SOFTWARE TESTING
AND RELIABILITY GROWTH MODELS

by

Deming Zhuang

Submitted in partial fulfillment of the requirements
for the degree of Master of Science

at

Dalhousie University
Halifax, Nova Scotia
August, 1998

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

0-612-36545-X

Canada

To Shumin and Jonathan

Contents

Abstract

In this thesis, we survey fundamental concepts of software testing. We review general testing process and the key testing methods.

We discuss the theory of software reliability, which is an important quantitative measure of software quality. We present the derivation of several well-known software reliability growth models and describe how they can be used in real-life software development.

Parameters in the software reliability models are usually estimated based on failure time data. We review several commonly used methods of parameter estimation. Maximum likelihood methods and curve fitting using least error methods are presented for two fundamental models: the exponential model and the logarithmic model. The descending mean algorithm is introduced. We apply a global minimization method based on the descending mean algorithm to parameter estimation problems. Numerical experiments are performed on a set of well-known failure time data. The computation results show that the method is efficient and accurate.

Acknowledgements

I would like to express my sincere thanks to my supervisor Jacob Slonim, for his invaluable advice, guidance and help in every step of my thesis writing. His broad knowledge in software engineering and rich experience in the software industry has made this thesis writing a great learning experience for me.

I would like to thank Carl Hartzman for his constant encouragement and advice during the course of my study. His careful reading of the entire thesis and his many criticisms and comments have improved the quality of the thesis.

I would also like to thank Michael Shepherd for his encouragement, assistance and advice during my study in the computer science graduate program.

I am indebted to Richard Nowakowski who took time from his busy schedule of being a chair of Mathematics department of Dalhousie University to be my external examiner.

My special thanks go to John Reid, my good friend and colleague at the Mount. He has always been interested in discussing mathematics, and computer science with me ever since I started to work at the Mount. He read most of this thesis and provides many helpful suggestions and insightful critiques.

I am very grateful to Pat Keast for encouraging me to study computer science and for several interesting computer science courses I have taken from him. I have learned a lot from his teaching. I also benefit a great deal from the computer science courses taught by Peter Hitchcock, Arthur Sedgwick, Peter Bodoric, and Andrew Rau-Chaplin.

I would like to thank Jason Denton at Colorado State University for many valuable communications on software reliability growth models and parameter

estimation; for making available to me the integrated software reliability tool ROBUST and for providing me the information on the failure time data sets.

I would like to thank Colin Mackenzie for helping me to learn the software testing tool ATAC and many assistances during the course of thesis writing. I am very grateful to Yinghong Jillian Ye and her family for the hospitality and the assistance I received when I visited IBM Toronto Labs. Jillian's work in software testing has been a great model for me to follow.

I am indebted to Saul London and his group in Bellcore for assisting me to instrument lynx and Mozilla (Netscape V5.0) with ATAC.

The financial support from NSERC and Mount Saint Vincent University is gratefully acknowledged. My colleagues at the Mount have made my stay at the Mount an enjoyable experience. My thanks to you all.

Finally, I want to thank Shumin and Jonathan for their love and support without which I would not have finished the thesis.

Chapter 1

Introduction

The applications of computer systems are in every fabric of our modern society. Computer software life cycle which includes developments, operations and maintenance are important parts of computer professionals activities. We should expect that the software we develop work properly every time it is run. However, this is not a realistic expectation. Faults and errors do exit in every large application or/and software system. These faults cause software systems to fail. If a software system fails frequently, we say that the software system is *unreliable.* In today's competitive market, an unreliable software product will cost a company a great deal both in services and in reputation. Therefore, software testing is an integrated and important part of the the software development life cycle in every software development.

The goal of software testing is to detect and to eliminate all software faults and defects before releasing software products to market or delivering them to customers. Current software industry practice makes extensive use of testing to ensure the reliability of its products. Studies [1, 25] report that cost associated with testing range from 40% to 50% of the entire software product development life-cycle expenses (in both capital and time); and, even then, it is often considered insufficient. On the other hand, a software development shop often under pressure to release its product to the market quickly in order to protect its market share. Any delay beyond what the competitive pressure of the market might tolerate could jeopardize the product's marketplace acceptance and in some cases

the entire investment would be lost. It is estimated that only 20 % of software developed is used. Also, the resource allocated to software testing is usually limited. Therefore, the software testing process must be efficient and cost effective.

The software testing process has now evolved into a mature discipline of software engineering. Many researchers and practitioners actively work in the area of software testing. Testing tools are now available to assist software testing professionals. In particular, code coverage tools can be used to determine whether the software has been thoroughly tested by showing which parts of the software have in fact been executed by the tests. Code coverage has been shown to be a good measure of the quality of the testing. Various reliability growth models have been developed to measure the reliability of software products quantitatively.

In this thesis, we examine some essential concepts and techniques of software testing. We present detailed descriptions of several well-known software reliability models. We study a few well known parameter estimation techniques for software reliability growth models. We develop a simple global optimization method which provides an accurate way to estimate the parameters in the software reliability models. The method is applied to a set of well known failure time data. The parameters of the exponential model and the logarithmic model for these data sets are estimated by our method.

Chapter 2

Software Testing

Software development life cycle is divided into several well known phases, described in much of the software engineering literature [2, 23, 24]. Software testing is involved with all these phases, starting from the design of system test cases according to the product requirement specifications to the execution of acceptance testing of a completed product.

It is important to understand the fundamental concepts such as what a software failure is, what the purpose of testing is, what the goals of testing are, what the cost of testing is, how testing can be divided into easily manageable parts, what tools can be used and how testing is associated with Software Quality Assurance (SQA). We first discuss some of these essential concepts.

2.1 Definitions of Software Failures and Faults

A *software failure* is the departure of the external results of program operation from the software requirements. Software failures relate to the dynamic behavior of the program. In an ANSI/IEEE standard [4], a failure, or anomaly, of software is defined as follows: *An anomaly is any condition that departs from the expected. This expectation can come from documentation (requirement specifications, design documents, user documents, standards, etc.) or/and from the user's perceptions or/and experiences.*

An anomaly is not necessarily a problem in the software product; it may be

manifesting correct behavior An anomaly may also be caused by something other than the software, such as instrument, I/O, hardware, etc. [4]

A *fault* is the defect in the program that causes a failure when the program is executed under particular conditions. A fault is a property of the program but not a property of its execution or behavior.

Software products are not just programming code. They include product requirements, functional specifications, drafts and final versions of user manuals, drafts and final versions of data sheets, technical support notices and many other things which are all part of different phases of the software development process. Software production is then a series of imperfect translation processes. Each of these translations produces a work product or deliverable. Each of these translations is imperfect because of human activities involved in the process. A fault is created when a developer makes an *error.*

In the ANSI/IEEE standard errors are divided into eight main classes: Logic problem, computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document quality problem and enhancement. Each of these classes is divided into subclasses, which define the error in question in detail. For example, document quality problem class is divided into the following five error classes: Application standards not met, not traceable, not current, inconsistent and incomplete. We will not discuss the classification in details.

2.2 The Role of Software Testing

It is important to have an appropriate understanding of the role of software testing. Some people think that testing is a process of demonstrating that errors are not present. However, such a goal for software testing is impossible to achieve both practically and theoretically.

A more suitable role for testing is: *to execute a software or a system with the intention of finding errors* [5]. This is a less pernicious definition than the previous one. It is a more practical goal to find as many errors as possible than

to prove that the program works correctly, which can be a nontrivial task even for a small size program. It motivates a tester better.

An important addition to the definition of testing is that the objective of testing is to find errors, but not the solution to the errors (i.e. to test but not to debug).

The definition of testing according to the ANSI/IEEE 1059-1993 standard is that *testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item.*

The purpose of testing is the improvement of product quality. However, when customer-designed products are made on demand, quality is entirely dependent on matching the customer's specification. If the customer does not like the end result, the product must be repaired, even if the product meets the specification that the customer had agreed upon.

Software testing consists of two processes: *verification* and *validation.* Verification, as defined by IEEE/ANSI, is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Verification is the process of evaluating, reviewing, inspecting requirement specification, design specification, and source code. It involves looking at static documentations .

Validation, on the other hand, involves the execution of software on a computer. Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Testing is directly related to the cost of software development. Frequently, fifty per cent or more of the development cost is devoted to error detection and removal [13]. Insufficient testing implies poor quality of software products. Exhaustive testing implies that many products never reach the market because testing would never end and the cost of testing become astronomical. Thus, we cannot expect that the software testing process is carried out perfectly for every

software product. The level of thoroughness for software testing depends on the software under test. For example, requirements for software reliability in the medical and nuclear industries are much higher than requirements for software reliability used in a business office. Some software is not allowed to "behave in a wrong way", and cannot contain any errors at all. Such software need more thorough and accurate tests than some other software, which implies higher cost. On the other hand, there are situations when the users of the software more easily tolerate software failures in the products. Software development organizations set their standards for the reliability of their products; their tolerance of the number of software errors existing in the products. These limits are determined by requirements from customers and from common quality standards. Experience from the software industry has shown that it is impossible to produce and release large commercial software without a single error.

Due to the time and cost constraints, it is not always possible to discover and remove faults that cause failure. Sometimes, only severe failures such as system hanging, violations of data consistency and integrity, get fixed immediately. Therefore, there are concepts of *priority* and *severity* of failures which are used in software industry [30, 29, 31, 33]. For example, in a Windows application, a failure such as start up window crashing should be assigned a very high severity because the failure prevents the testing of other functionality of the product. An isolated failure with little affect to the main functionality of the product, such as a spelling error in a documentation, may receive a low severity. The corrections for failures with very low severity may be deferred to the next release if the product is near the end of development cycle.

In some organizations, testing is still considered to be the last phase in the software development cycle. The testing starts just before the software release. This causes the testing process to be its own isolated phase and accounts for errors being found late, re-executing work, and poorly planned test cases. The cost of fixing the faults found by testing becomes extremely high.

Software testing must be understood as a continuous activity over the entire

software development cycle. Software testing should be designed before coding begins. The testers can inform the developer what kinds of tests will be run, including the special cases that will be checked. The developer can use that information while thinking about the design, during design inspections and in his own unit testing.

2.3 Software Testing Technology

There are two distinct categories of testing: **Black-box testing**, also known as *functional testing*, and **white-box testing**, also known as *structural testing*. White-box testing examines how the program works, taking into account possible pitfalls in the structure and logic. Block-box testing examines what the program accomplishes, without regard to how it works internally.

Black-box Testing:

Black box testing is to look at a system from the outside looking in. The term "black-box" simply means to view the system as if it were a sealed box whose particular internal contents are not visible. In black-box testing, knowledge concerning the application area or user view is exploited. Test data are chosen using the requirement analysis and software specification documents, and/or the operational profile, ignoring the details of what is inside the box. The focus of the black-box testing is on the end-user visible interfaces, features and behavior of the system. Black-box testing attempts to apply sets of inputs that will fully exercise all the functional requirements of a system. Black-box testing finds errors such as:

1. incorrect or missing functions,
2. interface errors,
3. errors in data structures or external database access,
4. performance errors, and
5. initialization and termination errors.

Two main strategies are often used in black box testing: those based on function specifications of a software (which is often referred to as *functional testing*), and those concerned about program input and output domain. An operational profile is often required for the latter, since it is usually infeasible to test all possible combinations of input data.

Black-box testing will not test those functions that are implemented but not described in the functional design specification.

White-box Testing:

White-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. The term "white-box" means to view the system as if it were an open box whose contents are very important. White-box testing requires an intimate knowledge of the internal structure of the software. For this reason, it is also called *Structural Testing.* The test designs are derived from the internal design specification of the code. Thus, the tester must have the appropriate knowledge and skills to interpret design specifications and code.

White-box tests are designed to exploit weaknesses in a product's design and implementation. Test cases are designed to:

1. guarantee that all independent paths within a module have been exercised at least once,

2. exercise all logical decisions on their true and false sides,

3. execute all loops at their boundaries and within their operational bounds, and

4. exercise internal data structures to ensure their validity.

Randomly selected test cases may not execute certain modules such as exception handling or error recovery routines. However, these are critical components of a system that must be highly reliable. White-box testing can help testers to design test cases which force the coverage of such critical components.

The majority of the white-box testing techniques require the use of test data to force execution of selected paths in order to satisfy specific criteria. These techniques are often called *path-testing* techniques. There are two main classes of path testing techniques: *control-flow* driven and *date-flow* driven testing techniques.

Control-flow driven testing involves constructing test data to execute the control flow features of a program. Statement and the branch (decision) testing are examples of "Control-flow" driven testing strategies.

From data-flow point of view, during the execution of the system, there are only three possible actions that can be applied to a variable:

1. definition: when a value is given to a variable through assignment or input.
2. reference: when the value of a variable is a recalled from memory, and used in an expression.
   - c-use: a reference to a variable which is a computation use; that is, occurs in an expression part of computation statement.
   - p-use: a reference to a variable which is a predicate use, occurred in the conditional portion of a branching statement.
3. undefined: when a variable becomes unavailable as, for example, local variables become unavailable on exit from of block or function. All variables are assumed to be in this state initially.

Data-flow driven testing is to construct test data which forces the execution of different interactions between variable definitions and their reference in a program.

2.4 Testing in Software Development Process

The software development process used in an organization is based on the spiral model or the waterfall model of software development. Software testing is involved in the following stages in the process:

- The requirement analysis and specification phase;
- The design phase;
- The coding and unit-testing phase;
- The function verification and integration test phase;
- The system verification and acceptance testing phase;
- The system maintenance.

The detailed description of these stages can be found in [33]. The unit test phase is performed by the developers of the code. Unit tests consist mainly of "white-box" testing, often with the help of a debugger, with the objective of exercising all possible code paths. During the unit test (also called private code) phase , the tester is not required to have a formal test plan or to use standard test cases and concrete exit criteria. There are some guidelines prescribing how the unit test should be carried out. It is incumbent upon the developers to adhere these guidelines and to perform the unit tests to the best of their abilities. Once the developers have completed the unit test phase of their code, they integrate the code in the main code (also called public code) base. A verified executable code base is often referred to as a "build".

The function verification test phase is also completed by developers. However, the function verification test phase usually has formal test plans. Several test cases are written for this phase of testing. Exit criteria are based on the successful execution of all planned test cases and the absence of open defects. The test cases are then integrated into an automated suite (bucket), which is then handed off to the members of a testing team who run this continuous regression bucket for the remainder of the development cycle until the product is shipped.

A system verification test is conducted by an independent department. Testing is broken down into smaller components based on the software functionality and usage. Each component verification test determines the stability of the product form one component's perspective. When all component verification tests

are completed, formal system verification testing is performed. Test plans and scenarios are written based on random workloads tailored by operational and environmental profiles. Exit criteria are defined as the successful execution of all the scenarios detailed in the system verification test plan and the absence of open defects.

The final continuous regression testing is performed by a separate department in some organizations. Its purpose is to ensure that changes made to software, such as adding new features or modifying existing features, do not adversely affect features of the software that should not change. This phase occurs during the entire development cycle, and is executed on the latest available build from development. It consists of executing automated and manual Function Verification Test buckets from the previous and current releases of the product. New test cases are added from the current release under test once Function Verification Test has exited. The entry criterion is the start of a new release and the exit criterion is the product ship of the new release.

Regression testing can take several days to several weeks depending on the size of the software product and the test buckets. Before the release of software, the last run should have a full regression test.

It is not cost and time effective to re-run all the tests in a regression suite; a method is needed to reduce the testing effort. The method should support test set minimization and test prioritization. The method should also help testers identify a representative subset of tests which should be re-executed to validate modified software.

2.5 Code Coverage Analysis

An important technique of software testing is so called *code coverage analysis* which measures the reliability of the software by measuring the software source code coverage made by the test suites. It is often difficult to evaluate the quality of the testing for a software system until the software is released. Code coverage provides a numerical measure that managers can easily understand. Software code coverage analysis can be used throughout the entire life cycle process, from coding and unit testing to final system testing. It can be used for both functional and structural tests. Many theoretical studies and industrial practices indeed conclude that an increase software code coverage is likely to locate more faults in the code and therefore increase the reliability of the software. Slonim et al [30, 29, 31] have studied code coverage and reliability growth on large scale commercial software in an industrial setting. Several researchers have proposed models that incorporate code coverage in their reliability models. [6, 11, 17, 15]. Code coverage has also been studied in [11, 32].

Software testing coverage analysis is the process of:

1. finding areas of a program not exercised by a set of test cases;
2. creating additional test cases to increase coverage;
3. identifying redundant test cases which do not increase coverage, and
4. determining a quantitative measure of test coverage.

Coverage analysis requires access to test program source code and often requires recompiling it with a special command. Testers can choose from a range of measurement methods. Testers establish a minimum percentage of coverage, to determine when to stop analyzing coverage.

Test coverage analysis can be used as a structural testing technique: *white-box testing*. Structural testing compares test program behavior against the apparent intention of the source code. Test coverage analysis can also be applied to functional testing: *black-box testing*, which compares test program behavior against

the requirements specifications of the software system, as we have discussed in the previous section.

Some fundamental assumptions about coverage analysis are the following:

- Faults in the software system relate to control flow. Testers can expose faults by varying the control flow [26].

- Testers can look for faults without knowing what failures might occur and whether all tests are reliable, so that successful test runs imply program correctness [22]. Thus, we assume that the tester understands what a correct version of the program would do and can identify differences from the expected behavior.

Other assumptions include achievable specifications, no faults of omission, and no unreachable code.

Clearly, these assumptions do not always hold. Coverage analysis exposes some faults but does not expose all classes of faults. Some people argue that coverage analysis provides more benefit when applied to an application which makes a lot of decisions rather than data-centric applications, such as a database application.

Many different coverage measures exist in software testing practice. Some of them are described below:

Basic Block Coverage: A basic block is a sequence of instructions that, except for the last instruction, is free of branches and function calls. So, the instructions in any basic block are either executed all together, or not at all. Block coverage ensures that all basic blocks are executed at least once. In C and C++, a block may contain more than one statement if no branching occurs between statements; a statement may contain multiple blocks if branching occurs inside the statement; an expression may contain multiple blocks if branching is implied within the expression (e.g., conditional, logical-and, and logical-or expressions).

Basic block coverage is sometime referred to as *statement coverage* when the unit of code measured is each executable statement.

The main advantage of block coverage is that it can be applied directly to object code and does not require processing source code. Performance profilers commonly implement this measure.

The main disadvantage of statement coverage is that it is insensitive to some control structures. Thus, basic block coverage is affected more by computational statements than by decisions.

Decision Coverage: Decision coverage ensures that each of the branches within a conditional statement evaluate to both true and false, at least once. A conditional statement may contain a number of conditional expressions, each having a true and false decision path passing through it. Each of these decision paths corresponds to a different testable attribute to be covered.

Decision coverage reports whether boolean expressions tested in control structures (such as the if-statement and while-statement) evaluated to both true and false. The entire boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Additionally, this measure includes coverage of switch-statement cases, exception handlers, and interrupt handlers. Completely adequate decision coverage implies completely adequate block coverage, except for programs with no branches.

C-use Coverage: A *c-use path* is a path through a program from each point where the value of a variable is modified for each computation use or more concisely *c-use*. The c-use coverage measures the fraction of the total number c-uses that have been covered by one c-use path during the testing.

C-use (computational use) coverage ensures that if a variable is defined (assigned a value) and later used within a computation that is not part of a conditional expression, at least one path between this def-use pair is executed.

P-use Coverage: A *p-use path* is a path from each point where the value of a variable is modified to each use on a predicate or decision without modifications to the variable along the path. *P-use coverage* measures the total number of p-uses that have been covered by one p-use path during the testing.

P-use (predicate variable use) coverage ensures that if a variable is defined

and later used within a conditional expression, this def-use pair is executed at least once causing the surrounding decision to evaluate true, and once causing the decision to evaluate false.

The intuition behind c-use and p-use coverage is, when a variable is assigned a value and that value is later used, a good test set will exercise this relationship. For any use of a variable, this should occur for each assignment that might have given rise to the variable's value.

All-uses coverage is the sum of p-use and c-use coverage measures.

Extensive code coverage is costly. In general, 100% code coverage is not practical in later stages of testing for large scale software systems. Note also that even if 100 % test coverage is achieved for some software, there is still no guarantee that all faults in product are uncovered. Many of the faults in software products are the result of code omission. In other words, there are scenarios which were not considered by the designer or the programmer of the product. For these types of faults, code coverage alone cannot assist testers very much. Therefore, software testing should not focus exclusively on code coverage.

ATAC is a coverage analysis tool for C and C++. ATAC is a part of test tool suite developed by Bellcore. ATAC measures how thoroughly a program has been exercised by a set of tests, identifies code within the program that is not well tested, and determines the overlap among individual test cases. ATAC is used by software developers and testers to measure the adequacy of a test set and identify areas in a program that require further testing. These measures may be used for project tracking to indicate progress during testing, and as acceptance criteria to subsequent stages of development and testing. Regression testers also may use ATAC to identify a particular subset of a test set that achieves high coverage at limited cost.

ATAC provides an integrated suite of software tools that support coverage testing for a number of coverage measures such as function-entry, function-call, function- return, block, decision, c-use, p-use, and all-uses. The application of ATAC in software testing on large scale commercial software in an industrial

setting can be found in [33, 32].

2.6 Summary

In this chapter, we review some fundamental concepts and methods in software testing. We differentiate faults and failures; emphasize the role of software testing; study black-box testing and white-box testing and software testing in software development process. We also study code coverage analysis and different measurements in code coverage.

Most of the materials presented in this chapter are standard, and can be found in software testing literature, e.g. [2, 3, 13, 23, 24].

Chapter 3

Software Reliability Growth Models

The reliability of a software system is the probability of its failure-free operation for a specified duration under a specified environment. From a user's perspective, reliability of the software product refers to the correctness and completeness of that product: Does it function correctly according to the specification and is it robust enough? Therefore, reliability represents a user-oriented, rather than developer-oriented view of software quality.

Reliability and cost of a software product are two most important characteristics of the product [23]. Reliability of software is intimately related to faults that the software system contains. It plays an important role in determining what the total cost of operating the product is.

Software reliability measurements can be used to guide managerial and engineering decisions on projects involving software. They can also guide customers and users of systems in purchasing and operating them. Software reliability measurements can be used to determine what software engineering methods are most effective in enhancing reliability.

A software reliability growth model (SRGM) is a mathematical expression of the software error occurrence and the removal process. Software reliability growth models are generally formulated in terms of random processes. The models are distinguished from each other by the probability distribution of failure times or

number of failures experienced and by the nature of the variation of the random process with time.

A software reliability model specifies the general form of the dependence of the failure process on the factors that introduces faults into software. Such factors include the developed code size, software engineering technologies being applied during development and the level of experience of programmers and managers.

Many software reliability growth models have been established in literature. See, for example, [23, 20] and references therein.

In this chapter, we study fundamental concepts in software reliability theory. We examine the mathematical background of random processes and reliability theory. We present careful derivations of several important quantities related to software reliability. These derivations are based on general software reliability growth models. Finally, we focus our study on two popular models, namely the *Musa Basic Execution Model*, also known as *the exponential model*, and *the logarithmic model*. Our presentation follows closely the book by Musa et al [23].

3.1 Basic concepts in software reliability

We mentioned above that software reliability is the probability of failure free operation of a software system for a specified time in a specified execution *environment*. For example, if we say that a multi processing system has a reliability of 0.92 for 8 hours when operated by the "average user", then when the system is executed for 8 hours, it would operate without failure for 92 of these periods out of 100.

Reliability is the most important measure of "*software quality*". Reliability can be presented in many ways. Other than presenting it as the probability of the next failure occurrence, we can also present it as mean-time-to-failure (MTTF), cumulative number of failures up to a specified time, failure intensity or the number of failures remaining.

The reliability of software depends on its execution environment. To characterize the environment, we divide the execution of the program into a set of

runs. Runs that are identical repetitions of each other are said to form a *run type*. The *operation profile* is the set of relative frequencies of occurrence of the run types and the set of all input states to the software that would occur in normal operation by the user of the software.

The environment is specified by the operational profile. The operational profile plays a large role in software reliability studies to date.

Reliability measurements are determined by software testing. A suite of test cases can be randomly selected to run on a software system. These tests are of the same probability expected to occur in the normal operations of the system. The input space of the system should be well covered by the tests in order to obtain an accurate reliability measurement.

The measurements of reliability in the operational phase of the software product are established by taking failure data.

Although different variations of software reliability models have been established, the fundamental theory is the same: it is assumed that a number of faults exist in the software, and software failures occur probabilistically over time according to a certain failure intensity. Through testing and debugging, problems are found, faults are removed, and reliability then increases. Therefore, the failure intensity decreases, the cumulative number of defects remains stable, and the time between failures increases. The inverse relationship between cumulative failures and failure intensity is a basis for software reliability growth theory: through the testing and debugging process, the cumulative number of the failures increases; eventually, the failure number is expected to *level off* after most of the failures are removed. Meanwhile, testing costs (in human-hours and the CPU-hours) tend to rise in a complementary fashion as fewer defects are remain in the code.

Many reliability growth models can be derived based on these relationships, with a few additional assumptions or some refinements of assumptions, presented by different failure intensity formulas.

As we are going to see in the next few sections, a number of assumptions

are made in order to establish concise software reliability growth models. Key assumptions often included in software reliability growth models are the following:

- Times between failures are independent.
- Equal probability to the exposure of each fault.
- Embedded faults are independent of each other.
- A detected fault is immediately removed (before the start of next test case), or when it is not, it is assumed that the continuous existence of the detected faults will not block other faults from exposure.
- No new faults are introduced during corrections of previously discovered faults.
- Testing intervals are independent of the trial.

Not all of these assumptions are presented in all models. Some models make refinements to relax some of these assumptions.

The above assumptions are necessary to reduce the complexity of real world situations in order to build a theoretical model. However, it is also those assumptions that restrict the applicability of a model in practice. For example, in a real software industrial setting, faults in a software are usually not independent of each other. Once a failure occurs, it may not be immediately corrected. Failures may be assigned a priority and a severity. Only the failures with very high priority or very high severity, such as those preventing the executions of other planned testing tasks, those that make the system hang, or produce wrong data, will be corrected as soon as they occurred. Corrections of previous failures may introduce other faults to the software system.

Therefore, there are some limitations to the applicability of software reliability growth models.

However, it is our belief that when used appropriately, software reliability models, while not perfect, are of some value to product management. Like mod-

eling in other experimental sciences such as biology, chemistry, geophysics, medical science, environments of software reliability growth models are often idealized and simplified by making imperfect assumptions in order to establish concise and practical models. As long as the factors omitted by the idealization are statistically insignificant, the models can be of some value.

For example, in general, defects in a software system are not independent. However, classes of defects often are. When a group of defects is found, testers can shift their strategy to find different classes of defects. Software reliability growth models can introduce so called *lump smoothing* techniques to accommodate such testing processes [14]. Also, correction of found faults usually introduces new faults. However, Musa suggests that on average about 5% new faults are spawned during the correction process [23], p.121. (Though some people who have been working in software industry do not agree the 5 % estimation.) This is generally held to be small enough so as not to be a significant factor. Again, smoothing techniques can be applied sometimes to improve the accuracy of models' predication. The Littlewood-Verrall model [23] allows the occasional decrease in program reliability (faults added) as long as the general trend is towards a more reliable program.

The usefulness of software reliability growth models can be described as follows. In general terms, a good model enhances communication on a project and provides a common framework of understanding for the software development process. It also enhances visibility to management and other interested parties. These advantages are valuable, even if the projections made with the model in the specific case are not particularly accurate.

Reliability generally increases with the amount of testing. Thus, reliability can be closely linked with project schedules. Furthermore, the cost of testing is highly correlated with failure intensity improvement. Since two of the key process attributes that a manger must control are schedule and cost, reliability can be closely tied in with project management.

Quantitative measures provided by software reliability models offer manage-

ment the possibility of evaluating development status during the test phases of the project. A software reliability model can be used to help a manager answer the following questions relating to project status and scheduling:

- Is this software ready for release?
- When will it be ready to release?
- Should we regress to a previous version from the present one?

Many methods such as intuition of designers or test teams, percent of tests completed, and successful execution of critical functional tests, have been used to evaluate testing progress. These methods have not been very satisfactory [23]. An *objective reliability measure* established from test data provides a sound means of determining status. Code coverage analysis provides one objective measure. Appropriate software reliability models also provide such measures.

Developers and managers can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software. The reliability of the software usually decreases as a result of such change. A reliability objective can be used to determine when, and perhaps how large, a change will be allowed. The objective would be based on the user and other requirements.

A quantitative understanding of software quality and various factors influencing it and affected by it enriches the manager's insight into the software product and the software development process. The manager is then much more capable of making informed decisions.

Some of the characteristics of a good software reliability model include satisfactory predictability; conciseness and simplicity of the model; the existence of clear physical interpretation of the parameters in the model; the relevance to information about the program such as the size of the program, the competence of the programmers etc. that exists before the program has been executed.

In the following sections, we present some fundamental reliability growth models. We start by reviewing some essential material in the theory of reliability.

Then, we derive general Poisson models based on reliability theory. Specializations of the general models will lead to several well-known reliability growth models in software reliability literature. This chapter provides the foundation for the next chapter, which is on the estimation of the parameters of these models in practice.

3.2 Reliability and Hazard Functions

The basic building block of a reliability model is the *time-to-failure distribution* of a system. Let T represent a continuous time-to-failure random variable. Assume that the density function of T is $f(t)$. $F(t)$ is the probability distribution function of T. Note that T is often referred as *failure probability* and f is called *failure density*. If T is differentiable, then $f(t) = \frac{dT}{dt}$. Assume $T \geq 0$ throughout. We define the reliability function as:

$$R(t) = P(T > t) = 1 - F(t) = \int_t^\infty f(x)dx$$

A reliability function $R(t)$ is monotonically decreasing, $R(0) = 1$ and

$$\lim_{t \to \infty} R(t) = 0.$$

Let $R_1(t)$ and $R_2(t)$ be reliability functions of software systems S_1 and S_2 respectively. We say that S_1 is more reliable than S_2 if $R_1(t) \geq R_2(t)$ for all t. In practice, such requirement is unrealistic. It is often the case that reliability functions for two systems are not strictly ordered. In other words, for some t, we may have $R_1(t) > R_2(t)$ while for some $t_1 \neq t$ we may have $R_1(t_1) < R_2(t_1)$.

To overcome this problem, the *mean-time-to-failure* (MTTF) is often employed as a measure of reliability. For the time-to-failure random variable T with density $f(t)$ and the reliability function $R(t)$, the MTTF is simply the expected value of T. The MTTF can be expressed in terms of either the time-to-failure density $f(t)$ or the reliability function:

$$E(T) = \Theta = \int_0^\infty tf(t)dt = \int_0^\infty R(t)dt, \tag{3.1}$$

provided that $R(t) = o(\frac{1}{t})$ as $t \to \infty$. The software system with the greater MTTF is said to be more reliable.

The probability that a system will fail in the time interval $(t, t+\Delta t)$ under the condition that it has survived to time t is given by the conditional probability:

$$\begin{aligned} P(t < T < t + \Delta t \mid T > t) &= \frac{P(t < T < t + \Delta t, T > t)}{P(T > t)} \\ &= \frac{P(t < T < t + \Delta t)}{P(T > t)} \\ &= \frac{F(t + \Delta t) - F(t)}{R(t)}. \end{aligned}$$

The *average rate of failure* in the interval $(t, t + \Delta t)$, given the system has survived to time t, is:

$$\frac{P(t < T < t + \Delta t \mid T > t)}{\Delta t}.$$

Letting $\Delta t \to 0$, we have the **instantaneous failure rate**:

$$\begin{aligned} z(t) &= \lim_{\Delta t \to 0} \frac{P(t < T < t + \Delta t \mid T > t)}{\Delta t} \\ &= \lim_{\Delta t \to 0} \frac{F(t + \Delta t) - F(t)}{\Delta t R(t)} \\ &= \frac{F'(t)}{R(t)} \\ &= \frac{f(t)}{R(t)}. \end{aligned}$$

The function $z(t)$ plays an important role in the reliability theory and deserves a special name. For a time-to-failure random variable T possessing density $f(t)$ and reliability function $R(t)$,

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

is called the **hazard function**, and gives the instantaneous failure rate.

The following proposition shows that the reliability function can be expressed in terms of the hazard function.

Proposition 3.2.1 *If $f(t), z(t)$ and $R(t)$ are the density, hazard function, and reliability function, respectively, of a time-to-failure random variable T, then*

$$R(t) = e^{-\int_0^t h(x)dx},$$

and

$$f(t) = z(t)\, e^{-\int_0^t h(x)dx}.$$

Proof. Since $f(t) = F'(t)$ and $R(t) = 1 - F(t)$, we have

$$z(t) = \frac{f(t)}{R(t)} = -\frac{R'(t)}{R(t)}.$$

Integrating both sides from 0 to t and recall that $R(0) = 1$, we get

$$\int_0^t z(x)dx = -\int_0^t \frac{R'(x)}{R(x)}dx = -\log\left[R(x)\right]\Big|_0^t = -\log[R(t)]$$

To obtain the desired expression of $R(t)$ in terms of $z(t)$, we apply the exponential to the above equation. The identity $f(t) = z(t)R(t)$ yields the second expression stated in the proposition. □

Recall that a Markov process has the property that the future of the process depends only on the present state and is independent of its history. This property allows us to use Markov processes to model random behavior of software reliability and fault intensity, because software failure process depends mainly on the faults remaining and the current operational profile. The number of faults remaining in the future will not be known with certainty.

We use $U(t)$ to denote a random variable representing faults remaining at time t.

The Markov property for the process $U(t)$ can be roughly stated as follows: For an interval Δt, the number of faults remaining at time $t + \Delta t$ depends only on the present state at time t. Assuming that at time t, the software has i failures, then the future behavior of the $U(t)$ can be described by the conditional probability:

$$P[U(t + \Delta t) = j \mid U(t) = i]$$

This conditional probability is often referred to as the *transition function* of the process.

Let $P_{ij}(t, \Delta t)$ denote the above transition function. The probability that the number of faults remaining at time $t + \Delta t$ is equal to j, is given by:

$$P[U(t + \Delta t) = j] = \sum_i P_{ij}(t, \Delta t) P[U(t) = i]. \tag{3.2}$$

Poisson processes are special Markov processes with the following additional property: *The occurrence or nonoccurrence of an event in any interval is independent of the occurrence or nonoccurrence of events in any other interval*

The Poisson distribution is appropriate for describing random behavior in many applications. For example, the Poisson distribution is often used to predict the number of phone calls arriving at a given telephone exchange within a certain period of time; the number of automobile accidents occurring within a certain period of time; and the number of radioactive particles passing through a counter during a certain time interval.

Using these basic concepts, we can study software reliability growth models. The word *growth* here comes from the assumption that once defects are removed, the software become more reliable. The reliability of a software product *grows* as the result of removing defects from the software.

Many different software reliability growth models have been proposed and used by various researchers. Two types of models: *binomial* -type models and *Poisson*-type models have been studied extensively in literatures. The key distinction between the two types of models is that the binomial type models assume that there is a fixed number of initial faults in a software product whereas Poisson-type models assume that the initial number of faults is not known with certainty. In this thesis, we concentrate on Poisson-type software reliability models which is closer to real world of practitioners.

3.3 Properties of General-Type Models

In this section, we derive General-Type reliability models and some useful quantities that are associated with the models. These quantities allow us to answer the common questions regarding the reliability of software such as:

Q1: How many failures will occur in a certain amount of time?

Q2: How many more failures remain after testing for a certain amount of time?

Since the number of total faults is an extremely complex function of many variables such as program size, complexity, and programmers' competency, it is appropriate to consider it as a random variable. Throughout this work, we assume that *The occurrence or nonoccurrence of failures in any interval is independent of the occurrence or nonoccurrence of failures in any other interval.* Then the number of occurrences of failures is a random variable which can be appropriately described by the *Poisson distribution function*

Lwt $M(t)$ be the random variable representing the number of failures experienced by time t. Then $\mu(t) = E[M(t)]$ is the expected number of failures experienced by time t. $M(t)$ can be specified by its mean value function $\mu(t)$ or by its failure intensity function $\lambda(t)$.

The exact expression for the distribution of $M(t)$ is difficult to obtain. We approximate it using Poisson processes. To do so, we make the following assumptions about the occurrence of failures:

A1: There are no failures experienced at time $t = 0$, or $M(0) = 0$ with probability 1.

A2: The process has independent increments. In other words, the number of failures experienced during $(t, t+\Delta t]$, $M(t+\Delta t) - M(t)$, is independent of the history. This assumption implies the Markov property that the future $M(t + \Delta t)$ of the process depends only on the present state $M(t)$ and is independent of its past $M(x)$ for $x < t$.

A3: The probability that a failure will occur during $(t, t+\Delta t]$ is $\lambda(t)\Delta t + o(\Delta t)$, where $\lambda(t)$ is the failure intensity of the process and

$$\lim_{\Delta t \to 0} \frac{o(\Delta t)}{\Delta t} = 0.$$

A4: The probability that more than one failure will occur during $(t, t+\Delta t]$ is $o(\Delta t)$.

The above assumptions imply that the transition function for $M(t)$ is

$$P_{ij}(t, \Delta t) = \begin{cases} 1 - \lambda(t)\Delta t + o(\Delta t) & \text{if } j = i \\ \lambda(t)\Delta t & \text{if } j = i+1 \\ o(\Delta t) & \text{otherwise} \end{cases}$$

Now, let $P_m(t)$ denote the probability of $M(t)$ being m, then, by (3.2), we have

$$\begin{aligned} P_m(t+\Delta t) &= P[M(t+\Delta t) = m] \\ &= [1 - \lambda(t)\Delta t]P_m(t) + \lambda(t)\Delta t P_{m-1}(t) + o(\Delta t). \end{aligned}$$

Solving for $P_m(t)$ from the above, we get:

$$P_m(t) = \frac{\mu^m(t)}{m!} e^{-\mu(t)}, \tag{3.3}$$

where

$$\mu(t) = \int_0^t \lambda(x)dx \tag{3.4}$$

is the mean value function of $M(t)$ for $t \geq 0$.

The actual steps involved in solving (3.3) is routine but tedious. We leave them to the last section of this chapter. (See Proposition 3.6.1)

Suppose that during the time interval $(0, t_e]$, the software under the study has experienced m_e failures. The probability that the software will have m failures by time $t(> t_e)$ is given by the conditional probability $P[M(t) = m \mid M(t_e) = m_e]$. Note that since the Poisson process $[M(t),\ t \geq 0]$ has independent increments, the conditional probability can then be calculated as:

$$\begin{aligned} P[M(t) = m \mid M(t_e) = m_e] &= P[M(t) - M(t_e) = m - m_e] \\ &= \frac{[\mu(t) - \mu(t_e)]^{m-m_e}}{(m-m_e)!} e^{-[\mu(t)-\mu(t_e)]}. \end{aligned} \tag{3.5}$$

Therefore, we have proved the following proposition.

Proposition 3.3.1 *With the assumptions [A1 - A4], the probability of $m - m_e$ additional failures will occur during $(t_e, t]$, given that there have been m_e failures during the time interval $(0, t_e]$ can be obtained by Equation (3.5).*

This proposition answers the question **Q1**.

We note that the number of *remaining* failures is again a random variable. The above distribution can also provide an answer to the question of how many more failures will be experienced in the future, given that m_e failures have been experienced during $(0, t_e]$, as shown in the next proposition.

Proposition 3.3.2 *The probability that there are q remaining failures, given that m_e failures have been experienced during $(0, t_e]$, is:*

$$P[M(\infty) - M(t_e) = q] = \frac{[\mu(\infty) - \mu(t)]^q}{(q)!} e^{-[\mu(\infty) - \mu(t_e)]}. \tag{3.6}$$

Proof. Let $t = \infty$ in (3.5). □

We would like to provide answers to the following questions typically asked by the end user of software products:

Q3: How long does it take to experience a certain number of failures?

Q4: What is the probability of failure-free operation during a certain amount of time?

Let T_i denote the random variable representing the ith failure time. First we note that the following two events are the same:

1. the event that there are at least i failures experienced by time t, denoted by $[M(t) \geq i]$;

2. the event that time to the ith failure is at most t, denoted by $[T_i \leq t]$.

The following proposition answers question **Q3**.

Proposition 3.3.3 *The probability that the time to the ith failure ($i \geq m_e$) is at most t is:*

$$\begin{aligned} P[T_i \leq t] &= P[M(t) \geq i] \\ &= \sum_{j=i}^{\infty} \frac{[\mu(t)]^j}{j!} e^{-\mu(t)}. \end{aligned} \tag{3.7}$$

Proof. This follows from (3.3) directly. □

Given that there are m_e failures experienced by the time t_e, the conditional probability that the time to the ith failure is at most t can be derived from Equation (3.5):

$$\begin{aligned} P[T_i \leq t \mid M(t_e) = m_e] &= P[M(t) \geq i \mid M(t_e) = m_e] \\ &= \sum_{j=i}^{\infty} P[M(t) - M(t_e) = j - m_e] \\ &= \sum_{j=i}^{\infty} \frac{[\mu(t) - \mu(t_e)]^{j-m_e}}{(j-m_e)!} e^{-[\mu(t)-\mu(t_e)]}. \end{aligned} \tag{3.8}$$

In particular, if $m_e = i - 1$, we have

$$\begin{aligned} P[T_i \leq t \mid M(t_{i-1}) = i-1] &= \sum_{j=i}^{\infty} \frac{[\mu(t) - \mu(t_{i-1})]^{j-i+1}}{(j-i+1)!} e^{-[\mu(t)-\mu(t_{i-1})]} \\ &= e^{-[\mu(t)-\mu(t_{i-1})]} \sum_{k=1}^{\infty} \frac{[\mu(t) - \mu(t_{i-1})]^k}{k!} \\ &= e^{-[\mu(t)-\mu(t_{i-1})]} \left[e^{\mu(t)-\mu(t_{i-1})} - 1\right] \\ &= 1 - e^{-[\mu(t)-\mu(t_{i-1})]}. \end{aligned} \tag{3.9}$$

Therefore, we have proved the following proposition:

Proposition 3.3.4 *The conditional probability that the time to the ith failure is at most t, given that there are $i-1$ failures experienced by the time t_{i-1}, is*

$$P[T_i \leq t \mid M(t_{i-1}) = i-1] = 1 - e^{-[\mu(t)-\mu(t_{i-1})]}. \tag{3.10}$$

Let $T_i'(i = 1, 2, \ldots)$ be a random variable representing the ith failure interval; t' be a realization of T_i'. Recall that $T_i(i = 1, 2, \ldots)$ is a random variable representing the ith failure time. Then,

$$T_i = \sum_{j=1}^{i} T_j' = T_{i-1} + T_i', \quad i = 1, 2, \ldots,$$

with $T_0 = 0$.

With previous propositions, it is easy to derive various quantities associated with reliability.

First, the conditional reliability of T_i' on the last failure time $T_{i-1} = t_{i-1}$ can be derived as follows:

$$\begin{aligned} R(t_i' \mid t_{i-1}) &= P[T_i' > t_i' \mid T_{i-1} = t_{i-1}] \\ &= 1 - P[T_i \leq t_i \mid M(t_{i-1}) = i - 1] \\ &= e^{-[\mu(t_{i-1}+t_i') - \mu(t_{i-1})]}, \quad i = 1, 2, \ldots \end{aligned} \tag{3.11}$$

This establishes the following proposition, which answers question **Q4**.

Proposition 3.3.5 *If the software has experienced $i-1$ failures by the time t_{i-1}, then the probability that the software will operate failure-free for at least t' is given by*

$$e^{-[\mu(t_{i-1}+t_i') - \mu(t_{i-1})]}, \quad i = 1, 2, \ldots$$

Next, recall that, for the failure distribution $F(t)$, we have $F(t) = 1 - R(t)$ and the failure density $f(t)$ is the derivative of $F(t)$ and $\mu(t)$ is the derivative of $\lambda(t)$. Then the conditional failure density function can be expressed as follows:

$$f(t_i' \mid t_{i-1}) = \lambda(t_{i-1} + t_i') e^{-[\mu(t_{i-1}+t_i') - \mu(t_{i-1})]}, \quad i = 1, 2, \ldots . \tag{3.12}$$

Thirdly, from Equation (3.12), we can also show that the program hazard rate, using the fact that $z(t) = f(t)/R(t)$, as follows:

$$z(t_i' \mid t_{i-1}) = \lambda(t_{i-1} + t_i'), \quad i = 1, 2, \ldots . \tag{3.13}$$

Next, we assueme that each failure, caused by a fault, occurs independently and randomly, according to the "pre-fault hazard rate" $z_a(t)$, which is the instantaneous failure rate caused by the fault a. Let T_a be the random variable representing time to failure of a fault "a", where $a = 1, 2, \ldots$. We make the following assumption:

The random variables T_a are independently distributed as $F_a(t)$ for all remaining faults.

Let $f_a(t)$ be the probability density function of $T_a(t)$. That is:

$$f_a(t) = \frac{dF_a(t)}{dt}. \tag{3.14}$$

The *per-fault hazard rate* $z_a(t)$ can be written as:

$$z_a(t) = \frac{f_a(t)}{1 - F_a(t)}. \tag{3.15}$$

The per-fault hazard rate gives the instantaneous failure rate at time t for the failure a. As we are going to see in the next section, for Musa basic model, the per-fault hazard rate is assumed to be a constant.

Now, we obtain the cumulative distribution function $F_a(t)$ from the per-fault hazard rate $z_a(t)$:

$$F_a(t) = 1 - e^{-\int_o^t z_a(x)dx}. \tag{3.16}$$

Recall that we have used $U(t)$ to denote a random variable representing faults remaining at time t. If we assume that the number of faults remaining at $t = 0$ follows a Poisson distribution with a mean ω_0, we see that the distribution of the number of failures experienced by time t is given by the following;

$$P[M(t) = m] = \sum_{x=0}^{\infty} P[M(t) = m \mid U(0) = x]P[U(0) = x].$$

Solving the above, we obtain:

$$P[M(t) = m] = \frac{[\omega_0 F_a(t)]^m}{m!} e^{-\omega_0 F_a(t)}. \tag{3.17}$$

The derivation of the solution (3.17) is routine but lengthy. We postpone it to the end of this chapter. (See Proposition 3.6.3).

The mean value function of this distribution is

$$\mu(t) = \omega_0 F_a(t). \tag{3.18}$$

The failure intensity function can be obtained by differentiating $\mu(t)$ as

$$\lambda(t) = \omega_0 f_a(t). \tag{3.19}$$

The conditional reliability of T_i' on the last failure time $T_{i-1} = t_{i-1}$, Equation (3.11), now becomes:

$$\begin{aligned} R(t_i' \mid t_{i-1}) &= P[T_i' > t_i' \mid T_{i-1} = t_{i-1}] \\ &= e^{-\omega_0[F_a(t_{i-1}+t_i')-F_a(t_{i-1})]}, \quad i = 1, 2, \ldots \end{aligned} \tag{3.20}$$

Note from Equation (3.20) that the reliability for the Poissone-type models depends on the last failure time t_{i-1}, but not on the failures remaining. Note also that if we let $t_i' = \infty$ in Equation (3.20), that is we consider the reliability in an infinite amount of time after the failure at t_i, we have, from (3.16),

$$R(\infty \mid t_{i-1}) = e^{-\omega_0[1-F_a(t_{i-1})]}.$$

Since $e^x > 0$, we see that the Poisson-type models allow nonzero probability of no failures in an infinite amount of time.

Substituting (refcarl2) into (refhazard), we obtain the program hazard rate as

$$z(t_i' \mid t_{i-1}) = \omega_0 f_a(t_{i-1} + t_i').$$

This shows that the program hazard rate for the ith failure interval can be determined if the last failure time t_{i-1} is known.

For some Poisson-type reliability models, we can relax some of the assumptions we have made. We can consider imperfect repair process where faults cannot be located, extra faults are spawned. For such imperfect fault removal process we

can assume that the fault correction rate is proportional to the hazard rate. Let B denote the *fault reduction factor*. The fault correction rate is then the product of the fault reduction factor B and the hazard rate $z_a(t)$.

For a given number of faults ω_0 the total expected number of failures υ_0 will be $\upsilon_0 = \frac{\omega_0}{B}$.

Let $G_a(t)$ be the cumulative distribution function of time to remove a fault, then

$$G_a(t) = 1 - e^{-B\int_0^t z_a(x)dx}$$

Denote the probability density function associated with $G_a(t)$ by $g_a(t)$. Thus, $g_a(t)$ is the density function of time to remove a fault.

Now, the distribution of the random variable representing the number of failures experienced by time t can be approximated by the Poisson distribution:

$$P[M(t) = m] = \frac{[\upsilon_0 G_a(t)]^m}{m!} e^{-\upsilon_0 G_a(t)}, \quad m = 0, 1, \ldots,$$

The mean value of the distribution is

$$\mu(t) = \upsilon_0 G_a(t);$$

the failure intensity function is

$$\lambda(t) = \upsilon_0 g_a(t);$$

and the program hazard rate is given

$$z(t_i'|t_{i-1}) = \upsilon_0 g_a(t_{i-1} + t_i').$$

Next, we study two reliability models that are widely used in software testing theories and applications. The two models are the *Musa Basic Model* and the *Logarithmic Poisson Model*. These two models use an effective dual approach for characterizing failure behavior. The parameters in these two models have a clear physical interpretation. Both models assume that failure occurs as a nonhomogeneous Poisson process. The difference between the two models is best described in

terms of slope or decrement per failure experienced. The decrement in the failure intensity function remains constant for the basic execution time model whether it is the first failure that is being fixed or the last. In contrast, for the logarithmic Poisson execution time model, the decrement per failure decreases exponentially. The first failure initiates a repair process that yields a substantial decrement in failure intensity, while later failures result in much smaller decrements. Note that both models use CPU execution time as the model variable. To distinguish CPU execution time from calendar time, we denote the time varible in the models as τ, instead of t, following [23].

3.4 Musa Basic Execution Model

The basic execution time model is claimed in the literature to be the most practical model of all. Farr mentions that this model has had the widest distribution among software reliability models [8]. The parameters of the model have explicit physical interpretation and can be related to information that exists before the program has been executed. We assume that there are finitely many faults in a software system. Let ω_0 be the number of inherent faults. Let υ_0 be the total failures that would be experienced in infinite time. Through the various stages of testing, faults are discovered and removed. So the total number of faults inherent in the program is reduced. The net number of faults removed is only a portion of the failures experienced. The total number of faults corrected is frequently larger than the net number of faults. The reason is that in the correction process, some new faults are introduced. The fault reduction factor, denoted by B, is the ratio of net fault reduction to failures experienced as time of operation approaches infinity. The fault reduction factor B can be used to provide an estimation of υ_0:

$$\upsilon_0 = \frac{\omega_0}{B}. \tag{3.21}$$

Assume that the pre-fault hazard rate $z_a(\tau)$ is a constant ϕ. We derive the Musa Basic model based on the postulate that the failure intensity has a constant slope

with respect to average failure experienced. We write:

$$\lambda(\mu) = \phi B(\upsilon_0 - \mu),$$

or

$$\lambda(\tau) = \phi B[\upsilon_0 - \mu(\tau)], \tag{3.22}$$

where ϕ is the pre-fault hazard rate.

Since $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we have

$$\frac{d\mu(\tau)}{d\tau} + \phi\, B\, \mu(\tau) = \phi\, B\, \upsilon_0.$$

Solving this differential equation, we have

$$\mu(\tau) = \upsilon_0 \left[1 - e^{-\phi B \tau}\right]. \tag{3.23}$$

By differentiating (3.23) with respect to τ, we obtain the expression for the failure intensity $\lambda(\tau)$:

$$\lambda(\tau) = \upsilon_0\, \phi\, B\, e^{-\phi B \tau}. \tag{3.24}$$

i(3.23) and (3.24) and with formula (3.7) we established in the previous section, we can get the cumulative probability distribution of time to the ith failure as the following:

$$P[T_i \leq \tau] = e^{-\upsilon_0[1-e^{-\phi B \tau}]} \sum_{j=i}^{\infty} \frac{\upsilon_0^j\, [1 - e^{-\phi B \tau}]^j}{j!}.$$

The software reliability of this model can be obtained by substituting the equation of the expected number of failures experienced by τ (3.23) into the equation of the conditional reliability (3.11):

$$R(\tau_i' \mid \tau_{i-1}) = e^{-[\upsilon_0 \exp(-\phi B \tau_{i-1})][1-\exp(-\phi B \tau_i')]}. \tag{3.25}$$

For the convenience of the discussion in the next chapter, we denote $\beta_0 = \upsilon_0$ and $\beta_1 = \phi B$ so that the basic model can be described by $\mu(\tau)$ or $\lambda(\tau)$ as following:

$$\mu(\tau) = \beta_0(1 - e^{-\beta_1 \tau});$$
$$\lambda(\tau) = \beta_0\, \beta_1\, e^{-\beta_1 \tau}.$$

Parameter β_0 is the total faults that would be experienced in infinite time and parameter β_1 is the per-fault hazard rate.

The basic model implies a uniform user operational profile. With the highly nonuniform user operational profiles, where some functions are executed much more frequently than others, the logarithmic Poisson model may be superior. We discuss the model in the next section.

3.5 Musa-Okumoto Logarithmic Poisson Model

The logarithm Poisson execution time model established by Musa and Okumoto in 1984 has been widely adopted by the researchers and practitioners in the area of software reliability theory and practice. Malaiya et al [20] evaluated the prediction accuracy of five two-parameter models and concluded that the logarithmic model has the best overall prediction capability. They also found that this superiority is statistically significant. From our experience, another pleasant property of this model is that the computation for estimating the parameters in model is very stable, compared with the Musa basic model.

The logarithmic Poisson model can be derived based on the postulate that the intensity function decreases exponentially with respect to the expected failures experienced:

$$\lambda(\tau) = \lambda_0 \, e^{-\theta \, \mu(\tau)},$$

where λ_0 denotes the initial failure intensity and $\theta > 0$ denoted the failure intensity decay parameter.

The quantity μ represents the expected number of failures. Given the initial failure intensity λ_0 and the failure intensity decay parameter θ, we can derive the expression of the expected number of failures μ and the failure intensity function $\mu(\tau)$ in terms of λ_0 and θ. Indeed, since $\lambda(\tau)$ is the derivative of $\mu(\tau)$, we have:

$$\frac{d\,\mu(\tau)}{d\,\tau} = \lambda_0 \, e^{-\theta \mu(\tau)}.$$

Multiplying both sides by $e^{-\theta\mu(\tau)}$, we get

$$\frac{d\,\mu(\tau)}{d\,\tau}\,e^{\theta\mu(\tau)} = \lambda_0.$$

Solving this differential equation with the initial condition $\mu(0) = 0$, we get:

$$\mu(\tau) = \frac{1}{\theta}\ln(\lambda_0\theta\tau + 1). \tag{3.26}$$

Notice that μ is a logarithm function of τ, so the name *logarithmic Poisson Model.* The failure intensity function is obtained by taking the derivative of (3.26) with respect to τ.

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0\,\theta\,\tau + 1}. \tag{3.27}$$

The cumulative probability distribution of time to failure is obtained by substituting Equations (3.26) and (3.27) into Equation (3.7):

$$P[T_i \le \tau] = (\lambda_0\,\theta\,\tau + 1)^{-\frac{1}{\theta}} \sum_{j=i}^{\infty} \frac{\ln(\lambda_0\,\theta\,\tau + 1)^j}{\theta^j\;j!}. \tag{3.28}$$

The program reliability is obtained by substituting equations (3.26) into Equation (3.11):

$$R(\tau_i' \mid \tau_{i-1}) = \left[\frac{\lambda_0\,\theta\,\tau_{i-1} + 1}{\lambda_0\,\theta\,(\tau_{i-1} + \tau_i') + 1}\right]^{\frac{1}{\theta}}. \tag{3.29}$$

The program hazard rate is obtained by substituting equation (3.27) into Equation (3.13).

$$z(\tau_i' \mid \tau_{i-1}) = \frac{\lambda_0}{\lambda_0\,\theta\,(\tau_{i-1} + \tau_i') + 1}. \tag{3.30}$$

The mean-time-to-failure (MTTF) is defined only for $\theta < 1$, which is generally the case in actual software systems. By substituting Equation (3.29) into Equation (3.1):

$$\Theta\,(\tau_{i-1}) = \frac{1}{\lambda_0(1-\theta)}\,[\lambda_0\,\theta\,\tau_{i-1} + 1]\,. \tag{3.31}$$

Again, for the convenience of the following chapters, we write $\beta_0 = \dfrac{1}{\theta}$ and $\beta_1 = \lambda_0\,\theta$. Then the logarithmic model can be written as:

$$\mu(\tau) = \beta_0\,\ln(1 + \beta_1\,\tau);$$
$$\lambda(\tau) = \frac{\beta_0\,\beta_1}{1 + \beta_1\,\tau}.$$

3.6 Proofs of Some Propositions

In this section, we supply some technical details on derivations of some results we have discussed in the previous sections. These derivations require mainly elementary calculus. However, since they are fairly lengthy, we gather them in this section.

Proposition 3.6.1 *Under the assumption [A1-A4], let*

$$P_m(t+\Delta t) = [1-\lambda(t)\Delta t]P_m(t) + \lambda(t)\Delta t P_{m-1}(t) + o(\Delta t), \tag{3.32}$$

where $\lim_{\Delta t\to 0}\frac{o(\Delta t)}{\Delta t} = 0$. *Then,*

$$P_m(t) = \frac{\mu^m(t)}{m!}e^{-\mu(t)},$$

where

$$\mu(t) = \int_0^t \lambda(x)dx$$

is the mean value function of $M(t)$ *for* $t \geq 0$.

Proof. From (3.32), it is clear that

$$\frac{P_m(t+\Delta t) - P_m(t)}{\Delta t} = -\lambda(t)P_m(t) + \lambda(t)P_{m-1}(t) + \frac{o(\Delta t)}{\Delta t}.$$

Let $\Delta t \to 0$, we have

$$\frac{d(P_m(t))}{dt} = -\lambda(t)P_m(t) + \lambda(t)P_{m-1}(t)$$

Multiplying both sides of the above by $e^{\mu(t)}$, we have:

$$e^{\mu(t)}\left[\frac{dP_m(t)}{dt} + \lambda(t)P_m(t)\right] = \lambda(t)\, e^{\mu(t)}\, P_{m-1}(t).$$

Therefore,

$$\frac{d[e^{\mu(t)}P_m(t)]}{dt} = \lambda(t)\, e^{\mu(t)}\, P_{m-1}(t).$$

Let $m = 0$, noting that $P_{-1}(t) = 0$, we get

$$e^{\mu(t)} P_0(t) = C,$$

for some constant C. Since $P_0(0) = 1$ (by assumption A1) and $\mu(0) = 0$, we see that $C = 1$. Thus,

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

This verifies that for $m = 0$, the proposition holds. Now, assume that the proposition holds for $m = k$:

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

Then from Equation (3.6),

$$\frac{d[e^{\mu(t)} P_{k+1}(t)]}{dt} = \lambda(t)\, e^{\mu(t)}\, \frac{[\mu(t)]^k}{k!}\, e^{-\mu(t)}$$

Integrating,

$$\begin{aligned} e^{\mu(t)}\, P_{k+1}(t) &= \int_0^t \lambda(x)\, \frac{[\mu(x)]^k}{k!} dx + C \\ &= \int_0^t \frac{[\mu(x)]^k}{k!} d(\mu(x)) + C \\ &= \frac{[\mu(t)]^{k+1}}{(k+1)!} + C. \end{aligned}$$

Since $P_{k+1}(0) = 0$, $\mu(0) = 0$, we have $C = 0$. Therefore

$$P_{k+1}(t) = \frac{[\mu(t)]^{k+1}}{(k+1)!} e^{-\mu(t)}.$$

This verifies that the proposition holds for $m = k + 1$. The induction completes the proof. □

Proposition 3.6.2 *With the assumptions that there are u_0 inherent faults in the program; whenever a failure occurs, the fault that caused it will be removed instantaneously; and the hazard rates for all faults are the same, then*

$$P[M(t) = m] = \binom{u_0}{m} [F_a(t)]^m [1 - F_a(t)]^{u_0 - m}, m = 0, 1, \ldots, u_0$$

where $\binom{u_0}{m} = \frac{u_0!}{m!(u_0-m)!}$ *are the binomial coefficients and*

$$F_a(t) = 1 - e^{-\int_0^t z_a(x)dx}$$

is the per-fault cumulative distribution function.

Proof. Note that by assumptions, the total number of faults in the program is a fixed number u_0. If $m-1$ failures have occurred by time t, there are $u_0-(m-1)$ faults remaining at time t and each fault has the hazard rate $z_a(t)$, and the probability of a failure in $(t, t+\Delta t)$ is given by

$$(u_0 - m + 1)z_a(t)\Delta(t) + o(\Delta t).$$

Similarly, when m failures have occurred by time t, there are $u_0 - m$ faults remaining at time t, and the probability of no failures in $(t, t+\Delta t)$ is $1-(u_0 - m)z_a(t)\Delta t$. Therefore, we see that

$$\begin{aligned} P_m(t+\Delta t) &= P[M(t+\Delta t) = m] \\ &= P[\text{a failure in } (t, t+\Delta t) \mid m-1 \text{ failures in } (0,t)]\ P_{m-1}(t) \\ &\quad + P[\text{no failures in}(t, t+\Delta t) \mid m \text{ failures in } (0,t)]\ P_m(t) \\ &= (u_0 - m + 1)z_a(t)\Delta t P_{m-1}(t) + [1-(u_0-m)z_a(t)\Delta t]P_m(t) + o(\Delta t). \end{aligned}$$

Rearranging it, we have:

$$P_m(t+\Delta t) - P_m(t) = (u_0 - m + 1)z_a(t)\Delta t P_{m-1}(t) - (u_0 - m)z_a(t)\Delta t P_m(t) + o(\Delta t).$$

Thus:

$$\frac{P_m(t+\Delta t) - P_m(t)}{\Delta t} = (u_0 - m + 1)z_a(t)P_{m-1}(t) - (u_0 - m)z_a(t)P_m(t) + \frac{o(\Delta t)}{\Delta t}.$$

Let $\Delta t \to 0$, we have

$$\frac{dP_m(t)}{dt} = (u_0 - m + 1)z_a(t)P_{m-1}(t) - (u_0 - m)z_a(t)P_m(t), m = 0, 1, \ldots, u_0. \tag{3.33}$$

To solve the above differential equation with the boundary conditions:

$$P_0(0) = 1, P_m(0) = 0, m = 1, 2, \ldots, u_0,$$

we use induction again.

Let $m = 0$, keep in mind that $P_{-1}(t) = 0$, we have:

$$\frac{dP_0(t)}{dt} = -u_0 z_a(t) P_0(t)$$

That is,

$$\ln \mid P_0(t) \mid = -u_0 \int_0^t z_a(x)dx + C.$$

Thus

$$P_0(t) = C_1 e^{-u_0 \int_0^t z_a(x)dx}$$

To determine the constant C_1, we let $t = 0$, and note that $P_0(0) = 1$, we see that $C_1 = 1$. Thus, by equation (3.16),

$$P_0(t) = e^{-u_0 \int_0^t z_a(x)dx} = [e^{-\int_0^t z_a(x)dx}]^{u_0} = [1 - F_a(t)]^{u_0}$$

This verifies that the proposition holds for $m = 0$. Assume that the proposition holds for $m = k$:

$$P_k(t) = P[M(t) = k] = \binom{u_0}{k} [F_a(t)]^k [1 - F_a(t)]^{u_0 - k} \tag{3.34}$$

Multiply $(1 - F_a(t))^{u_0-(k+1)}$ on both sides of (3.33) with $m = k+1$ and rearrange to get:

$$[1 - F_a(t)]^{-u_0+(k+1)} \left[\frac{dP_{k+1}(t)}{dt} + (u_0 - k - 1) z_a(t) P_{k+1}\right] = (u - k) z_a(t) P_k(t) [1 - F_a(t)]^{-u_0+(k+1)}.$$

Substituting (3.34), and noticing that the left hand side of the above equation is

$$\frac{d([1 - F_a(t)]^{-u_0+(k+1)} P_{k+1}(t))}{dt},$$

we have

$$\frac{d([1-F_a(t)]^{-u_0+(k+1)}P_{k+1}(t))}{dt} = (u-k)\binom{u_0}{k} f_a(t)[F_a(t)]^k.$$

Therefore,

$$\begin{aligned}[1-F_a(t)]^{-(u_0-k-1)}P_{k+1}(t) &= \int_0^t (u_0-k)\binom{u_0}{k} F_a(t)[F_a(t)]^k dt \\ &= \frac{u_0-k}{k+1}\binom{u_0}{k} F_a^{k+1}(t) + C.\end{aligned}$$

Again, using $P_{k+1}(0) = 0$, it is easy to see that $C = 0$.

Thus

$$P_{k+1}(t) = \binom{u_0}{k+1}[1-F_a(t)]^{u_0-(k+1)}[F_a(t)]^{k+1}.$$

This completes the proof. □

Proposition 3.6.3 *Suppose that*

(1) Whenever a software failure occurs, the fault that caused it will be removed instantaneously,

(2) The total number of faults remaining in the program at $t = 0$, $U(0)$, is a Poisson random variable with mean ω_0,

(3) The hazard rates for all faults are the same.

Then the distribution of the failure experienced $M(t)$ is

$$P[M(t) = m] = \frac{[\omega_0 F_a(t)]^m}{m!} e^{-\omega_0 F_a(t)}$$

Proof. Since the total number of faults remaining in the program at time $t = 0$, $U(0)$, follows a Poisson distribution,

$$P[U(0) = x] = \frac{\omega_0^x}{x!} e^{-\omega_0}.$$

According to the previous proposition, the distribution of $M(t)$ is

$$\begin{aligned} P[M(t)=m] &= \sum_{x=0}^{\infty} P[M(t)=m] \mid U(0)=x]P[U(0)=x] \\ &= \sum_{x=0}^{\infty} \binom{x}{m}[F_a(t)]^m[1-F_a(t)]^{x-m}\frac{w_0^x}{x!}e^{-\omega_0} \\ &= \frac{[\omega_0 F_a(t)]^m}{m!}e^{-\omega_0}\sum_{x=m}^{\infty}\frac{[\omega_0(1-F_a(t))]^{x-m}}{(x-m)!} \\ &= \frac{[\omega_0 F_a(t)]^m}{m!}e^{-\omega_0 F_a(t)} \end{aligned}$$

because $P[M(t)=m \mid U(0)=x]=0$ if $x<m$ and

$$\sum_{x=m}^{\infty}\frac{[\omega_0(1-F_a(t))]^{x-m}}{(x-m)!} = e^{\omega_0(1-F_a(t))}.$$

This completes the proof. □

3.7 Summary

In this chapter, we present general theory of software reliability growth models. The presentation follows closely with [23]. We have focused on Poisson-type models, which can be derived based on the assumption that the initial number of faults in a software product is not known with certainty. We assume the random variable representing the the initial number of faults follows a Poisson distribution. Several propositions are presented to provide answers to the questions commonly asked by product managers and end users of software products regarding their reliability properties. Two well-known software reliability growth models: the exponential models and the logarithmic models are derived by postulating special properties of the failure intensity function. We also see that, in the exponential models, the parameter β_0 represents the total faults that would be experienced in infinite time and the parameter β_1 represents the per-fault hazard rate. We will provide an illustration of the theory presented here in a later chapter.

Chapter 4

Parameter Estimation for SRGM

In the previous chapter, we have reviewed several important software reliability growth models. We have seen that for Musa basic model (the exponential model) and the logarithmic model, there are two parameters β_0 and β_1 that are to be estimated for each specific software project using the information related to the project. Estimation of parameters of software reliability growth models have been studied extensively by many researchers and software engineers [23, 12, 10]. In many situations, one would like to have some idea about software reliability even before starting the testing process. Malaiya et al [18, 14, 15, 21] have developed some techniques to estimate the parameters of the exponential model and the logarithmic model based on the information that is available before testing, without using the actual failure time data. In the first section, we present some of their work. In the remaining sections, we illustrate how some well developed general parameter estimation techniques such as maximum likelihood estimation and least error estimation can be applied to the exponential model and the logarithmic model using failure time data that are collected during testing.

Note that we consider only point estimation in this thesis.

4.1 Parameter Estimation Prior to Testing

One of the important features of the reliability growth models we have studied in the previous chapter is that the parameters in the models have clear physical

interpretations. The value of initial failure intensity λ_0, total failures experienced υ_0 and the failure intensity decay parameter θ_0 are usually predicted before the software is executed. Once the program has executed long enough so that failure data are available, we can *estimate* these parameters. Estimation is a statistical method that is based on either the failure times or the number of failures per time interval. First, some terminologies is given. Again, all these concepts and notations are standard in reliability literature. In particular, they can be found in [23] and [14].

In practice, it has been assumed that the size and complexity of a program has the most effect on the number of inherent faults it contains. As many researchers and software testers have discovered, the total number of faults inherent in a software system is linearly related to the software-measure of lines of code. Studies have developed estimates of fault density ranging from 3.0 to 5.0 faults per 1000 lines of code (KLOC)[7, 28]. Leading edge software development organizations typically achieve a fault density of about 2.0 faults per KLOC. The NASA Space Shuttle Avionics software has an estimated fault density of 0.1 defects per KLOC, which can be regarded as an example of what can be currently achieved by the best methods. A low defect density can be quite expensive to achieve. The Space Shuttle code has been reported to have cost about $2000 per line of code [9]

If we know the inherent fault density, then the number of inherent faults ω_0 is estimated by multiplying the inherent fault density and the program size. The size of a program is usually measured by the number of source statements, denoted by I_s. It is often convenient to convert I_s to I, where I is the *number of object instructions.* The relationship between I and I_s is described by the following formula:

$$I = I_s Q_x,$$

where Q_x is the so called the *average expansion ratio.* Musa et al [23] and other researchers [14, 15, 19] use $Q_x = 4$.

Let r be the *average instruction execution rate.* r is usually measured in MIPS: millions of instructions per second. For example, the average instruction

execution rate for Intel 486 is approximately 5 MIPS; Intel Pentium is about 90 MIPS.

Let f denote the *linear execution frequency* of a program. f is the number of times the program would be executed per unit time if it had no branches or loops. f is determined by the following formula:

$$f = \frac{r}{I},$$

Another important parameter in reliability models is the *fault exposure ratio* (FER) K, which represents the average detectability of the faults in software. Li and Malaiya [15] relate fault density to FER and present a model to predict FER.

If we have estimated the value of K, then the parameters β_0 and β_1 in the exponential model and the logarithmic model, can also be estimated.

In the following, we present their approach for estimating parameters β_0 and β_1 prior to actual testing.

Let T_L be the linear execution time, then

$$T_L = \frac{I_s Q_x}{r}.$$

Let $N(t)$ be the expected number of faults present in the system at time t, and denote $N_0 = N(0)$. Then, $\mu(t) = N_0 - N(t)$. It is proved in [21] that:

$$\frac{dN(t)}{dt} = -\frac{K}{T_L} N(t). \tag{4.1}$$

The actual amount of testing done in phases like unit testing and integration testing can very significantly for different project. Thus if we use only the testing time in a specific phase, it would not take into account the debugging that has taken place in the prior phases. Thus, *fault density* is used in [14] as a measure of the test stage, because it is independent of the project to project variation of the specific stage at which a given test phase begins. The fault exposure rate K is related to test stage. The actual amount of testing done in phases like unit testing and integration testing can vary significantly for different projects. Let

$D(t)$ be the fault density of time t, defined by

$$D(t) = \frac{N(t)}{I_s}.$$

In order to derive a formula to estimate parameters β_0, β_1 in the logarithmic model $\mu(t) = \beta_0 \ln(1 + \beta_1 t)$, it is convenient to express the fault exposure ratio K as a function of D. As we shall see below, such an expression will allow us to write β_0 and β_1 as functions of some qualities that can be estimated prior to the test.

Note that the logarithmic model $\mu(t) = \beta_0 \ln(1 + \beta_1 t)$ can be written as

$$e^{\frac{\mu(t)}{\beta_0}} = 1 + \beta_1 t.$$

Then the fault intensity function $\lambda(t)$ is:

$$\lambda(t) = \beta_0 \beta_1 e^{-\frac{\mu(t)}{\beta_0}}.$$

Therefore, using $\mu(t) = N(0) - N(t) = N(0) - I_s D(t)$, we have:

$$\lambda(t) = \beta_0 \beta_1 \exp\left[-\frac{N_0 - I_s D(t)}{\beta_0}\right]. \tag{4.2}$$

Note that from Equation (4.1) and the fact that $\frac{dN(t)}{dt} = -\lambda(t)$, we have

$$K(t) = T_L \frac{\lambda(t)}{N(t)}. \tag{4.3}$$

Substituting Equation (4.2) into the above, we have

$$\begin{aligned} K(t) &= \frac{T_L}{I_s D} \beta_0 \beta_1 \exp\left[-\frac{N_0 - I_s D}{\beta_0}\right] \\ &= \frac{T_L}{I_s D} \beta_0 \beta_1 \exp\left[-\frac{N_0}{\beta_0}\right] \exp\left[\frac{I_s D}{\beta_0}\right]. \end{aligned}$$

Let

$$\alpha_1 = \frac{I_s}{\beta_0}, \tag{4.4}$$

and

$$\alpha_0 = \frac{\beta_0 \beta_1 T_L}{I_s} e^{-\frac{N_0}{\beta_0}}. \tag{4.5}$$

Then we can write K as a function of D with parameters α_0 and α_1:

$$K(D) = \frac{\alpha_0}{D}\, e^{\alpha_1 D}.$$

From the definitions of α_0 and α_1, we see that if we know the values of β_0 and β_1, then we can compute the values of α_0 and α_1, because T_L, I_s and N_0 are usually available. Therefore, for each value of D, we can find the corresponding value of K. So the fault exposure ratio K is related to the testing stage measured by fault density D.

On the other hand, if we can estimate the values of α_0 and α_1, we can also estimate the values of β_0 and β_1.

The parameters α_0 and α_1 are related to the fault exposure ratio K and the default density D. Indeed, if we take the derivative of K with respect to D:

$$\frac{dK}{dD} = -\frac{\alpha_0}{D^2}\, e^{\alpha_1 D} + \frac{\alpha_0 \alpha_1}{D}\, e^{\alpha_1 D}.$$

and by setting $\frac{dK}{dD} = 0$, we find

$$D_{\min} = \frac{1}{\alpha_1}. \tag{4.6}$$

This shows that α_1 is the reciprocal of the minimum value of D. The minimum value of K is

$$K_{\min} = \frac{\alpha_0 e}{D_{\min}}. \tag{4.7}$$

Thus,

$$\alpha_0 = \frac{K_{\min} D_{\min}}{e}.$$

From $\alpha_1 = \frac{I_s}{\beta_0}$ and (4.6), we see that the parameter β_0 can be estimated from:

$$\beta_0 = \frac{I_s}{\alpha_1} = I_s D_{\min}. \tag{4.8}$$

In order to find an estimation of β_1, we use Equation(4.5):

$$\begin{aligned}\beta_1 &= \frac{\alpha_0 I_s e^{\frac{N_0}{\beta_0}}}{\beta_0 T_L} \\ &= \frac{K_{\min} D_{\min} I_s}{e I_s D_{\min} T_L} \exp\left[\frac{D_0 I_s}{I_s D_{\min}}\right] \\ &= \frac{K_{\min}}{T_L e} \exp\left[\frac{D_0}{D_{\min}}\right]. \end{aligned} \tag{4.9}$$

Therefore, we can estimate β_1 from the above, if we know the estimated values of $K_{\min}$ and $D_{\min}$.

Note that for a specific software project, I_s and T_L are available, and $D_0 = I_s N(0)$ is also usually available. However, $D_{\min}$ is difficult to estimate. The following method is as suggested by Malaiya and Denton [18].

1. If the initial fault density D_0 is less than 10 per KLOC, then the value of $D_{\min}$ can be chosen as 2.

2. Otherwise, $D_{\min}$ can be first estimated as

$$D_{\min} = \frac{D_0}{3}$$

A reasonable estimation for $K_{\min}$ is 1.5×10^{-7} for a typical software project as suggested by Musa et al [23]. Therefore, we can use Equations (4.8) and (4.9) to estimate parameters β_0 and β_1 of the logarithmic model *before* starting of the testing. The method described above provide a rough estimation for the parameters of the logarithmic model when no failure data is available. When enough actual failure data from system testing phase is available, we can use the methods described in the next few sections to estimate β_0 and β_1 more accurately.

To illustrate the application of the technique described above, let us consider an example. Suppose that there is a software system consisting of 50,000 lines of source code. The machine to be used for testing the system has MIPS rate of 16. The source to object instruction ratio is 4. It is estimated that the initial fault density D_0 is 16 faults per thousand line of code (KLOC). Then we take

$D_{\min} = \frac{16}{3} = 5.3$ and β_0 can be estimated as:

$$\beta_0 = I_s D_{\min} = 50 \times 5.3 = 265.$$

The linear execution time T_L can be estimated as

$$T_L = \frac{I_s Q_x}{r} = \frac{50000 \times 4}{16000000} = 0.0125.$$

We use the estimated value of $K_{\min} = 1.5 \times 10^{-7}$. Then we have an estimation for parameter β_1 as:

$$\beta_1 = \frac{K_{\min}}{T_L e} e^3 = \frac{1.5 \times 10^{-7}}{0.0125} e^2 = 8.88 \times 10^{-5}.$$

4.2 Maximum Likelihood Methods

Maximum likelihood estimation has been widely used as a formal parameter estimation technique. The method is to define the so called "*likelihood function*" which is the joint density of the observed data. For each data set, let Y_D be a set of observations of failures, let $\boldsymbol{\beta}$ be a vector of parameters of a software reliability growth model. The parameters $\boldsymbol{\beta}$ are estimated by maximizing the likelihood of occurrence of the set of failures. To do so, we construct a likelihood function $L(\boldsymbol{\beta}, Y_D)$, and maximize the likelihood function with respect to the parameter vector $\boldsymbol{\beta}$.

Suppose that estimation is to be performed at a specified time t_e, not necessarily corresponding to a failure, and with a total of m_e failures being experienced at times $t_1, \ldots, t_{m_e}$. The general likelihood function can be written as

$$L(\boldsymbol{\beta}; t_1, \ldots, t_{m_e}) = f(t_1, \ldots, t_{m_e}) P\left[T_{m_e+1} > t_e \mid T_1 = t_1, \ldots, T_{m_e} = t_{m_e}\right], \tag{4.10}$$

where $f(t_1, \ldots, t_{m_e})$ is the joint function of $(T_1, \ldots, T_{m_e})$ and T_i the random variable denoting the time of the ith failure. The above likelihood function can also be written as:

$$L(\boldsymbol{\beta}; t_1, \ldots, t_{m_e}) = \left[\prod_{i=1}^{m_e} f(t_i \mid t_1, \ldots, t_{i-1})\right] P[T_{m_e+1} > t_e \mid T_1 = t_1, \ldots, T_{m_e} = t_{m_e}]. \tag{4.11}$$

By the Poisson model property,

$$f(t_i \mid t_1, \ldots, t_{i-1}) = f(t_i \mid t_{i-1}),$$

so the likelihood function (4.11) can be simplified as

$$L(\boldsymbol{\beta}) = \left[\prod_{i=1}^{m_e} f(t_i \mid t_{i-1})\right] P[T_{m_e+1} > t_e \mid T_{m_e} = t_{m_e}]. \tag{4.12}$$

It follows from (3.10) that

$$P[T_{m_e+1} > t_e \mid T_{m_e} = t_{m_e}] = e^{-[\mu(t_e)-\mu(t_{m_e})]};$$

and

$$\begin{aligned} f(t_i \mid t_{i-1}) &= -\frac{d\{P[T_i > t_i \mid T_{i-1} = t_{i-1}]\}}{dt_i} \\ &= \lambda(t_i)\, e^{-[\mu(t_i)-\mu(t_{i-1})]}, \end{aligned}$$

where the last equality follows from Equation (3.12).

So Equation (4.12) becomes:

$$L(\boldsymbol{\beta}) = \left[\prod_{i=1}^{m_e} \lambda(t_i)\right] e^{-\mu(t_e)}. \tag{4.13}$$

By applying logarithms to equation (4.13), we get the logarithmic unconditional likelihood function as follows:

$$\ln L(\boldsymbol{\beta}) = \sum_{i=1}^{m_e} \ln \lambda(t_i) - \mu(t_e). \tag{4.14}$$

For exponential class of Poisson type models, we have

$$\mu(t) = \beta_0(1 - e^{-\beta_1 t});$$

and

$$\lambda(t) = \beta_0\beta_1 e^{-\beta_1 t}.$$

Thus, from equation (4.14), the logarithmic unconditional likelihood function for this class is:

$$\begin{aligned}\ln(L(\boldsymbol{\beta})) &= \sum_{i=1}^{m_e} \ln \lambda(t_i) - \mu(t_e) \\ &= \sum_{i=1}^{m_e} \ln\left[\beta_0\beta_1 e^{-\beta_1 t_i}\right] - \beta_0(1 - e^{-\beta_1 t_e}) \\ &= \sum_{i=1}^{m_e} [\ln\beta_0 + \ln\beta_1 - \beta_1 t_i] - \beta_0(1 - e^{-\beta_1 t_e}). \qquad (4.15)\end{aligned}$$

This is the objective function we maximize in the next chapter for our numerical experiments.

The traditional method of maximizing this function is based on calculus: Setting the partial derivative of (4.15) with respect to β_0 to zero, we have

$$\frac{\partial \ln L(\boldsymbol{\beta})}{\partial \beta_0} = \sum_{i=1}^{m_e} \frac{1}{\beta_0} - (1 - e^{-\beta_1 t_e}) = 0,$$

or

$$\frac{m_e}{\beta_0} = 1 - e^{-\beta_1 t_e}.$$

Therefore,

$$\beta_0 = \frac{m_e}{1 - e^{-\beta_1 t_e}}. \qquad (4.16)$$

Setting the partial derivative of $\ln L(\boldsymbol{\beta})$, with respect to β_1, to zero, we have:

$$\frac{\partial \ln L(\boldsymbol{\beta})}{\partial \beta_1} = \sum_{i=1}^{m_e} \frac{1}{\beta_1} - \sum_{i=1}^{m_e} t_i + \beta_0(-t_e)e^{-\beta_1 t_e} = 0.$$

By substituting Equation (4.16) into the above,

$$\sum_{i=1}^{m_e} \frac{1}{\beta_1} - \sum_{i=1}^{m_e} t_i - \frac{m_e t_e\, e^{-\beta_1 t_e}}{1 - e^{\beta_1 t_e}} = 0.$$

The maximum likelihood equation for unconditioned estimation for exponential class of Poisson type models is:

$$\frac{m_e}{\beta_1} - \sum_{i=1}^{m_e} t_i - \frac{m_e t_e\, e^{-\beta_1 t_e}}{1 - e^{\beta_1 t_e}} = 0. \qquad (4.17)$$

Note that this is an equation with only one unknown β_1. Note also that the equation has the same number of terms as the data points. Therefore, when the data points are many, the solution to the equation may not be so easy to obtain numerically. Now, let us consider the logarithmic model. We have

$$\mu(t) = \beta_0 \ln(1 + \beta_1 t);$$

and

$$\lambda(t) = \frac{\beta_0 \beta_1}{1 + \beta_1 t}.$$

Substituting the above to Equation (4.14), we have:

$$\begin{aligned} \ln L(\boldsymbol{\beta}) &= \sum_{i=1}^{m_e} [\ln \beta_0 + \ln \beta_1 - \ln(1 + \beta_1 t_i)] - \beta_0 \ln(1 + \beta_1 t_e) \\ &= m_e \ln \beta_0 + m_e \ln \beta_1 - \sum_{i=1}^{m_e} \ln(1 + \beta_1 t_i) - \beta_0 \ln(1 + \beta_1 t_e). \end{aligned} \quad (4.18)$$

Setting the partial derivative of $\ln L(\boldsymbol{\beta})$ with respect to β_0 to 0, we have:

$$\beta_0 = \frac{m_e}{\ln(1 + \beta_1 t_e)}. \quad (4.19)$$

Setting the partial derivative of $\ln L(\boldsymbol{\beta})$ with respect to β_1 to 0, we have:

$$\frac{m_e}{\beta_1} - \sum_{i=1}^{m_e} \frac{t_i}{1 + \beta_1 t_i} - \frac{\beta_0 t_e}{1 + \beta_1 t_e} = 0. \quad (4.20)$$

Solving the nonlinear equations (4.19) and (4.20), we find the unconditioned maximum likelihood estimates for β_0 and β_1 in the logarithmic model.

The estimation can also be based on a conditional approach. By selecting the time t_e as the basis of observation, the number of failures experienced in the interval $(0, t_e]$ will be a discrete random variable. The conditional likelihood function is obtained by dividing the unconditional likelihood function by the marginal probability of E equaling its observed value m_e because the event in (4.10) contains the event $M(t_e) = m_e$. In other words,

$$L(\boldsymbol{\beta} \mid m_e) = \frac{L(\boldsymbol{\beta})}{P[M(t_e) = m_e]}. \quad (4.21)$$

Using (3.17), (3.18) and (4.13), Equation (4.21) becomes:

$$L(\boldsymbol{\beta} \mid m_e) = \frac{m_e! \prod_{i=1}^{m_e} \lambda(t_i)}{[\mu(t_e)]^{m_e}}. \tag{4.22}$$

By applying logarithms to equation (4.22), the logarithmic conditional likelihood function is as follows:

$$\ln L(\boldsymbol{\beta} \mid m_e) = \ln m_e! + \sum_{i=1}^{m_e} \ln \lambda(t_i) - m_e \ln \mu(t_e). \tag{4.23}$$

Now we consider some special models. First, let us consider the exponential model. By substituting $\lambda(t) = \beta_0\beta_1 e^{-\beta_1 t}$ and $\mu(t) = \beta_0(1 - e^{-\beta_1 t})$ into Equation (4.23), we have

$$\begin{aligned} \ln L(\boldsymbol{\beta} \mid m_e) &= \ln L(\beta_1 \mid m_e) \\ &= \ln m_e! + \sum_{i=1}^{m_e} [\ln \beta_0 + \ln \beta_1 - \beta_1 t_i] - m_e [\ln \beta_0 + \ln(1 - e^{-\beta_1 t})] \\ &= \ln m_e! + m_e \ln \beta_1 - \beta_1 \sum_{i=1}^{m_e} t_i - m_e \ln(1 - e^{-\beta_1 t_e}). \end{aligned}$$

Setting the partial derivative of $\ln L(\boldsymbol{\beta})$ with respect to β_1 to 0, we have:

$$\frac{m_e}{\beta_1} - \sum_{i=1}^{m_e} t_i - \frac{m_e t_e e^{-\beta_1 t_e}}{1 - e^{-\beta_1 t_e}} = 0. \tag{4.24}$$

An estimation of β_1 can be obtained by solving Equation (4.24). Then an estimation of β_0 can be obtained by substituting the estimation of β_1 into the following equation:

$$\beta_0 = \frac{m_e}{1 - e^{-\beta_1 t_e}}. \tag{4.25}$$

Next, we consider the logarithmic model. By substituting $\lambda(t) = \dfrac{\beta_0\beta_1}{1 + \beta_1 t}$ and $\mu(t) = \beta_0 \ln(1 + \beta_1 t)$ into (4.23), we have:

$$\ln L(\beta_1 \mid m_e) = \ln m_e! + m_e \ln \beta_1 - \sum_{i=1}^{m_e} \ln(1 + \beta_1 t_i) - m_e \ln[\ln(1 + \beta_1 t_e)]. \tag{4.26}$$

Notice that this likelihood function $\ln L(\beta_1 \mid m_e)$ contains only one parameter β_1. The partial derivative of $\ln L(\beta_1 \mid m_e)$ with respect to β_1 is:

$$\frac{\partial \ln L(\beta_1 \mid m_e)}{\partial \beta_1} = \frac{m_e}{\beta_1} - \sum_{i=1}^{m_e} \frac{t_i}{1+\beta_1 t_i} - \frac{m_e t_e}{\ln(1+\beta_1 t_e)(1+\beta_1 t_e)}.$$

Setting the partial derivative to zero, and we get the conditional maximum likelihood equation for the logarithmic model:

$$\frac{m_e}{\beta_1} - \sum_{i=1}^{m_e} \frac{1}{1+\beta_1 t_i} - \frac{m_e\, t_e}{\ln(1+\beta_1 t_e)(1+\beta_1 t_e)} = 0. \tag{4.27}$$

Once we obtained an estimation for parameter β_1, the parameter β_0 can be estimated using the following equation:

$$\beta_0 = \frac{m_e}{\ln(1+\beta_1\, t_e)}. \tag{4.28}$$

Note that in the above calculus based approach, we set the partial derivatives of logarithmic likelihood functions to zero to obtain maximum likelihood equations. In order to verify that we do get maximum values of likelihood functions by solving these equations, we need to show that these logarithmic likelihood functions are in fact concave. This is equivalent to showing that the Hessian matrices of these functions with respect to β_0, β_1 are negative semidefinite. A simple Maple session can accomplish this task. We omit the details here.

4.3 Parameter Estimation by Curve Fitting

In this section, we discuss another commonly practiced method for estimating parameters in Software Reliability Growth Models (SRGM). The method has been used in many scientific computations for processing data from experiments.

In general, let $y_1, \ldots, y_{m_e}$ be m_e observations from an experiment. Let $f(\mathbf{x}, \boldsymbol{\theta})$ be the model where $\mathbf{x}$ is the variable vector and $\boldsymbol{\theta}$ is the parameter vector. Let

$$\epsilon_i = y_i - f(\mathbf{x}_i, \boldsymbol{\theta}), i = 1 \ldots m_e,$$

where $\mathbf{x}_i$ is the value of the variable for the ith experiment, and ϵ_i is the error of the ith experiment.

The least square method is to estimate the parameters by minimizing the sum of the squares of the errors.

$$\min_{\theta} \sum_{i=1}^{m_e} (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2.$$

Similarly, we can use

$$\min_{\theta} \sum_{i=1}^{m_e} \mid y_i - f(\mathbf{x}_i, \boldsymbol{\theta}) \mid . \tag{4.29}$$

This is called the *least absolute value* method.

We also define the relative least square method:

$$\min_{\theta} \sum_{i=1}^{m_e} \left(\frac{\mid y_i - f(\mathbf{x}_i, \boldsymbol{\theta}) \mid}{i} \right)^2 . \tag{4.30}$$

Khoshgoftaar et al [12] compares these parameter estimation techniques and conclude that relative least square method and relative least absolute value method appear to be superior. However, for models with only two parameters, we find any of above least error method produces satisfactory results.

For the logarithmic model and the exponential model, we consider the fitting functions as

$$\mu_1 = \beta_0 \ln(1 + \beta_1 t);$$

and

$$\mu_2 = \beta_0 (1 - e^{\beta_1 t})$$

respectively. Therefore, for the least square method applied to the logarithmic model and the exponential model, we have the following objective functions to minimize: (t_i here is a failure time given in the data file)

$$\sum_{i=1}^{m_e} [t_i - \beta_0 \ln(1 + \beta_1 t_i)]^2 ;$$

and

$$\sum_{i=1}^{m_e} [t_i - \beta_0 (1 - e^{\beta_1 t})]^2 .$$

If we use the least absolute value method, the following two objective functions are minimized in order to obtain estimations of parameters for the logarithmic model and the exponential model:

$$\sum_{i=1}^{m_e} \mid t_i - \beta_0 \ln(1 + \beta_1 t_i) \mid; \tag{4.31}$$

and

$$\sum_{i=1}^{m_e} \mid t_i - \beta_0(1 - e^{\beta_1 t}) \mid .$$

In our numerical experiment, we use the least absolute value method (4.31) to obtain estimations for parameters in the logarithmic model. See the next chapter for details.

4.4 Weighted Least Error Estimation

The least error estimation methods we discussed above give equal weight to each data point when a model is fitted. However, software testing practice indicates that the recent data points may influence the reliability projection more than earlier data points. Therefore, it is desirable to make the model fit better to the later data points, so that we can get better predictions for the future points in time.

Take a set of real numbers $w_1, \ldots, w_{m_e}$ in $[0, 1]$, with the property that

$$\sum_{i=1}^{m_e} w_i = m_e.$$

Define the weighted least square function as

$$\sum_{i=1}^{m_e} w_i (y_i - f(\mathbf{x}_i, \boldsymbol{\theta}))^2.$$

Similarly, we define the weighted least absolute value function, the weighted least square function, and the weighted relative absolute value function.

The weights w_i can be chosen in many different ways. Li and Malaiya in [15] conclude that a good way is to choose w_i as a linear function of of data points i. In other words, let

$$w_i = c + e \times i$$

where c is a constant in $[0, 1]$ and e can be determined from $\sum_{i=1}^{m_e} w_i = m_e$ and the chosen value of c.

4.5 Summary

In this chapter, we consider the problem of estimating the parameters in the exponential model and the logarithmic model. We present the method developed by Malaiya et al of estimating the parameters prior to the testing, without using the actual failure time. It is clear that such an estimating method cannot be very accurate. The values of a few quantities such as $D_{\min}, Q_x$, and $K_{\min}$ are given based on the experience of Malaiya et al. Practitioners may disagree with the values presented here. We present maximum likelihood estimation methods and least error methods for estimating the parameters in the exponential model and the logarithmic model using failure times. Two methods require optimize some objective functions. We present calculus based approaches for maximum likelihood estimation methods. But as we shall see in the next chapter, we can maximize the likelihood functions directly without using calculus. Various lease error methods, including weighted least error methods, are described. Examples will be presented in the next chapter.

Chapter 5

Numerical Methods

In the previous chapter, we discussed various parameter estimation techniques. Only very rarely do these techniques lead to analytic solutions. Most of techniques of estimating unknown parameters of a specified software reliability growth model require optimizing a particular objective function.

Musa et al point out in their book [23] that different techniques of parameter estimation are unlikely to have a substantial effect on predictive validity of the models. Therefore the choice of a particular parameter estimation technique depends largely on the numerical computation aspect of the technique.

We believe that a good method for estimating parameters in SRGM should be a method that can be applied to different models uniformly. The method should be independent of any particular structures of the model, and require minimum amount of modification to the optimization program code if the model is changed. It should also enjoy good stability and convergence properties, even for failure time data sets of large size and for models with many unknown parameters.

In the previous chapter, we have seen different methods of estimating parameters in software reliability growth models. We have seen that the maximum likelihood method and least error methods are derived from fundamentally different approaches. However, the idea of finding optimal values of some objective functions is the same. In the maximum likelihood method, it is the likelihood function we maximize; in the least error methods, we minimize the objective functions of errors in different norms. Calculus can be used to derive optimality

conditions and to find optimal solutions in both cases.

In maximum likelihood method, we take the gradient of the logarithmic likelihood function $\ln L$ with respect to the parameter vector $\boldsymbol{\theta}$ and derive the maximum likelihood equation from the equation

$$\nabla \ln L(\boldsymbol{\theta}) = 0.$$

The above is usually a system of *nonlinear* equations. Numerically solving such system of nonlinear equations is still a nontrivial task, especially when the parameter vector $\boldsymbol{\theta}$ is of large dimension.

There are many numerical algorithms of nonlinear optimization available for solving system of nonlinear equations. Most of them are calculus based.

Common methods for solving the least square problem are also calculus based. The objective function is the sum of the square of error terms. We set the gradient of the objective function to be the zero vector in order to find the minimum value of objective function. We face again the problem of solving a system of nonlinear equations.

Note that for some least error methods, such as least absolute value method and relative least absolute method, the objective functions are not differentiable. The calculus approach may become considerably more involved. Techniques in nonsmooth analysis have to be employed.

The common difficulty these methods share in actual numerical computation is that of selection of an initial guess. If the initial point is not good, then a local minimizer may be found or the algorithm may even fail to converge. Therefore, a global minimization method is desirable.

In this chapter, we present a general approach for estimating unknown parameters of software reliability growth models based on the data set of failure times. We describe the method and then present several numerical examples to demonstrate the applicability of our approach.

5.1 A global minimization method with descending mean algorithm

In the previous section, we have emphasized the importance of global optimization techniques in solving parameter estimation problems.

It is well known that Newton's method for nonlinear minimization is efficient once an appropriate initial point is chosen. However, it is not always a trivial matter to provide a suitable initial point. Incorrect initial points could lead to local minimum solutions. In our case, this will lead to incorrect parameter estimation for the model.

We present a general global optimization method that generates a suitable initial point automatically. Modifying the recommended optimization procedure presented in [23] by using the Descending Mean algorithm instead of Nelder-Mead direct search algorithm, our general method for global minimization can be simply stated as:

Step 1 : Generate an initial solution using Descending Mean method;

Step 2 : Feed the initial solution generated above to Newton's method;

Step 3 : If Newton's method converges, stop, output minimum solution else goto Step 1, until the limit of iteration is exceeded.

If the derivative information is not available, Newton's method cannot be applied directly. so Step 2 above cannot proceed. If this is the case, we can just use Descending Mean algorithm alone to complete the computation.

The heart of the above method is the Descending Mean algorithm. This is a direct search method with a clear objective: *descreasing the average function values obtained in each search iteration.* The algorithm is conceptually simple and can be easily implemented for parallel processing if multi-processor computers are available. The algorithm is inspired by the implementation of integral global optimization technique [35, 34, 27]. Note that maximizing an objective function f is equivalent to minimizing $-f$, we need to consider minimization only.

Let $f : \mathcal{R}^n \to \mathcal{R}$ be a continuous objective function to be minimized. Let $\mathbf{x}$ be the variable vector, $\mathbf{a} = (a_1, \ldots, a_n)$ and $\mathbf{b} = (b_1, \ldots, b_n)$ are constant vectors. Assume that the variable vector $\mathbf{x}$ constrained by

$$\mathbf{a} \leq \mathbf{x} \leq \mathbf{b}. \tag{5.1}$$

More specifically, we have the following *box constraints*:

$$a_i \leq x_i \leq b_i \quad i = 1, \ldots, n. \tag{5.2}$$

We want to find $\mathbf{x_0}$ such that $f(\mathbf{x_0}) \leq f(\mathbf{x})$ for all $\mathbf{x}$ that satisfy (5.1).

Let $D = [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n]$ be the set of all $\mathbf{x} = (x_1, \ldots, x_n)$ satisfying the constraints (5.2). D is often called the *constraint set.* The method can be described as follows:

Step 1 : Take D as our initial search domain. We randomly take k points $d_i, i = 1 \ldots k$, from D, evaluate the function at these k points, and find the smallest l function values. Let the corresponding points be $e_1, e_2, \ldots, e_l$. We compute the approximate average function value m over D by calculating

$$m = \frac{\sum_{i=1}^{l} f(e_i)}{l}.$$

Step 2 : Sort the points $e_1, e_2, \ldots, e_l$ according to their function values. Make a new search domain D that is the smallest rectangle box in D that contains $e_1, e_2, \ldots, e_l$.

Step 3 : In the new search domain D_1, we take k points $d_i, i = 1 \ldots k$, from D_1, evaluate the function at these k points, and find the smallest l function values that are less than m computed in Step 1. If we cannot find l points with such property, then take another k points $d_i, i = 1 \ldots k$, repeat the process until this is done or until a prespecified maximum number of iterations is exceeded. Let the corresponding points be $e_1, e_2, \ldots, e_l$. We compute the

approximate average function value m over D by calculating

$$m_1 = \frac{\sum_{i=1}^{l} f(e_i)}{l}.$$

Clearly, $m_1 < m$, so the name *descending mean.*

Step 4 : Continue from Step 2 to form a new search domain D The process stops when the difference between the current mean value m_1 and the previous mean value m is less than a specified tolerance, or the maximum number of iterations is exceeded. In the case of successful run, the smallest function value $f(e_{i_0})$ for some $i_0 \in \{1, 2, \ldots, l\}$ is returned by the program as the minimum function value and e_{i_0} is returned as minimum point.

The convergence property of the algorithm is intuitively clear. Every iteration guarantees the decrease of the mean of the function values over D. When all the function values $f(e_i)$ are the same as the global minimum function value, the decreasing of the mean in new iterations becomes impossible. In other words, the mean in the current iteration is the same as that in the previous iteration. The program stops and outputs the global minimum function value and the minimizer. For a more rigorous argument, the reader is referred to [35].

5.2 Parameter Estimation with Descending Mean Algorithm

In this section, we illustrate the application of the global minimization method described in the previous section to solving parameter estimation problems. We apply the method to the objective function established from the maximum likelihood method and to the objective function established from least errors methods.

The maximum likelihood method has been successfully applied in the book by Musa et al [23] to solve several parameter estimation problems. In our computation experiments, we find that it is difficult to solve Equation (4.17) or Equation

(4.27) when the data set is large, even the equation contains only one unknown variable β_1.

Instead, we apply our global minimization method to $-\ln L(\boldsymbol{\beta})$ directly. For the exponential model, the objective function is simply Equation (4.15):

$$-\sum_{i=1}^{m_e}[\ln\beta_0 - \ln\beta_1 - \beta_1 t_i] + \beta_0(1 - e^{-\beta_1 t_e}).$$

For the logarithmic model, the objective function is simply (4.18):

$$m_e \ln\beta_0 + m_e \ln\beta_1 - \sum_{i=1}^{m_e}\ln(1+\beta_1 t_i) - \beta_0 \ln(1+\beta_1 t_e).$$

Similarly, we can write out explicitly the objective functions for the exponential model and logarithmic model from conditional logarithmic likelihood functions.

With our *descending mean algorithm* described in the previous section, we can estimate parameters in logarithmic models using different norms, and be assured to have solved global minimization problems related to the least error approach, without need of using derivative information of the models. In other words, it saves a great deal of precious programmers' time. The price to pay for all these advantages is that in the course of solving global minimization problems, we may require fair number of function evaluations. However, with the rapid advance in computer hardware, the availability of fast CPUs, this is indeed a very small price. In our computation experiments, presented in the next section, we run our program on a Pentium II machine. It requires only a few seconds to find optimal parameters for the exponential model and the logarithmic model for data set `Musa5.fds`. The data set `Musa5.fds` contains 831 failure times and is the largest failure time data set we have been able to acquire.

We would like to emphasize that it is our belief that using the maximum likelihood method and the least error approach to estimate parameters for SRGM are in general stable, reliable and efficient. The methods are independent of any specific models. Given a new model and the related data set, the methods can be applied immediately without substantially changing the program code.

5.3 Numerical Tests

In this section, we present several numerical examples to illustrate the flexibility and efficiency of our approach for estimating the parameters in software reliability growth models.

Most examples presented here are taken from the data sets provided by the integrated software reliability tool ROBUST [16] developed at Colorado State University. Note that these data sets are well-known in the reliability literature. Most of them are collected by Musa and are of high quality and accuracy. For each data set, we estimate parameters β_0 and β_1 of the logarithmic model $\beta_0 \ln(1+\beta_1 t)$ by minimizing the objective function of least absolute error (4.31); we estimate that of the exponential model $\beta_0(1 - e^{\beta_1 t})$ by optimizing the objective function generated by the unconditional logarithmic likelihood function (4.15).

We present the results of our estimations in the following format: the values of parameters for each model are reported. The objective function values corresponding to the parameters are also reported for future comparison. Note that these values are not relevant to the applications of the reliability models.

For first data set **Musa1**, we list the failure times in a table. In the table, the kth entry in the column with the title i represents the kth failure, the corresponding entry in the column with the title t_i is the kth failure time. In our illustrations, the horizontal-axis represents the failure time, in CPU seconds; the vertical-axis represents the number of failures occurred. We use the notations `mu1(t)`, `mu2(t)` to denote the expected number of failures experienced by time t for the logarithmic and the exponential models

$$\begin{aligned} \mu_1 &= \beta_0 \ln(1 + \beta_1 t); \\ \mu_2 &= \beta_0(1 - e^{\beta_1 t}), \end{aligned}$$

respectively. These curves are plotted along with the failure data to show the fitness of the models.

With the exception of **Musa1**, we present only our computation results and plot for each failure data set, but not the data tables, in order to keep the thesis

to a reasonable size. The actual data sets can all be found in [16].

Table 5.1: Data Set for Musa 1

Musa 1							
i	t_i	i	t_i	i	t_i	i	t_i
1	3	2	33	3	146	4	227
5	342	6	351	7	353	8	444
9	556	10	571	11	709	12	759
13	836	14	860	15	968	16	1056
17	1726	18	1846	19	1872	20	1986
21	2311	22	2366	23	2608	24	2676
25	3098	26	3278	27	3288	28	4434
29	5034	30	5049	31	5085	32	5089
33	5089	34	5097	35	5324	36	5389
37	5565	38	5623	39	6080	40	6380
41	6477	42	6740	43	7192	44	7447
45	7644	46	7837	47	7843	48	7922
49	8738	50	10089	51	10237	52	10258
53	10491	54	10625	55	10982	56	11175
57	11411	58	11442	59	11811	60	12559
61	12559	62	12791	63	13121	64	13486
65	14708	66	15251	67	15261	68	15277
69	15806	70	16185	71	16229	72	16358
73	17168	74	17458	75	17758	76	18287
77	18568	78	18728	79	19556	80	20567
81	21012	82	21308	83	23063	84	24127
85	25910	86	26770	87	27753	88	28460
89	28493	90	29361	91	30085	92	32408
93	35338	94	36799	95	37642	96	37654
97	37915	98	39715	99	40580	100	42015
101	42045	102	42188	103	42296	104	42296
105	45406	106	46653	107	47596	108	48296
109	49171	110	49416	111	50145	112	52042
113	52489	114	52875	115	53321	116	53443
117	54433	118	55381	119	56463	120	56485
121	56560	122	57042	123	62551	124	62651
125	62661	126	63732	127	64103	128	64893
129	71043	130	74364	131	75409	133	81542
134	82702	135	84566				

Musa1:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 44.5184 \quad \beta_1 = 0.000231005$$

The least absolute value = 299.461

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 144.849 \quad \beta_1 = 3.30556e - 5$$

The maximum likelihood function value = -973.725.

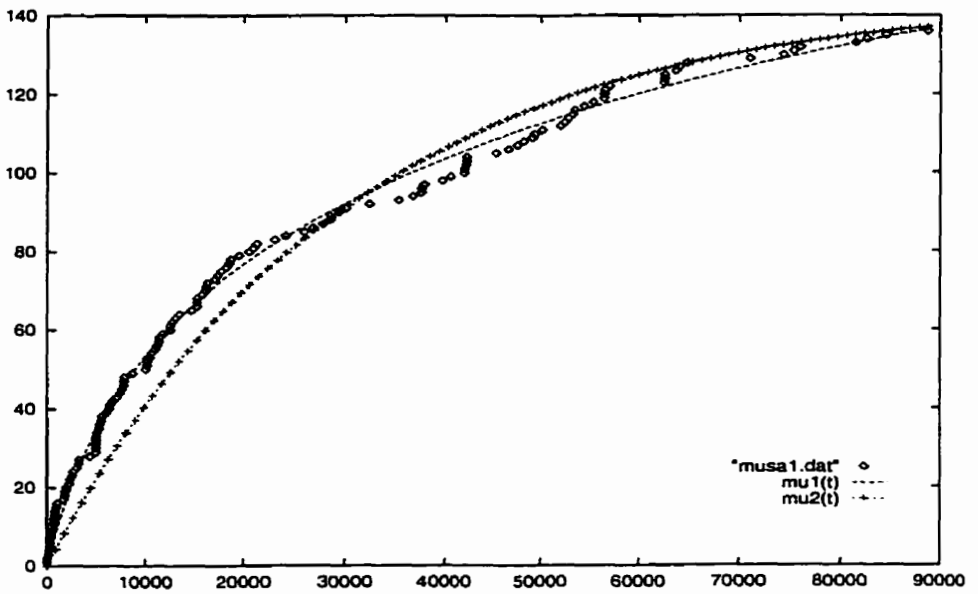


Figure 5.1: Plot of curves fitting for Musa1

Before we present more numerical tests, we would like to illustrate several results presented in the previous chapters using the concrete failure time data **Musa1** for a real time command and control system T_1. The data is collected under the supervision of Musa and is of high quality.

From our computation, we have obtained the exponential model for the data set **Musa1** as follows:

$$\begin{aligned} \mu(t) &= 144.849(1 - e^{-0.0000330556\,t}); \\ \lambda(t) &= 144.849 \times 0.0000330556\, e^{-0.0000330556\,t} \\ &= 0.004788070604\, e^{-0.0000330556\,t}. \end{aligned}$$

From the fact that β_0 in the exponential model represents the total failures that would be experienced in infinite time, we see that the system T_1 represented by the data set **Musa1** would experience 145 failures in its entire operation phase. The par-fault hazard rate in the exponential model is a constant. In this case, it is 0.0000330556.

Now, we are able to answer the questions regarding the reliability of the system T_1. In the data set **Musa1**, there are total of 135 failures and the last failure occurred when the system has been operating for 84566 CPU seconds. First, we can answer the question that after the 135th failure, what is the probability of failure-free operation for at least 1000 CPU seconds. According to Proposition 3.3.5, we see that the probability is computed by the formula:

$$P = e^{-[\mu(t_{i-1}+t_i')-\mu(t_{i-1})]}, \quad i = 1, 2, \ldots$$

If we take $t_{i-1} = 135$ and $t_i' = 1000$, using

$$\mu(t) = 144.849(1 - e^{-0.0000330556\,t}),$$

we get $P = 0.7499665301$. However, if we take $t_i' = 10000$, then $P = 0.08284581909$, a much smaller probability. On the other hand, if we take $t_i' = 100$, then $P = 0.9712196754$. It is clear that after the 135th failure, the system T_1 will operate failure-free for at least 100 CPU seconds. It is likely that is will operate

failure-free for 1000 CPU seconds and is very likely to experience some failures within 10000 CPU seconds.

According to Proposition 3.3.4, the probability that the time to the 136th failure is at most t, given that 135 failures has been experienced by the time 84566 is given by:

$$P = 1 - e^{\mu(t)-\mu(84566)}.$$

If we take $t = 100000$, then $P = 0.9708736058$. On the other hand, if we take $t = 84576$, that is 10 CPU seconds after the 135th failure, then $P = 0.0010817145$. In other words, the probability that the 136th failure occurs 10 CPU seconds after the 135th failure is very small.

Many quantities are be similarly computed. Also, if we use the logarithmic model:

$$\mu(t) = 44.5184 \ln(1 + 0.000231005\, t),$$

We can answer the similar questions using the same propositions in Chapter 3. For the system T_1, two models give similar results.

From this example, we see that the software reliability growth models do provide some useful information about the reliability of software products.

Now, we continue our numerical experiments. As we can see from the plots presented below, our optimization method provides accurate estimations for the parameters in the exponential model and the logarithmic model.

Musa2:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 16.4477 \quad \beta_1 = 0.000234217$$

The least absolute value = 53.8822

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 57.1298 \quad \beta_1 = 2.67171e-5$$

The maximum likelihood function value = -449.094

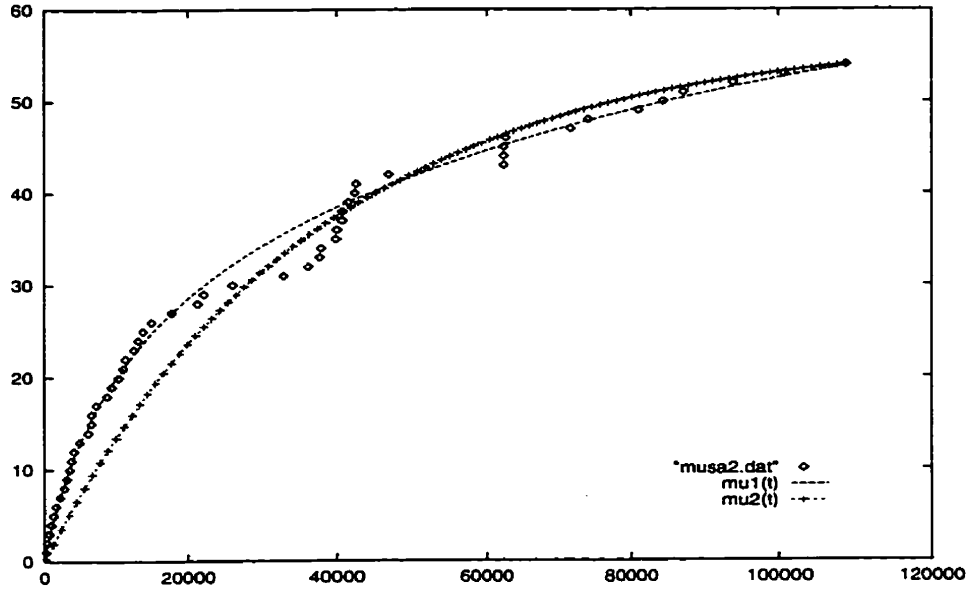


Figure 5.2: Plot of curves fitting for Musa2

Musa3:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 8.09241 \qquad \beta_1 = 0.00151402$$

The least absolute value = 449.094

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 38.6835 \qquad \beta_1 = 5.99138e - 5$$

The maximum likelihood function value = -303.795

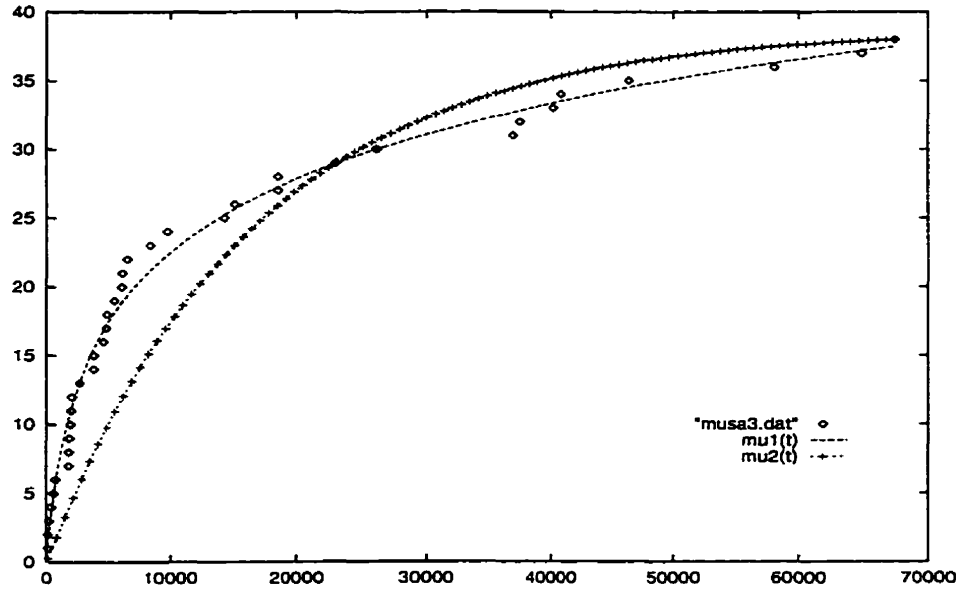


Figure 5.3: Plot of curves fitting for Musa3

Musa4:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 23.4261 \qquad \beta_1 = 0.000365225$$

The least absolute value = 114.82

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 52.1754 \qquad \beta_1 = 0.000109007$$

The maximum likelihood function value = -377.975

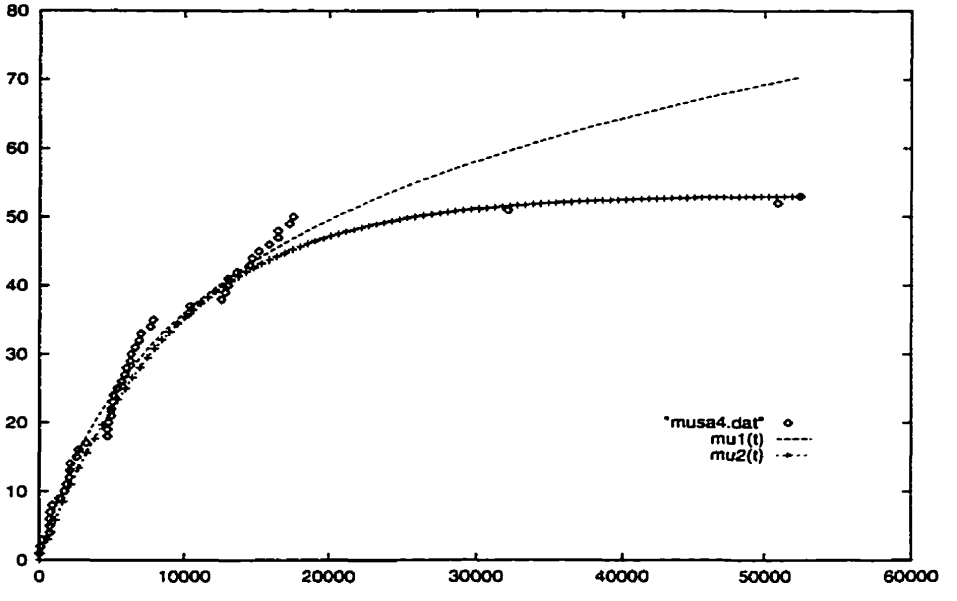


Figure 5.4: Plot of curves fitting for Musa4

Musa5:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 1117.8 \quad \beta_1 = 0.000509351$$

The least absolute value = 15480.5

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 1777.08 \quad \beta_1 = 0.000297626$$

The maximum likelihood function value = -1594.89

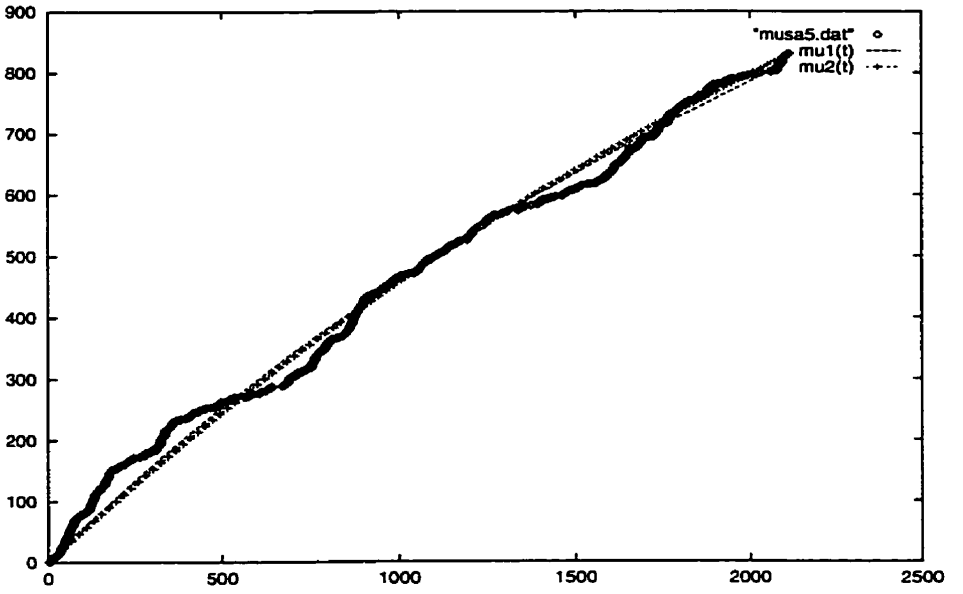


Figure 5.5: Plot of curves fitting for Musa5

Musa6:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 46.8598 \quad \beta_1 = 0.000735338$$

The least absolute value = 15480.5

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 95.9606 \quad \beta_1 = 0.000280974$$

The maximum likelihood function value = -376.934

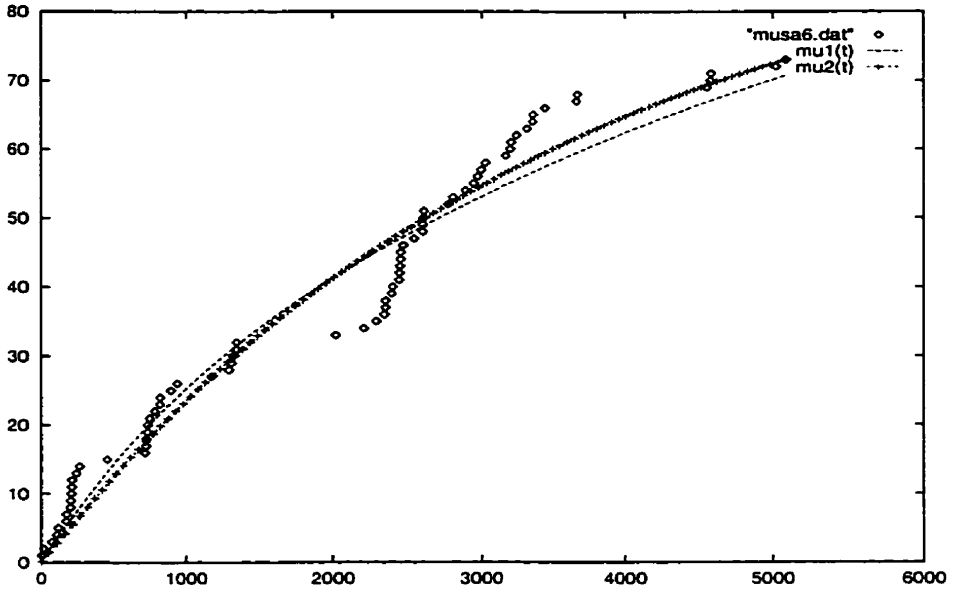


Figure 5.6: Plot of curves fitting for Musa6

Musa16:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 14.1012 \qquad \beta_1 = 0.0401427$$

The least absolute value = 76.056

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 43.1976 \qquad \beta_1 = 0.00690628$$

The maximum likelihood function value = -125.046

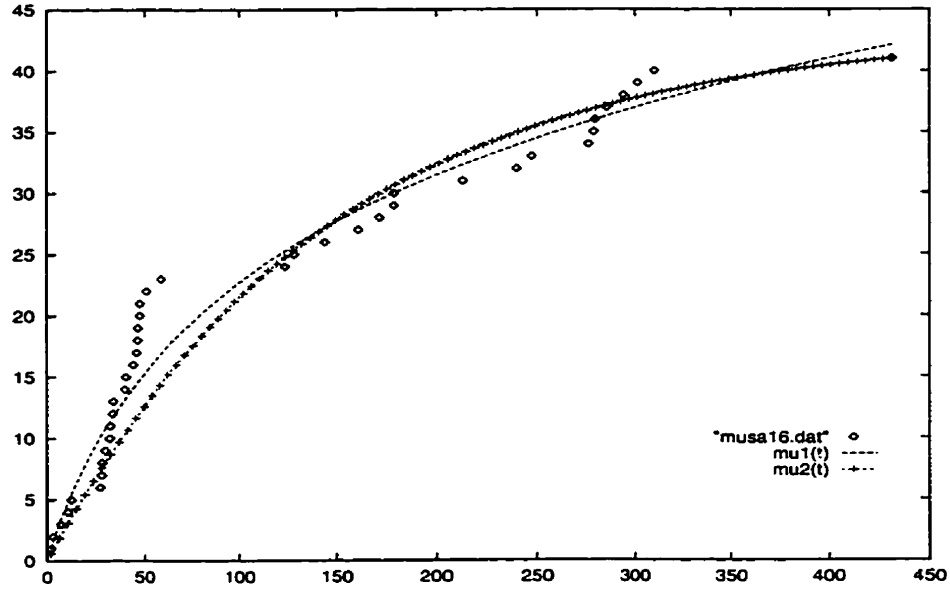


Figure 5.7: Plot of curves fitting for Musa16

Musa19:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 28.181 \quad \beta_1 = 0.0165376$$

The least absolute value = 59.2173

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 41.4454 \quad \beta_1 = 0.0106433$$

The maximum likelihood function value = -98.5311

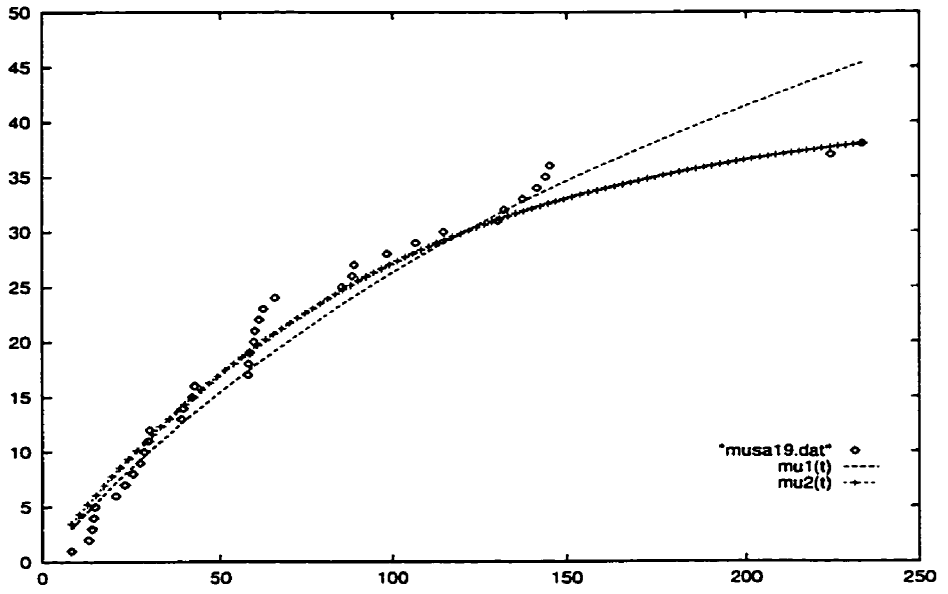


Figure 5.8: Plot of curves fitting for Musa19

Musa18:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 62.1423 \quad \beta_1 = 0.00928926$$

The least absolute value = 825.6881450

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 163.301 \quad \beta_1 = 0.00216108$$

The maximum likelihood function value = -493.88

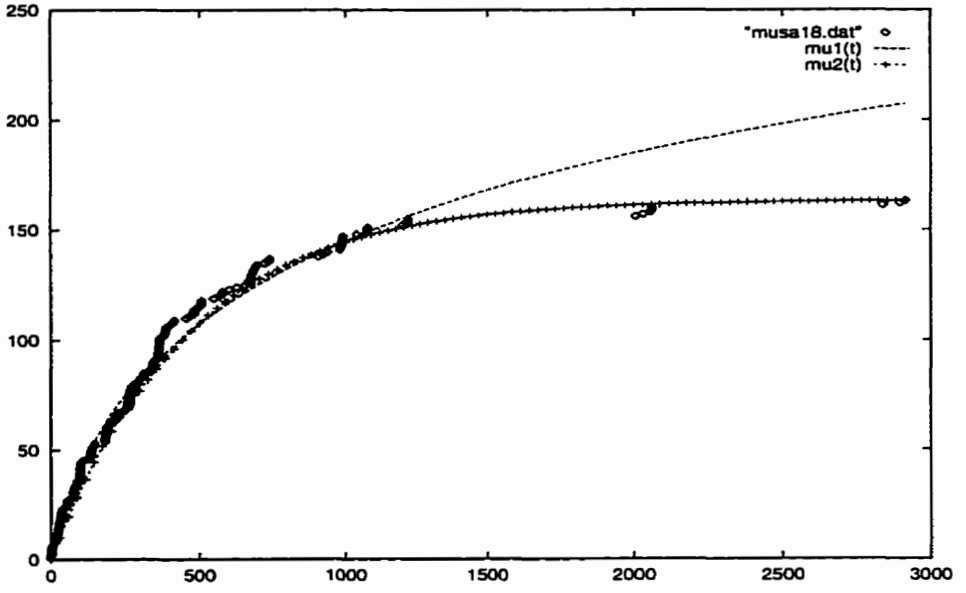


Figure 5.9: Plot of curves fitting for Musa18

Musa40:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 19.6043 \quad \beta_1 = 0.735934$$

The least absolute value = 589.5727346

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 102.853 \quad \beta_1 = 0.0205206$$

The maximum likelihood function value = -119.112

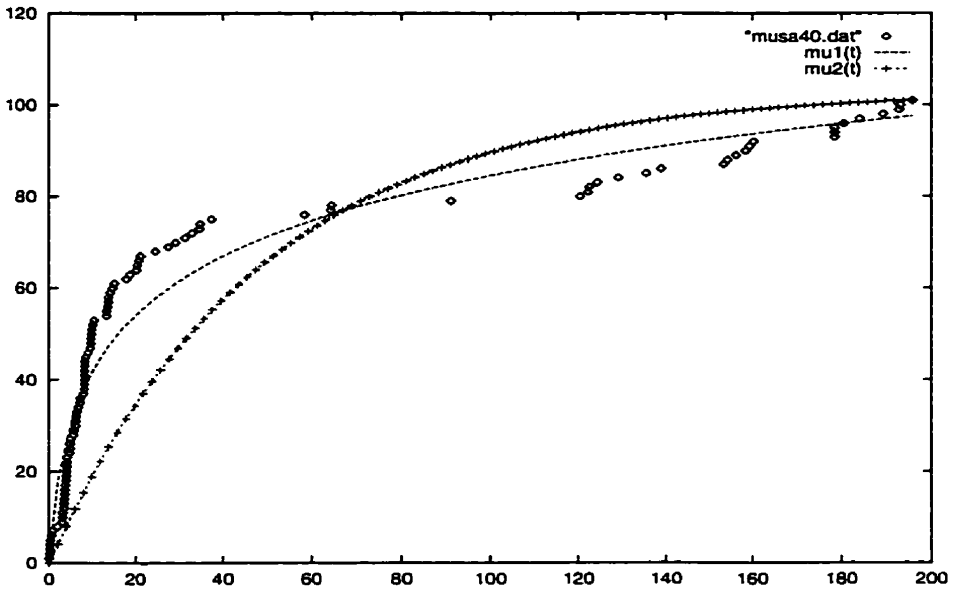


Figure 5.10: Plot of curves fitting for Musa40

CSR1:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 103.404 \quad \beta_1 = 0.0476026$$

The least absolute value = 6786.904578

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 404.057 \quad \beta_1 = 0.00421859$$

The maximum likelihood function value = -566.6

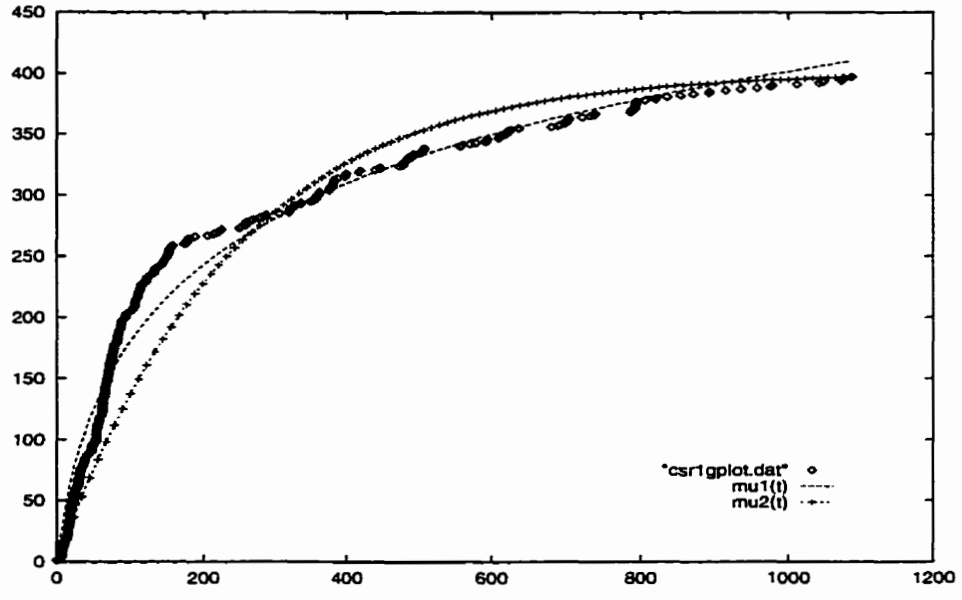


Figure 5.11: Plot of curves fitting for CSR1

CSR2:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 39.2673 \quad \beta_1 = 0.0297363$$

The least absolute value = 432.1091459

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 133.135 \quad \beta_1 = 0.00389929$$

The maximum likelihood function value = -328.217

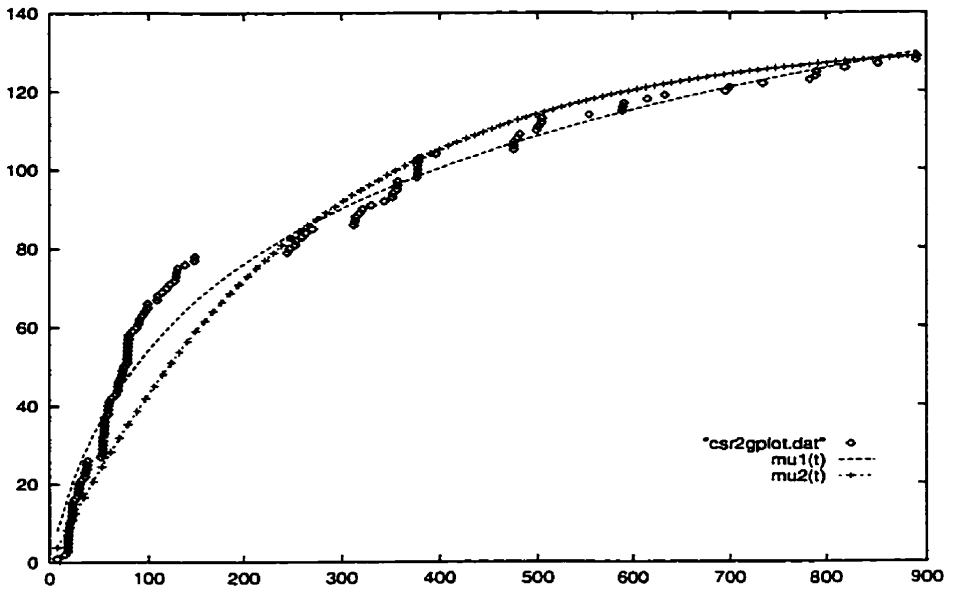


Figure 5.12: Plot of curves fitting for CSR2

NSingpur:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 19.6056 \quad \beta_1 = 0.7362$$

The least absolute value = 589.8601183

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 102.855 \quad \beta_1 = 0.0205304$$

The maximum likelihood function value = -119.059

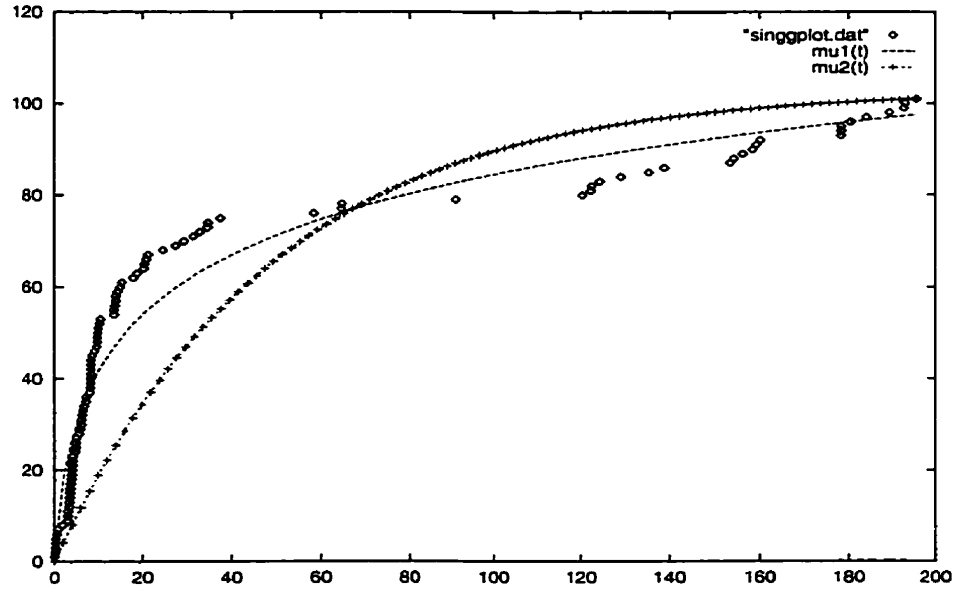


Figure 5.13: Plot of curves fitting for NSingpur

Usbar:

Estimation by Least Absolute Value Method for the Logarithmic Model

$$\beta_0 = 103.404 \quad \beta_1 = 0.0476027$$

The least absolute value = 6786.900404

Estimation by Maximum Likelihood Method for the Exponential Model

$$\beta_0 = 401.057 \quad \beta_1 = 0.00421859$$

The maximum likelihood function value = -566.6

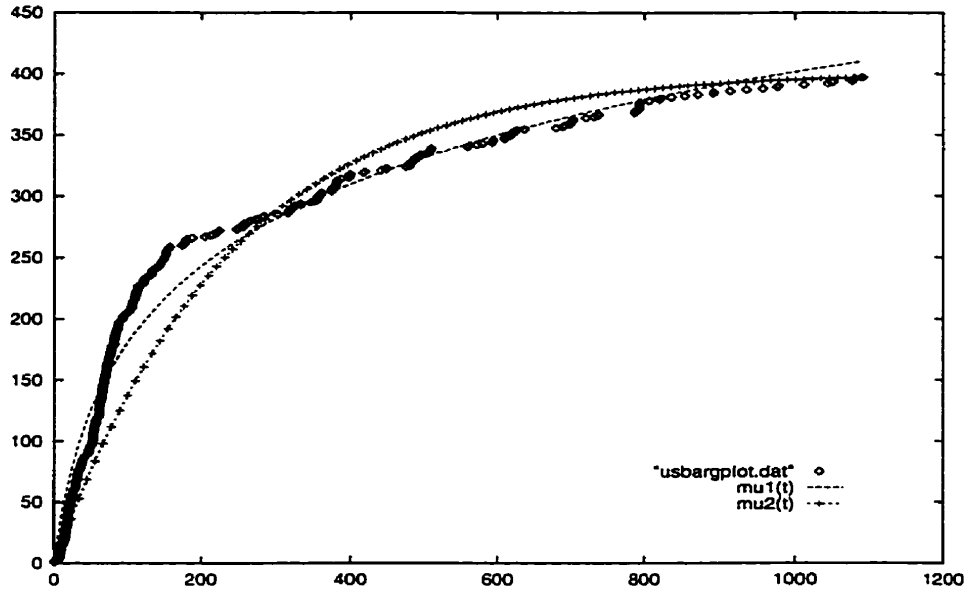


Figure 5.14: Plot of curves fitting for Usbar

Chapter 6

Conclusion

Quality of a software product is an important factor for the success of the software product. Testing costs the software industry 40% to 50% of the total product development life-cycle expenses on software testing to ensure the quality of software.

Software reliability is the probability of failure-free operations for a specified duration under a given usage environment. One fundamental assumption in software reliability theory is the existence of faults. The exposure of such faults is by activating software via testing and usage. A software reliability growth model provides a systematic way of assessing and predicting software reliability.

In this thesis, we review several well known software reliability growth models. Based on simple probability theory and Poisson process properties, we present derivations of some general Poisson type models as well as some important quantities associated with the models. These quantities allow managers and testers to answer questions regarding the reliability of the software product under test. In particular, two models widely adopted by researchers and testing practitioners, the Musa basic execution model, (also known as the exponential model), and the Musa logarithmic model, are studied in detail.

We emphasize that software reliability growth models should only be used as a guideline. Using the models, we can see the general trend of improvement of software reliability through the testing and debugging process. We should not take the numerical values generated by these models as the absolute measure for

the quality of software products and make important decisions about the quality of the software product based solely on the numerical output of the software reliability growth models. This is because of the human factor involved in the collection of failure time data for the input. Since the data collection process is imperfect and often biased, we should expect the output from the models to be only as good as the input. Also, some software reliability growth models are established in a simplified and idealized environment. In the real world, we may not work in such an idealized environment. Therefore, when we use a software reliability model in practice, we need to check the assumptions associated with the model. If the model assumptions are close to the operation profile the software product is using, then the output from the model is more meaningful than the output from those models that are not close.

In order to apply a software reliability growth model to a specific software product, we need to estimate the parameters of the model for this product. The estimations are usually based on failure time data. Statistical methods for estimating model parameters are well developed. We present several commonly used methods of parameter estimation. Specifically, maximum likelihood methods and curve fitting using least error methods are presented for two fundamental models: the exponential and the logarithmic models. The key idea of these methods is to optimize some appropriate objective function. In order to overcome some difficulties faced by traditional calculus based optimization techniques, we introduce the descending mean algorithm. We apply a global minimization method based on the descending mean algorithm to parameter estimation problems. Our method has less stringent requirements on the objective functions than those calculus based methods. For example, when the derivative information of the objective function is not available, such as in the case of least absolute value functions, unlike most calculus based methods, we can still perform optimization procedures to find appropriate parameter estimations. The descending mean algorithm is conceptually simple and can be easily implemented for parallel processing in a multi-processor computer. The disadvantage of this simple method is the large

number of function evaluations, compared with the traditional methods based on derivative information. However, with rapid development of computer hardware, such a disadvantage become less and less significant.

We apply the method on a set of well-known, high quality, failure time data to estimate parameters in the exponential model and the logarithmic model. The numerical experiments are performed on a Pentium II personal computer running Linux. The computation on the largest data file obtainable takes only a few seconds. The results of our numerical experiments show that the method is efficient and accurate for these models.

There are a lot more can be said about software testing and software reliability growth models. We would like to suggest that the following topics can be studied further in the future research:

- More work should be devoted to the topic of validating software reliability growth models. During the testing process, we can collect failure time data. We can use different reliability models. It is important to develop some methods that will allow us to validate if the model we choose represent the software product best. Validations should include the model assumptions, the range of data, the operational profile and test cost.

- Intuition and empirical evidence suggests that code coverage is related to reliability. It is desirable to establish some software reliability growth models that directly associates with code coverage analysis. Malaiya et al have established such models. More studies and experiments on such models will be useful.

- In this thesis, we apply global optimization techniques to point parameter estimations. It would be interesting to know that if the algorithms developed here can be applied to interval estimation of the parameters in software reliability growth models.

Bibliography

[1] M. J. Bassman, F. McGarry, and Pajerski R. Software measurement guidebook. Technical Report SEL-94-002, NASA, July 1994.

[2] Boris Beizer. *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Company, Inc., New York, NY, 1984.

[3] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, Inc., New York, NY, 1990.

[4] IEEE Standards Board. *IEEE Standard Classification for Software Anomalies.* The Institute of Electrical and Electronics Engineers, Inc, New York, NY, 1992.

[5] Hetzel W. C. *The Complete Guide to Software Testing.* QED Information Sciences Inc., 1988.

[6] M. H. Chen. *Tools and Techniques for Testing Based Software Reliability Estimation.* PhD thesis, Purdue University, W. Lafayette, IN, 1992.

[7] A. Endres. An analysis of errors and their causes in system programs. In *1975 International Conference on Software Reliability*, pages 327–336, April 1975.

[8] W. Farr. *Software Reliability Modeling Survey*, pages 71–117. McGraw-Hill, 1996. Ed. by M. R. Lyu.

[9] L. Hatton. N-version design versus one good design. *IEEE Software*, Nov./Dec.:71–76, 1997.

[10] Syed A. Hossain and Ram C. Dahiya. Estimating the parameters of a non-homogeneous poisson-process model for software reliability. *IEEE Transactions on Reliability*, 42(4):604–612, 1993.

[11] Richard M. Karcich and Robert Skibbe. On software reliability and code coverage. *IEEE*, 8:297–308, 1996.

[12] Taghi M. Khoshgoftaar, Bibhuti B. Bhattacharyya, and Gary D. Richardson. Predicting software errors, during development, using nonliear regression models: A comparative study. *IEEE Transactions on Reliability*, 41(3):390–395, 1992.

[13] Edward Kit. *Software Testing in the Real World.* Addison-Wesley, Harlow, England, 1995.

[14] Naixin Li and Yashwant K. Malaiya. Enhancing accuracy of software reliability prediction. In *4th International Symposium on Software Reliability Engineering*, pages 71–79, Denver, November 1993.

[15] Naixin Li and Yashwant K. Malaiya. Fault exposure ratio estimation and applications. Technical Report CS-96-130, Colorado State University, 1996.

[16] Yashwant Malaiya, Jason Denton, and Naixin Li. Robust, an integrated software reliability tool. Technical Report User's Guide, Colorado State University, 1998.

[17] Yashwant K. Malaiya, Jim Bieman, and Naixin Li. Software test coverage and reliability. *Submitted to IEEE Trans. Software Reliability*, 1995.

[18] Yashwant K. Malaiya and Jason Denton. What do the software reliability growth model parameters represent? In *Proc. 8th International Symposium On Software Reliability Engineering, 1997, Albuquerque, NM.*, IEEE Conference Proceeding, pages 124–135, 1997.

[19] Yashwant K. Malaiya and Jason Denton. Estimating defect density using test coverage. Technical Report 98-104, Colorado State University, 1998.

[20] Yashwant K. Malaiya, N. Karunanithi, and P. Verma. Predictability of software reliability models. *IEEE Trans. Reliability*, pages 539–546, December 1992.

[21] Yashwant K. Malaiya, A. von Mayrhauser, and P. K. Srimani. The nature of fault exposure ratio. In *Proc. International Symposium on Software Reliability Engineering*, pages 23–32, 1992.

[22] Larry Morell. A theory of fault-based testing. *IEEE Transaction on Software Eng.*, 16(8):844–857, 1990.

[23] John Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability:Measurement, Prediction, Application.* McGraw-Hill, Inc., New York, NY, 1987.

[24] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., 1979.

[25] Tausworthe R. Experience more reliable than theory. *IEEE Software*, 12(2):110–112, 1995.

[26] Marc Roper. A comparison of some structural testing strategies. *IEEE Transaction on Software Eng.*, 14(6):868–874, 1988.

[27] Shuzhong Shi, Quan Zheng, and Deming Zhuang. Discontinuous robust mappings are approximatable. *Transaction of America Math. Society*, 26:33–50, 1995.

[28] M. L. Shooman and M. I. Bolsky. Types, distribution, and test and correction times for programming errors. In *1975 International Conference on Software Reliability*, pages 347–362, April 1975.

[29] J. Slonim, M. Bauer, and J. Y. Ye. An empirical study of software testing and reliability in an industrial setting. In *Proc. of the Twentiety Annual Software Engineering Workshop*, 1996. To appear.

[30] J. Slonim, M. Bauer, and J. Y. Ye. Software reliability assurance in early development phase: A case study in an industrial setting. In *Proc. IEEE 1996 Aerospace Application Conference*, volume 4 of *IEEE Conference Proceeding*, pages 279–295, 1996.

[31] J. Slonim, M. Bauer, and J. Y. Ye. Structural measurement of functional testing: A case study in an industrial setting. In *Proc. Ninth Internation Software Quality Week Conference*, 1996. To appear.

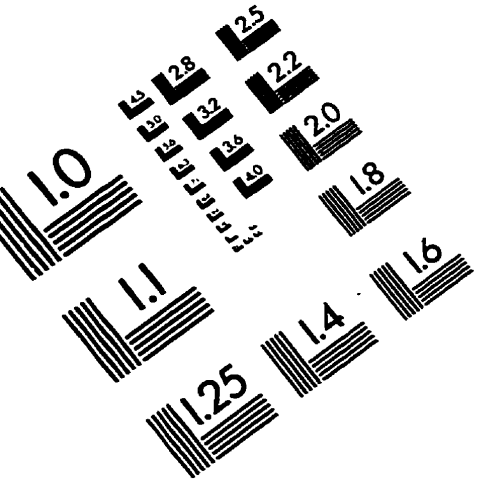
[32] Jillian Ye, David Godwin, and Colin Mackenzie. What is uncovered by coverage testing? Technical Report TR-74. 158, IBM Canada Ltd. Laboratory, 1996.

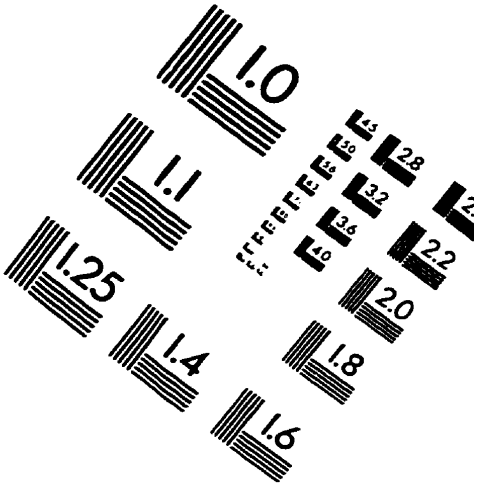
[33] Yinghong Jillian Ye. An empirical study of testing large-scale commercial software. Master's thesis, The University of Western Ontario, London, Ontario, Canada, March 1996.

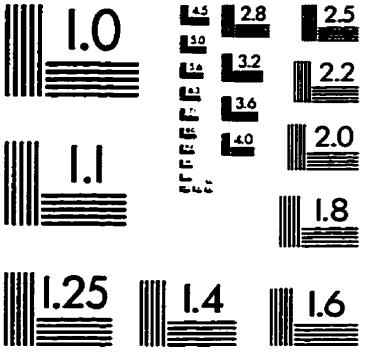
[34] Quan Zheng and Deming Zhuang. On global optimization and monte carlo implementations. In *Proc. on Scientific and Engineering Computing*, pages 262–266, Beijing, China, 1994. National Defense Industry Press.

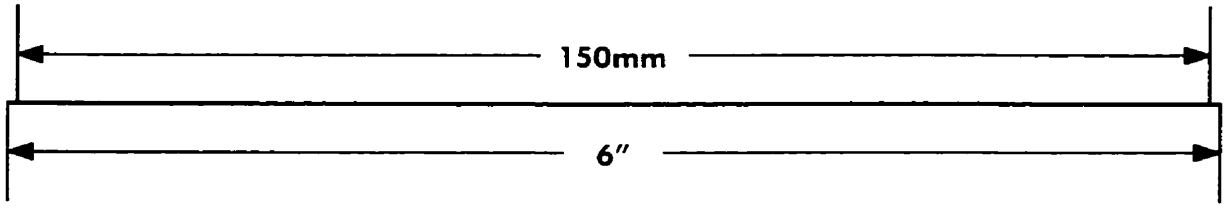
[35] Quan Zheng and Deming Zhuang. Integral global minimization: Algorithms, implementations and numerical tests. *Journal Of Global Optimization*, 7(4):421–454, 1995.

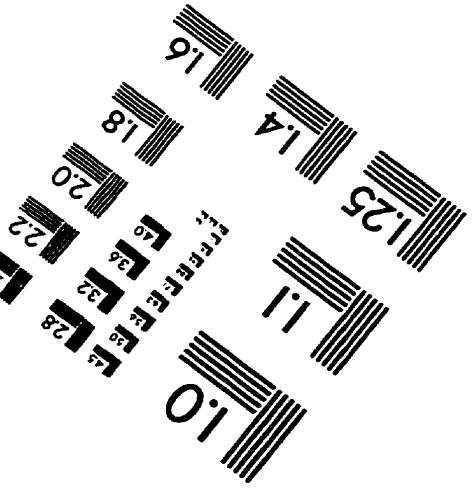
IMAGE EVALUATION
TEST TARGET (QA-3)

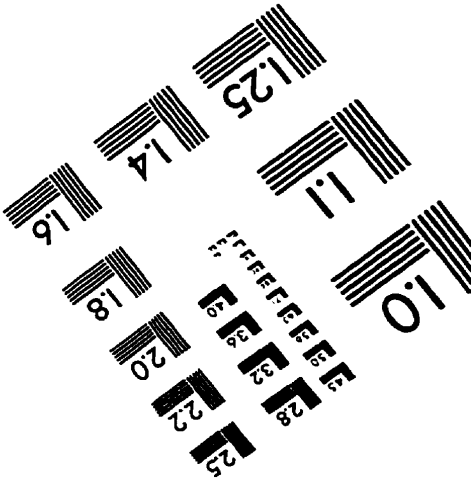












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989