j\right]. \tag{5.42}$$

5.4. Determination of the Optimum Testing Policy for the Operational Profile

We will first review some basic concepts on nonlinear optimization. Given a point x in R^n, we wish to determine, if possible, whether or not the point is a local or a global maximum of a function f. There are first-order and second-order necessary conditions using the Hessian matrix for x^* to be a global maximum.

Suppose that $f: R^n \to R^1$ is differentiable at x^*. If x^* is a local maximum, then $\nabla f(x^*) = 0$ where $\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_n}\right)$.

Suppose that $f: R^n \to R^1$ is twice differentiable at x^*. If x^* is a local maximum, then $\nabla f(x^*) = 0$ and $x^T H^f x \le 0 \ \ \forall x \in R^n$ where H^f is the Hessian matrix defined as

$$H_{ij}{}^f(x) = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1{}^2} & \frac{\partial^2 f(x)}{\partial x_1 x_2} & \cdots\cdots & \frac{\partial^2 f(x)}{\partial x_1 x_n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{\partial^2 f(x)}{\partial x_n x_1} & \frac{\partial^2 f(x)}{\partial x_n x_2} & \cdots\cdots & \frac{\partial^2 f(x)}{\partial x_n{}^2} \end{bmatrix} \tag{5.43}$$

The conditions above are necessary conditions; that is, they must be true for every local optimal solution. $\nabla f(x^*) = 0$ is also sufficient for x^* to be a global maximum if and only if f is concave at x^*. If $\nabla f(x^*) = 0$ and $H(x)$ is negative semi-definite for all x (i.e. $x^T H^f x \le 0$), then f is concave. As a result, x^* is a global maximum.

Let $S = [0,1]^K$ and for any $x = (x_1, x_2, \ldots\ldots, x_n) \in S$, define

$$L(x) = \sum_{i=1}^{K} \left[-c_i h_i(x_i) + b_i(1 - x_i) \prod_{j=1}^{i-1} x_j \right]. \tag{5.44}$$

If we let

$$g_k(x) = g_k(x_1, x_2, \ldots\ldots, x_k) = -c_k h_k(x_k) + b_k(1 - x_k) \prod_{j=1}^{k-1} x_j \tag{5.45}$$

it then follows that

$$L(x) = \sum_{i=1}^{K} g_i(x) \tag{5.46}$$

and

$$H^L = \sum_{i=1}^{K} H^{g_i} \tag{5.47}$$

where H^L and H^{g_i} denote the Hessian matrices corresponding to L and g_i respectively.

Proposition 1 Suppose that $b_1 \geq b_2 \geq \ldots\ldots \geq b_K$ and, $h_k(.)$ is twice differentiable and convex decreasing on S, then L is negative semi-definite on S; i.e.,

$$x^T H^f x \leq 0 \tag{5.48}$$

for all $x \in S$.

Proof Using equation (5.45) we get

$$\frac{\partial g_k(x_1, x_2, \ldots, x_k)}{\partial x_i} = \begin{cases} b_k(1-x_k)\prod_{j=1, j\neq i}^{k-1} x_j, & i = 1,2,\ldots,k-1 \\ -c_k h_k'(x_k) - b_k \prod_{j=1}^{k-1} x_j, & i = k \\ 0, & i = k+1, k+2, \ldots, K \end{cases} \tag{5.49}$$

so that we determine the value of the $(i,j)th$ entry of the Hessian matrix as

$$H^{g_k}{}_{ij}(x) = \begin{cases} b_k(1-x_k)\prod_{m=1, m\neq i,j}^{k-1} x_m, & i,j = 1,2,\ldots,k-1, i \neq j \\ -b_k \prod_{m=1, m\neq i}^{k-1} x_m, & i = 1,2,\ldots,k-1, j = k \\ -b_k \prod_{m=1, m\neq i,j}^{k-1} x_m, & i = k, j = 1,2,\ldots,k-1 \\ -c_k h_k''(x_k), & i,j = k \\ 0, & \textit{otherwise.} \end{cases} \tag{5.50}$$

After some manipulations we get

$$x^T H^{g_k} x = -c_k h_k''(x_k) x_k^{\,2} + (k-1)(k-2) b_k \prod_{m=1}^{k-1} x_m - k(k-1) b_k \prod_{m=1}^{k} x_m . \tag{5.51}$$

Using the fact that

$$\begin{aligned} H^L &= \sum_{i=1}^{K} H^g \\ x^T H^L x &= \sum_{k=1}^{K} x^T H^{g_k} x \end{aligned} \tag{5.52}$$

we obtain

$$\begin{aligned} x^T H^L x &= -\sum_{k=1}^{K} c_k h_k''(x_k) x_k^{\,2} + \sum_{k=1}^{K} (k-1)(k-2) b_k \prod_{m=1}^{k-1} x_m - \sum_{k=1}^{K} k(k-1) b_k \prod_{m=1}^{k} x_m \\ &= -\sum_{k=1}^{K} c_k h_k''(x_k) x_k^{\,2} + \sum_{i=1}^{K-1} i(i-1) b_{i+1} \prod_{m=1}^{i} x_m - \sum_{k=1}^{K} k(k-1) b_k \prod_{m=1}^{k} x_m \\ &= -\sum_{k=1}^{K} c_k h_k''(x_k) x_k^{\,2} + \sum_{i=2}^{K-1} i(i-1)(b_{i+1} - b_i) \prod_{m=1}^{i} x_m - K(K-1) b_K \prod_{m=1}^{K} x_m . \end{aligned} \tag{5.53}$$

This completes our proof since all terms are non-positive for all $x \in S$.

As examples, two cases will be examined. The first case is for the model with a single operation so that

$$\begin{aligned} &\max_{0 \le x_1 \le 1} -c_1 h_1(x_1) + b_1 (1 - x_1) \\ &\nabla g(x_1) = -c_1 h_1'(x_1) - b_1 \end{aligned} \tag{5.54}$$

and

$$H^L(x) = -c_1 h_1'(x_1) \tag{5.55}$$

such that

$$x^T H^L x = x_1\left[-c_1 h_1'(x_1)\right]x_1 \le 0. \tag{5.56}$$

In the second case, K=2 so that

$$\max_{0\le x_1,x_2\le 1} -c_1 h_1(x_1) - c_2 h_2(x_2) + b_1(1-x_1) + b_2(1-x_2)x_1 \tag{5.57}$$

$$\begin{aligned}\frac{\partial g(x)}{\partial x_1} &= -c_1 h_1'(x_1) - b_1 + b_2 - b_2 x_2 \\ \frac{\partial g(x)}{\partial x_2} &= -c_2 h_2'(x_2) - b_2 x_1\end{aligned} \tag{5.58}$$

$$H^L = \begin{bmatrix} -c_1 h_1'(x_1) & -b_2 \\ -b_2 & -c_2 h_2''(x_2) \end{bmatrix} \tag{5.59}$$

such that

$$x^T H^L x = -c_1 h_1'(x_1){x_1}^2 - b_2 x_1 x_2 - b_2 x_1 x_2 - c_2 h_2''(x_2){x_2}^2 \le 0 \tag{5.60}$$

for all $x \in S$.

Proposition 2 Suppose that $G_k(.)$ is convex decreasing on S, then $h_k(.)$ is also convex decreasing on S.

Proof Using equations (5.35) and (5.38)

$$h_k(x_k) = G_k^{-1}\left(G_k(t_k)\right) = t_k \tag{5.61}$$

and taking the derivative,

$$h_k'(x_k) = \left[G_k^{-1}\left(G_k(t_k)\right)\right]' = (t_k)' \tag{5.62}$$

we obtain

$$h_k'(x_k) = \left(G_k(t_k)\right)'\left[G_k^{-1}\left(G_k(t_k)\right)\right]' = 1 \tag{5.63}$$

so that

$$h_k'(x_k) = \left[G_k^{-1}\left(G_k(t_k)\right)\right]' = \frac{1}{\left(G_k(t_k)\right)'}. \tag{5.64}$$

The equation in (5.64) is negative for all $x \in S$. Taking the second derivative from (5.64), we obtain

$$h_k''(x_k) = \left(G_k(t_k)\right)'\left[G_k^{-1}\left(G_k(t_k)\right)\right]'' = -\frac{\left(G_k(t_k)\right)''}{\left[\left(G_k(t_k)\right)'\right]^2} \tag{5.65}$$

and

$$h_k''(x_k) = \left[G_k^{-1}\left(G_k(t_k)\right)\right]'' = -\frac{\left(G_k(t_k)\right)''}{\left[\left(G_k(t_k)\right)'\right]^3}. \tag{5.66}$$

The expression in (5.66) is non-negative for all $x \in S$. This completes our proof since the two expressions are proved to be true for all $x \in S$.

Theorem 1 Suppose that $b_1 \geq b_2 \geq b_3 \geq \ldots \geq b_K$ and, $h_k(.)$ is twice differentiable and convex decreasing on S, then the optimal solution of the problem

$$\max_{0\le x_i\le 1}\sum_{i=1}^{K}\left[-c_i h_i(x_i)+b_i(1-x_i)\prod_{j=1}^{i-1}x_j\right] \tag{5.67}$$

satisfies

$$x_2=\frac{\left[b_1-b_2+c_1h_1'(x_1)\right]x_1}{c_2h_2'(x_2)} \tag{5.68}$$

$$x_{n+1}=\frac{\left[\left(\frac{b_n-b_{n+1}}{b_{n-1}-b_n}\right)c_nh_n'(x_n)x_n-\left(\frac{b_n-b_{n+1}}{b_{n-1}-b_n}\right)c_{n-1}h_{n-1}'(x_{n-1})x_{n-1}+c_nh_n'(x_n)\right]x_n}{c_{n+1}h_{n+1}'(x_{n+1})} \quad 2\le n\le K-1$$

$$h_K'(x_K)=-\frac{b_K}{c_K}\prod_{m=1}^{K-1}x_m \tag{5.69}$$

Proof Since L is concave by Proposition (1), it suffices to show that the first order conditions are satisfied. We now define

$$\nabla L_i(x)=\frac{\partial L(x_1,x_2,.....,x_K)}{\partial x_i} \tag{5.70}$$

so that

$$\begin{aligned}\nabla L_i(x)&=\sum_{k=1}^{K}\frac{\partial g_k(x_1,x_2,.....,x_K)}{\partial x_i}\\&=-c_ih_i'(x_i)-b_i\prod_{m=1}^{i-1}x_m+\sum_{k=i+1}^{K}b_k(1-x_k)\prod_{m=1,m\neq i}^{k-1}x_m.\end{aligned} \tag{5.71}$$

For $n=1$; we obtain

$$\begin{aligned}\nabla L_1(x) &= -c_1 h_1'(x_1) - b_1 + \sum_{k=2}^{K} b_k(1-x_k)\prod_{m=1,m\neq 1}^{k-1} x_m = 0\\ \nabla L_2(x) &= -c_2 h_2'(x_2) - b_2 x_1 + \sum_{k=3}^{K} b_k(1-x_k)\prod_{m=1,m\neq 2}^{k-1} x_m = 0\end{aligned} \tag{5.72}$$

so that

$$-c_1 x_1 h_1'(x_1) - b_1 x_1 + \sum_{k=2}^{K} b_k(1-x_k)\prod_{m=1}^{k-1} x_m = 0 \tag{5.73}$$

$$-c_2 x_2 h_2'(x_2) - b_2 x_1 x_2 + \sum_{k=3}^{K} b_k(1-x_k)\prod_{m=1}^{k-1} x_m = 0 \tag{5.74}$$

subtracting equation (5.74) from equation (5.73) we obtain the following result:

$$\begin{aligned}-c_1 x_1 h_1'(x_1) + c_2 x_2 h_2'(x_2) &= b_1 x_1 - b_2 x_1 x_2 - b_2(1-x_2)x_1\\ &= (b_1 - b_2)x_1\end{aligned} \tag{5.75}$$

and the end result is

$$x_2 = \frac{[b_1 - b_2 + c_1 h_1'(x_1)]x_1}{c_2 h_2'(x_2)}. \tag{5.76}$$

The next step is to show that equation (5.68) is true for $2 \le n \le K-1$. Note that now

$$\begin{aligned}\nabla L_n(x) &= -c_n h_n'(x_n) - b_n\prod_{m=1}^{n-1} x_m + \sum_{k=n+1}^{K} b_k(1-x_k)\prod_{m=1,m\neq n}^{k-1} x_m = 0\\ \nabla L_{n+1}(x) &= -c_{n+1} h_{n+1}'(x_{n+1}) - b_{n+1}\prod_{m=1}^{n} x_m + \sum_{k=n+2}^{K} b_k(1-x_k)\prod_{m=1,m\neq n+1}^{k-1} x_m = 0\end{aligned} \tag{5.77}$$

so that

$$-c_n x_n h_n'(x_n) - b_n \prod_{m=1}^{n-1} x_m + \sum_{k=n+2}^{K} b_k (1 - x_k) \prod_{m=1}^{k-1} x_m = 0 \tag{5.78}$$

$$-c_{n+1} x_{n+1} h_{n+1}'(x_{n+1}) - b_{n+1} x_{n+1} \prod_{m=1}^{n} x_m + \sum_{k=n+2}^{K} b_k (1 - x_k) \prod_{m=1}^{k-1} x_m = 0 \tag{5.79}$$

subtracting equation (5.79) from equation (5.78) we obtain

$$\begin{aligned} -c_n x_n h_n'(x_n) + c_{n+1} x_{n+1} h_{n+1}'(x_{n+1}) &= (b_n - b_{n+1} x_{n+1}) \prod_{m=1}^{n} x_m - b_{n+1}(1 - x_{n+1}) \prod_{m=1}^{n} x_m \\ &= (b_n - b_{n+1}) \prod_{m=1}^{n} x_m \end{aligned} \tag{5.80}$$

and the result is

$$\prod_{m=1}^{n} x_m = \frac{1}{(b_n - b_{n+1})} \left[-c_n x_n h_n'(x_n) + c_{n+1} x_{n+1} h_{n+1}'(x_{n+1}) \right]. \tag{5.81}$$

Similarly,

$$\prod_{m=1}^{n-1} x_m = \frac{1}{(b_{n-1} - b_n)} \left[-c_{n-1} x_{n-1} h_{n-1}'(x_{n-1}) + c_n x_n h_n'(x_n) \right] \tag{5.82}$$

and

$$x_n = \frac{\prod_{m=1}^{n} x_m}{\prod_{m=1}^{n-1} x_m} = \frac{(b_{n-1} - b_n) \left[-c_n x_n h_n'(x_n) + c_{n+1} x_{n+1} h_{n+1}'(x_{n+1}) \right]}{(b_n - b_{n+1}) \left[c_n x_n h_n'(x_n) - c_{n-1} x_{n-1} h_{n-1}'(x_{n-1}) \right]}. \tag{5.83}$$

As a result of (5.83),

$$x_{n+1}=\frac{\left[\frac{(b_n-b_{n+1})}{(b_{n-1}-b_n)}c_n x_n h_n'(x_n)-\frac{(b_n-b_{n+1})}{(b_{n-1}-b_n)}c_{n-1}x_{n-1}h_{n-1}'(x_{n-1})+c_n h_n'(x_n)\right]x_n}{c_{n+1}h_{n+1}'(x_{n+1})}. \quad (5.84)$$

For the case $n=K$,

$$\nabla L_K(x)=-c_K h_K'(x_K)-b_K\prod_{m=1}^{K-1}x_m=0 \quad (5.85)$$

the following result is obtained

$$h_K'(x_K)=-\frac{b_K}{c_K}\prod_{m=1}^{K-1}x_m. \quad (5.86)$$

5.5. A Heuristic Algorithm

In this section, we will suggest a heuristic algorithm to determine a near optimal testing strategy. The algorithm is based on the fact that the formulas given in (5.68) are recursive so that if the generating functions of the failure processes are known, then the values for x_2, x_3,....., x_K can be determined by giving an arbitrary initial value to x_1.

We recall the characteristics of the generating functions as

1. $Lim_{t\to\infty}G_i(t)=0$,
2. $G_i(0)=1$,
3. $G_i(t)$ is convex decreasing on $[0,1]$,

and they are true whenever $q\in(0,1)$ for all $i=1,2,.....K$.

The steps of the heuristic algorithm are as follows:

1. Give an arbitrary initial value to x_1 such that $0 < x_1(1) \leq 1$.

2. Determine the values of $x_2(k), x_3(k), \ldots\ldots, x_K(k)$ using (5.68) where $x_i(k)$ denotes the value for the *ith* operation $(i = 2,3,\ldots\ldots, K)$ at the *kth* iteration $(k = 1,2,\ldots\ldots)$.

3. Obtain another value $X_K^2(k)$ for the *Kth* operation from (5.69).

4. Determine the error term $\varepsilon_K(k)$ such that $\varepsilon_K(k) = X_K^2(k) - x_K^2(k)$.

5. If $\varepsilon_K(k) = 0$

   a. if $0 < x_i(k) \leq 1$ for all $i = 1,2,\ldots\ldots K$, the solution is optimal, stop the algorithm.

   b. if $x_i(k) < 0$ or $x_i(k) > 1$ for one or more of the operations, set the first such $x_i(k) = 1$, eliminate the operation, and go to step 1.

6. If $sign[\varepsilon_K(k-1)] = sign[\varepsilon_K(k)] = positive$

   a. if $x_i(k) > 1$ for one or more of the operations, set the first such $x_i(k) = 1$, remove the operation, and go to step 1,

   b. else set $x_1(k+1) = x_1(k) + \Delta h$, $0 < \Delta h < 1$, go to step 2.

7. If $sign[\varepsilon_K(k-1)] = sign[\varepsilon_K(k)] = negative$, set $x_1(k+1) = x_1(k) + \Delta h$, $-1 < \Delta h < 0$, go to step 2.

8. If $sign[\varepsilon_K(k-1)] \neq sign[\varepsilon_K(k)]$

   a. if $x_i(k-1) > 1$ and $x_i(k) > 1$ for one or more of the operations, set the first such $x_i(k) = 1$, disregard the operation, and go to step 1,

   b. else set $u = x_1(k-1)$ and $l = x_1(k)$

   c. $x_1(k+1) = \dfrac{u+l}{2}$, determine $x_2(k+1), \ldots\ldots, x_K(k+1)$ using (5.68) and $X_K^2(k+1)$ using (5.69). Determine $\varepsilon_K(k+1)$.

9. If $sign[\varepsilon_K(k+1)] = sign[\varepsilon_K(k-1)]$, then go to step 8 and use the data of the $(k+1)th$ iteration instead of $(k-1)th$ iteration, else (i.e. $sign[\varepsilon_K(k+1)] = sign[\varepsilon_K(k)]$) go to step 8 and use the data of the $(k+1)th$ iteration instead of the *kth* iteration. Repeat the steps 8 and 9 until the condition stated in step 5 is obtained.

Although we have seen on several numerical illustrations that our heuristic algorithm converges to the solution in a finite number of iterations, we can not provide a formal proof of it in this study.

For an illustration of the algorithm, it is assumed that the generating functions of the failure processes are of the following form:

$$G_i(t_i) = \frac{m_i}{m_i + n_i \lambda_i p_i t_i}. \tag{5.87}$$

This type of functions satisfies the three requirements stated above. Thus,

$$h_i(x_i) = G_i^{-1}(x_i) = \frac{m_i(1-x_i)}{n_i \lambda_i p_i x_i} \tag{5.88}$$

and

$$h_i'(x_i) = -\frac{m_i}{n_i \lambda_i p_i x_i^2}. \tag{5.89}$$

Setting all m_i and n_i values to one for simplicity, the equations in (5.68) and in (5.69) turn out to be (using (5.89))

$$x_2 = \frac{-\dfrac{c_2}{\lambda_2 p_2}}{(b_1 - b_2)x_1 - \dfrac{c_1}{\lambda_1 p_1 x_1}} \tag{5.90}$$

$$x_{n+1} = \frac{-\dfrac{c_{n+1}}{\lambda_{n+1} p_{n+1}}}{\left[\left(\dfrac{b_n - b_{n+1}}{b_{n-1} - b_n}\right)\dfrac{-c_n}{\lambda_n p_n} + \left(\dfrac{b_n - b_{n+1}}{b_{n-1} - b_n}\right)\dfrac{c_{n-1}}{\lambda_{n-1} p_{n-1}}\dfrac{1}{x_{n-1}}x_n - \dfrac{c_n}{\lambda_n p_n}\dfrac{1}{x_n}\right]} \qquad 2 \le n \le K-1$$

$$x_K = \sqrt{\frac{c_K}{b_K \lambda_K p_K \prod_{m=1}^{K-1} x_m}} \tag{5.91}$$

and our objective function turns out to be (using (5.88))

$$\max_{0 \le x_i \le 1} \sum_{i=1}^{K} \left[-c_i \frac{(1-x_i)}{\lambda_i p_i x_i} + b_i (1-x_i) \prod_{j=1}^{i-1} x_j \right]. \tag{5.92}$$

We will give the numerical illustrations of the heuristic algorithm in Section 6.

5.6. The Exponential Model

In this section, a special case of the general model presented in Section (5.3) will be examined and an explicit optimal solution will be obtained. The objective function can be restated as

$$\max_{0 \le t_i \le \infty} \sum_{i=1}^{K} \left\{ -c_i t_i + \mu f_i \left[1 - G_i(t_i) \right] \prod_{j=1}^{i-1} G_j(t_j) \right\}. \tag{5.93}$$

Assumption 2 The failure generating process $\{N_t^i; t \ge 0\}$ during operation i is a Poisson process with rate $\lambda_i p_i$. In other words, $G_i(t_i) = e^{-\lambda_i p_i t_i}$.

Setting

$$x_i = e^{-\lambda_i p_i t_i} \tag{5.94}$$

and taking the log of both sides of the equation (5.95), we obtain

$$t_i = -\frac{1}{\lambda_i p_i}\ln(x_i). \tag{5.95}$$

Combining with the equations (5.94) and (5.95), the objective function turns out to be

$$\max_{0\le x_i\le 1}\sum_{i=1}^{K}\left[\frac{c_i}{\lambda_i p_i}\ln(x_i)+\mu f_i(1-x_i)\prod_{j=1}^{i-1}x_j\right]. \tag{5.96}$$

Define

$$\begin{aligned} a_i &= \frac{c_i}{\lambda_i p_i} \\ b_i &= \mu f_i \end{aligned} \tag{5.97}$$

so that our problem in (5.96) can be rewritten as

$$\max_{0\le x_i\le 1}\sum_{i=1}^{K}\left[a_i\ln(x_i)+b_i(1-x_i)\prod_{j=1}^{i-1}x_j\right]. \tag{5.98}$$

Proposition 3 Suppose that $b_1 \ge b_2 \ge \ldots\ldots \ge b_K$, then the objective function in (5.98) is concave on $[0,1]$.

Proof The objective function given in (5.98) is the same as one in [25], and thus the proof is also the same as the proof in [25]. But we should recall that the definitions of parameters a_i and b_i are different then those in [25].

Theorem 2 Suppose that $b_1 \ge b_2 \ge \ldots\ldots \ge b_K$, then the optimal solution of the problem

$$\max_{0\le x_i\le 1}\sum_{i=1}^{K}\left[a_i\ln(x_i)+b_i(1-x_i)\prod_{j=1}^{i-1}x_j\right] \tag{5.99}$$

is

$$x_n^* = \begin{cases} \dfrac{a_1 - a_2}{b_1 - b_2}, & n = 1 \\ \dfrac{b_{n-1} - b_n}{a_{n-1} - a_n} \dfrac{a_n - a_{n+1}}{b_n - b_{n+1}}, & 2 \le n \le K-1 \\ \dfrac{b_{K-1} - b_K}{a_{K-1} - a_K} \dfrac{a_K}{b_K}, & n = K \end{cases} \tag{5.100}$$

whenever $K \ge 2$ and all ratios given above are strictly between 0 and 1.

Proof The proof is same as one given in [25].

5.7. Interpretations

We already stated that the objective function obtained in the exponential model is the same as one in [25], and thus we obtained the same explicit solution. However, the definitions of the parameters are different such that $a_i = c_i / \lambda_i p_i$ and $b_i = \mu\left(e - d_i / p_i\right)$ in our study, and $a_i = c_i / \lambda_i$ and $b_i = \mu\left(e - d_i\right)$ in [25] where a_i stands for the testing cost until a fault is removed, and b_i for the total benefit of removing μ faults during testing the *ith* operation. The difference is caused by imperfect debugging since in this case the expected cost of debugging until a fault is removed is d_i / p_i instead of d_i, and the expected time until a fault is removed is $1/\lambda_i p_i$ instead of $1/\lambda_i$.

The following interpretations are true for the exponential model discussed in the Section (5.6). The condition imposed by Theorem (2) (i.e. $b_1 \ge b_2 \ge ..... \ge b_K$) implies that $b_1 > b_2 > ..... > b_K$ and also $a_1 > a_2 > ..... > a_K$ since $0 < x_n^* < 1$. If $b_i > b_{i+1}$ with $a_i < a_{i+1}$, then $x_{i+1} = 1$ as $t_{i+1} = 0$ by intuition. In such a case, the formulas can not be applied, we disregard the operation $(i+1)$, and begin the procedure again with one less operation in the profile. If $b_i = b_{i+1}$, no testing is done in operation i if $a_i > a_{i+1}$, and testing is done in

operation i if $a_i < a_{i+1}$. If $a_i = a_{i+1}$ at the same time, then $x_i = x_{i+1}$. In this case, there are multiple optima if $\lambda_i p_i = \lambda_{i+1} p_{i+1}$. That is, any linear combination of the total testing time during operation i and operation $(i+1)$ can be done in operation i and in operation $(i+1)$ without a change in the objective value (i.e. testing time = $\lambda\left(t_i^* + t_{i+1}^*\right)$ in operation i and testing time = $(1-\lambda)\left(t_i^* + t_{i+1}^*\right)$ in operation $(i+1)$ where $\lambda \in [0,1]$.)

Note also that once $\{x_i^*\}$ are calculated using equations in (5.100), the optimal testing times $\{t_i^*\}$ can be obtained from

$$t_i^* = -\frac{1}{\lambda_i p_i} \ln\left(x_i^*\right).$$

6. ILLUSTRATIONS

In this section, some illustrations of the proposed exponential model and the heuristic algorithm will be presented. These results will be compared to the results of a package that solves nonlinear programming models. The illustrations for the proposed exponential model can be classified into two categories: illustrations that satisfy the assumptions of the model, and those where the parameters do not satisfy our assumptions. The first example given above for the exponential model falls into the first group. The second example for the exponential model is the illustration where our assumptions are not satisfied. The results of the package will be given followed by the results derived from the equations of the model and the iterations of the heuristic algorithm. The software used for determining the solution is LINGO.

We consider a model with three operations. The model is

$$\max_{0\le x_i\le 1} a_1\ln(x_1)+a_2\ln(x_2)+a_3\ln(x_3)+b_1(1-x_1)+b_2(1-x_2)x_1+b_3(1-x_3)x_1x_2 \quad (6.1)$$

with the solution

$$x_1^* = \frac{a_1-a_2}{b_1-b_2}, x_2^* = \frac{b_1-b_2}{a_1-a_2}\frac{a_2-a_3}{b_2-b_3}, x_3^* = \frac{b_2-b_3}{a_2-a_3}\frac{a_3}{b_3} \quad (6.2)$$

provided that the $\{x_i^*\}$ values satisfy our assumptions.

We take a_1=3, a_2=2, a_3=1, b_1=15, b_2=12, b_3=8 so that the formulation becomes

$$\max_{0\le x_i\le 1} 3\ln(x_1)+2\ln(x_2)+1\ln(x_3)+15(1-x_1)+12(1-x_2)x_1+8(1-x_3)x_1x_2. \quad (6.3)$$

The solution obtained from the equations in (6.2) is

$$x_1^* = \frac{3-2}{15-12}, x_2^* = \frac{15-12}{3-2}\frac{2-1}{12-8}, x_3^* = \frac{12-8}{2-1}\frac{1}{8} \tag{6.4}$$

so that

$$x_1^* = \frac{1}{3}, x_2^* = \frac{3}{4}, x_3^* = \frac{4}{8}. \tag{6.5}$$

The output of LINGO is given by

$$\begin{aligned} L^* &= 7.435652 \\ x_1^* &= 0.3333332 \\ x_2^* &= 0.7499958 \\ x_3^* &= 0.5000063 \end{aligned}$$

and the results are exactly the same.

An example that does not satisfy our assumptions is obtained by when $a_1=3$, $a_2=2$, $a_3=1$, $b_1=18$, $b_2=15$, $b_3=6$, where the formulation is

$$\max_{0\le x_i\le 1} 3\ln(x_1) + 2\ln(x_2) + 1\ln(x_3) + 18(1-x_1) + 15(1-x_2)x_1 + 6(1-x_3)x_1x_2 \tag{6.6}$$

and the solution given by the formulas is

$$x_1^* = \frac{3-2}{18-15}, x_2^* = \frac{18-15}{3-2}\frac{2-1}{15-6}, x_3^* = \frac{15-6}{2-1}\frac{1}{6} \tag{6.7}$$

so that

$$x_1^* = \frac{1}{3}, x_2^* = \frac{3}{9}, x_3^* = \frac{9}{6}. \tag{6.8}$$

This solution is infeasible since the condition stated in Theorem (2) is violated. The optimal solution obtained from LINGO is

$$L^* = 9.871582$$
$$x_1^* = 0.3333539$$
$$x_2^* = 0.4000418$$
$$x_3^* = 1.0000000.$$

As can be seen above, the two solutions are different and operation 3 must be discharged. We again begin the procedure with a model of two operations, where the optimal solution is given by

$$x_1^* = \frac{a_1 - a_2}{b_1 - b_2}, x_2^* = \frac{b_1 - b_2}{a_1 - a_2}\frac{a_2}{b_2} \tag{6.9}$$

so that

$$x_1^* = \frac{3-2}{18-15}, x_2^* = \frac{18-15}{3-2}\frac{2}{15} \tag{6.10}$$

$$x_1^* = \frac{1}{3}, x_2^* = \frac{6}{15} \tag{6.11}$$

which is the same as the LINGO solution given above.

For the numerical illustration of the heuristic algorithm, we use the following parameter values.

Table 6.1. Parameters for the Illustration

Operations	b_i	c_i	λ_i	p_i
1	10	7	5	0.8
2	9	6	4	0.5
3	8	5	4	0.8

The iterations of the heuristic algorithm are as follows:

Table 6.2. Iteration 1

Operations	$x_i(1)$	$X_4^2(1)$	$\varepsilon_4(1)$
1	0.1000	11.3281	**11.3200**
2	0.1724		
3	0.0899		

Table 6.3. Iteration 2

Operations	$x_i(2)$	$X_4^2(2)$	$\varepsilon_4(2)$
1	0.2000	2.7832	**2.7943**
2	0.3509		
3	0.1843		

Table 6.4. Iteration 3

Operations	$x_i(3)$	$X_4^2(3)$	$\varepsilon_4(3)$
1	0.3000	1.2008	**1.1162**
2	0.5422		
3	0.2909		

Table 6.5. Iteration 4

Operations	$x_i(4)$	$X_4^2(4)$	$\varepsilon_4(4)$
1	0.4000	0.6470	**0.4660**
2	0.7547		
3	0.4254		

Table 6.6. Iteration 5

Operations	$x_i(5)$	$X_4^2(5)$	$\varepsilon_4(5)$
1	0.5000	0.3906	**0.0000**
2	1.0000		
3	0.6250		

To compare the above solution, the same problem is resolved using LINGO and the following solution is obtained:

$$x_1 = 0.5000002$$
$$x_2 = 1.0000000$$
$$x_3 = 0.6249999$$
$$obj. = 38.125.$$

As can be seen above, the two solutions are exactly the same.

For another illustration of the heuristic algorithm, we use the following parameter values.

Table 6.7. Parameters for the Illustration

Operations	b_i	c_i	λ_i	p_i
1	100	70	5	0.8
2	90	60	4	0.5
3	80	50	4	0.8
4	70	40	3	0.6

The iterations of the heuristic algorithm are as follows:

Table 6.8. Iteration 1

Operations	$x_i(1)$	$X_4^2(1)$	$\varepsilon_4(1)$
1	0.1000	204.8406	**204.8243**
2	0.1724		
3	0.0899		
4	0.1279		

Table 6.9. Iteration 2

Operations	$x_i(2)$	$X_4^2(2)$	$\varepsilon_4(2)$
1	0.2000	24.5511	**24.4822**
2	0.3509		
3	0.1843		
4	0.2625		

Table 6.10. Iteration 3

Operations	$x_i(3)$	$X_4^2(3)$	$\varepsilon_4(3)$
1	0.3000	6.7088	**6.5345**
2	0.5422		
3	0.2909		
4	0.4174		

Table 6.11. Iteration 4

Operations	$x_i(4)$	$X_4^2(4)$	$\varepsilon_4(4)$
1	0.4000	2.4721	**2.0790**
2	0.7547		
3	0.4254		
4	0.6269		

Table 6.12. Iteration 5

Operations	$x_i(5)$	$X_4^2(5)$	$\varepsilon_4(5)$
1	0.5000	1.0159	**-0.0161**
2	1.0000		
3	0.6250		
4	1.0159		

Table 6.13. Iteration 6

Operations	$x_i(6)$	$X_4^2(6)$	$\varepsilon_4(6)$
1	0.4500	1.5766	**0.9665**
2	0.8724		
3	0.5129		
4	0.7811		

Table 6.14. Iteration 7

Operations	$x_i(7)$	$X_4^2(7)$	$\varepsilon_4(7)$
1	0.4750	1.2652	**0.4841**
2	0.9348		
3	0.5651		
4	0.8838		

Table 6.15. Iteration 8

Operations	$x_i(8)$	$X_4^2(8)$	$\varepsilon_4(8)$
1	0.4875	1.1338	**0.2402**
2	0.9670		
3	0.5939		
4	0.9453		

Table 6.16. Iteration 9

Operations	$x_i(9)$	$X_4^2(9)$	$\varepsilon_4(9)$
1	0.4938	1.0733	**0.1142**
2	0.9834		
3	0.6092		
4	0.9793		

Table 6.17. Iteration 10

Operations	$x_i(10)$	$X_4^2(10)$	$\varepsilon_4(10)$
1	0.4969	1.0442	**0.0497**
2	0.9917		
3	0.6170		
4	0.9972		

Table 6.18. Iteration 11

Operations	$x_i(11)$	$X_4^2(11)$	$\varepsilon_4(11)$
1	0.4984	1.0299	**0.0170**
2	0.9958		
3	0.6210		
4	1.0065		

Table 6.19. Iteration 12

Operations	$x_i(12)$	$X_3^2(12)$	$\varepsilon_3(12)$
1	0.1000	11.3281	**11.3200**
2	0.1724		
3	0.0899		

Table 6.20. Iteration 13

Operations	$x_i(13)$	$X_3^2(13)$	$\varepsilon_3(13)$
1	0.2000	2.7832	**2.7493**
2	0.3509		
3	0.1843		

Table 6.21. Iteration 14

Operations	$x_i(14)$	$X_3^2(14)$	$\varepsilon_3(14)$
1	0.3000	1.2008	**1.1162**
2	0.5422		
3	0.2909		

Table 6.22. Iteration 15

Operations	$x_i(15)$	$X_3^2(15)$	$\varepsilon_3(15)$
1	0.4000	0.6470	**0.4660**
2	0.7547		
3	0.4254		

Table 6.23. Iteration 16

Operations	$x_i(16)$	$X_3^2(16)$	$\varepsilon_3(16)$
1	0.5000	0.3906	**0.0000**
2	1.0000		
3	0.6250		

The same problem is resolved using LINGO and the following solution is obtained:

$$x_1 = 0.5$$
$$x_2 = 1$$
$$x_3 = 0.625$$
$$x_4 = 1$$
$$obj. = 38.125.$$

As can be seen, we again obtained the same solution.

7. CONCLUSION

The determination of the optimal release time of software is an important problem in the software industry. In most cases, the optimal release time is determined by minimizing a cost function which is usually composed of testing cost, cost of a failure during testing phase, and cost of a failure during operational phase. Thus, there must be a trade-off between the costs incurred during testing and the costs incurred during operation to determine the optimal release time.

The operational profile describes how a user employs system. Therefore, the consideration of the operational profile in testing phase makes the procedure faster and more efficient, which means faster reliability growth. The problem is to determine the duration of testing in each operation.

In this thesis, we constructed and analyzed a new imperfect debugging model where the time-to-failure distribution is not necessarily Poisson, and determined an implicit optimal solution under reasonable assumptions. We suggested a heuristic algorithm to determine the optimal solution. Under the Poisson process assumption, the implicit solution of the general model turned into an explicit solution. The decision problem involves to determine the optimal testing strategy so that the nonlinear total cost function is minimized.

We have seen on the illustrations that our heuristic algorithm reached to the optimal solution, if there is one, in a finite number of iterations. Thus, for future work, it will be interesting to prove the convergence of the algorithm to the optimal solution in a finite number of iterations.

REFERENCES

1. Musa, J.D., "Operational Profiles in Software Reliability Engineering," *IEEE Software*, pp. 14-32, 1993.

2. Musa, J.D., "The Operational Profile," in Özekici, S. (Ed.), *Reliability and Maintenance of Complex Systems*, pp. 332-343, NATO ASI Series, Berlin, 1996.

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

4. Cavano, J.P. and J.A. McCall, "A Framework for the Measurement of Software Quality," *Proceedings of the ACM Software Quality Assurance Workshop*, pp. 133-139, 1978.

5. Jelinski, Z. and P. Moranda, "Software Reliability Research," in Freiberg, W. (Ed.), *Statistical Computer Performance Evaluation*, pp. 465-484, Academic Press, New York, 1972.

6. Littlewood, B., and J.L. Verall, "A Bayesian Reliability Growth Model for Computer Software," *Applied Statistics*, Vol. 22, pp. 332-346, 1973.

7. Goel, A.L. and K. Okumoto, "An Analysis of Recurrent Software Failures in a Real-Time Control Systems," *Proceedings of ACM Annu. Tech. Conf.,* pp. 496-500, 1978.

8. Schick, G.J. and R.W. Wolverton, "Assessment of Software Reliabilty," *Proc. Oper. Res.*, pp. 395-422, 1973.

9. Goel, A.L. and K. Okumoto, "A Time-Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Trans. On Reliability,* Vol. R-28, pp. 206-211, 1979.

10. Goel, A.L., "Software Reliability Modeling and its Estimation," Rome Air Development Center Technical Report, RADC-TR-82-263, 1982.

11. Musa, J.D., "A Theory of Software Reliability and its Applications," *IEEE Trans.on Software Engineering*, Vol. SE-1, pp. 312-327, 1971.

12. Musa, J.D. and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *Proc. 7th Int. Conf. Soft. Eng.,* pp. 230-237, Orlando, 1983.

13. Mills, H.D., "On the Statistical Validation of Computer Programs," IBM Federal Syst. Div., Gaithersburg, MD, 72-6015, 1972.

14. Lipow, M., "Estimation of Software Package Residual Errors," TRW, Relando Beach, CA, Software Series Report, TRW-SS-72-09, 1972.

15. Basin, S.L., "Estimation of Software Error Rate via Capture-Recapture Sampling: A Critical Review," *Science Applications Inc.,* Palo Alto, CA, 1973.

16. Forman, E.H. and N.D. Singpurwalla, "An Empirical Stopping Rule for Debugging and Testing Computer Software," *J. Amer. Stat. Ass.*, Vol. 72, pp. 750-757, 1977.

17. Forman, E.H. and N.D. Singpurwalla, "Optimal Time Intervals for Testing Hypotheses on Computer Software," *IEEE Trans. On Reliabilty*, Vol. R-28, pp. 250-253, 1979.

18. Koch, H.S. and P. Kubat, "Optimal Release Time of Computer Software," *IEEE Trans. On SE.*, Vol. SE-9, pp. 323-327, 1983.

19. Yamada, S. and S. Osaki, "Cost-Reliability Optimal Release Policies for Software Systems," *IEEE Trans. On Rel.*, Vol. R-34, pp. 422-424, 1985.

20. Ross, S.M., "Software Reliability: The Stopping Rule Problem," *IEEE Trans. On SE*, Vol. SE-11, pp. 1472-1475, 1985.

21. Singpurwalla, N.D., "Determining an Optimal Time Interval for Testing and Debugging Software," *IEEE Transactions on Software Engineering*, Vol. 17, Iss. 4, pp. 313-319, 1991.

22. Wohlin, C. and P. Runeson, "Certification of Software Components," *IEEE Transactions on Software Engineering*, Vol. 20, Iss. 6, pp. 494-499, 1994.

23. Hou, R.H., S.Y. Kuo and Y.P. Chang, "Optimal Release Times for Software Systems with Scheduled Delivery Time-Based on the Hgdm," *IEEE Transactions on Computers*, Vol. 46, Iss. 2, pp. 216-221, 1997.

24. Özekici S. and N. Çatkan, "A Dynamic Software Release Model," *Computational Economics*, Vol. 6, pp.77-97, 1993.

25. Özçelikyürek, S., "Testing Software with an Operational Profile," MS. Thesis, Boğaziçi University, 1997.

26. Goel, A.L., "Software Reliability Models: Assumptions, Limitations and Applicability," *IEEE Trans. on SE.*, Vol. SE-11, pp. 1411-1422, 1985.