Spring 2001

A Software Reliability Model Combining Representative and Directed Testing

Brian Michael Mitchell
Old Dominion University

A SOFTWARE RELIABILITY MODEL COMBINING REPRESENTATIVE AND DIRECTED TESTING

by

Brian Michael Mitchell
B.S. May 1991, Hampden-Sydney College
M.S. August 1993, Old Dominion University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fullfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
May 2001

Approved by:

Steven Zeil (Director)

Larry Wilson

J. Christian Wild

C. Michael Overstreet

Larry Lee

ABSTRACT

A SOFTWARE RELIABILITY MODEL COMBINING REPRESENTATIVE AND DIRECTED TESTING

Brian Michael Mitchell
Old Dominion University, 2001
Director: Dr. Steven Zeil

Traditionally, software reliability models have required that failure data be gathered using only representative testing methods. Over time, however, representative testing becomes inherently less effective as a means of improving the actual quality of the software under test. Additionally, the use of failure data based on observations made during representative testing has been criticized because of the statistical noise inherent in this type of data. In this dissertation, a testing method is proposed to make reliability testing more efficient and accurate. Representative testing is used early, when the rate of fault revelation is high. Directed testing is used later in testing to take advantage of its faster rate of fault detection. To make use of the test data from this mixed method approach to testing, a software reliability model is developed that permits reliability estimates to be made regardless of the testing method used to gather failure data. The key to being able to combine data from both representative testing and directed testing is shifting the random variable used by the model from observed interfailure times to a post-mortem analysis of the debugged faults and using order statistics to combine the observed failure rates of faults no matter how those faults were detected. This shift from interfailure times removes the statistical noise associated with the use of this measure, which should allow models to provide more accurate estimates and predictions. Several experiments were conducted during the course of this research. The results from these experiments show that using the mixed method approach to testing with the new model provides reliability estimates that are at

least as good as estimates from existing models under representative testing, while requiring fewer test cases. The results of this work also show that the high level of noise present in failure data based on observed failure times makes it very difficult for models that use this type of data to make accurate reliability estimates. These findings support the suggested move to the use of more stable quantities for reliability estimation and prediction.

Acknowledgements

This research was supported in part by grant 9803879 from the National Science Foundation.

Table of Contents

List of Figures

Chapter 1

Introduction

During the early years of computer use, hardware costs far outweighed software costs . However, due to steadily dropping hardware costs and increasing demand for more complex software systems, the majority of computing costs is attributed to software. Software systems are becoming more prevalent in life critical applications, such as flight control systems and space exploration. For these reasons, there is increasing demand for software systems that are fault-free. Unfortunately, in order to guarantee that a software system is completely free of faults it is neccesary that we exercise the software for every possible input in the system's input domain and to check the correctness of the subsequent output. This process is known as *exhaustive testing*. For some types of programs (such as concurrent systems) exhaustive testing may not be sufficient to guarantee that the software is completely free of faults.

For most software systems, exhaustive testing is not possible. For example, consider a program P which reads inputs from multiple remote sensors at an industrial plant and makes decisions about how operation of the plant should proceed based upon the input. There is no predictable limit on the number of inputs that will be read from the sensors by the program, so the number of possible inputs to the system is, for all intents and purposes, infinite.

Even when the input domain of a program is finite, exhaustive testing may

This dissertation follows the style of *The Physical Review*

not be feasible. If the input parameters are chosen from a finite set with many elements, then it may take an infeasibly long time to run all possible test cases and check all of the output of the system for correctness. Even if the set(s) from which the input is chosen is fairly small, if there are many input parameters, then testing the system for every possible combination of these inputs will result in a combinatorial explosion in the number of test cases. For example, consider a program P, that takes six input parameters $i_1, i_2, \ldots i_6$ chosen from a set S, where S is the set of integers from 1 to 1000. Now, although each of the input parameters is chosen from a finite domain with only 1000 elements, the number of possible inputs to the program is 10^{18}. If we assume that each test case can be generated, executed, and checked for correctness in one nanosecond, then conducting exhaustive testing on P will take about 10^9 seconds, or about 31.7 years. Obviously, conducting testing for this amount of time is infeasible.

Since it is generally impossible to conduct exhaustive testing of a software system, methods are needed for testing a software system with a manageable number of test cases, while still insuring that system components are rigorously exercised. Such methods are known as *software testing methods.*

Software testing methods can be divided into two classes: representative methods and directed methods. When a software system is tested using representative methods, inputs are chosen from the input domain of the system according to a model that represents the environment under which the system will eventually operate. This model is called the *operational profile* of the system. Inputs to the program that are expected to occur frequently during actual system use will be weighted more heavily by the operational profile than inputs that are not expected to occur as often. Therefore, the more frequently used components of the program will be tested more thoroughly. Additional weight may be given to functions in the system that are considered to be critical to operation.

When software testing is conducted using directed methods, test cases are designed to satisfy some coverage criteria with respect to the program's structure or functionality. For example, a common criterion used to drive directed testing is that every statement in the source code must be executed at least once during the

testing process. This particular coverage criterion is known as *statement coverage.* Conducting directed testing usually requires more effort on the part of the test team to design the test cases in order to meet the specified coverage requirements.

For many types of software, such as life critical or financial applications, it may not be enough to simply test the program. There may be a requirement for the quantification of the quality of the software. For this reason, methods have been developed to use data from the testing process to predict the future failure behavior of a software system. Suppose, for example, that a program P is tested for some period of time, t. Further, suppose that during the time interval $(0, t)$, m system failures, denoted by $f_1, f_2, \ldots f_m$ occur at times $t_1, t_2, \ldots t_m$, respectively. As the failures occur, the faults responsible for causing the failures are located and corrected. A logical question to ask would be, "Given this failure behavior of P up until time t what is the expected current failure rate of this program and when is the next failure expected to occur?" The field of *software reliability* attempts to provide an answer to this question.

Generally, the quantification of the reliability of a software system is based upon a mathematical model, called a *software reliability growth model.* A wide variety of models have been proposed, but the goal of all of the models is to estimate the current failure rate of the system under test and to provide estimates of the mean time to failure of the system. Typically, predictions made by software reliability models are only considered to be accurate if the failure data used as the basis for prediction is gathered using representative methods. Data obtained from directed testing methods has traditionally been considered to be poorly suited for reliability estimation. However, the accelerated pace of fault detection under directed testing methods would provide real advantages for test engineers whose job is to insure ultra-reliability [1]of a program. Therefore, it would benefit the field of software reliability if a technique were available for using failure data gathered during directed testing to make accurate estimates of the reliability of a program.

In the past, software reliability growth models have used failure data based

[1]According to Butler and Finelli [2], For a program to be ultra-reliable, the probability of program failure during one hour of operation must not exceed 10^{-7}.

on time-based quantities observed during testing. However, several researchers [9] [17] have observed that such measures are prone to statistical noise which may compromise model results. One goal of this research is to examine these claims and to suggest new types of data that can be used as input into reliability models to provide more stable and accurate results.

During this work, a mixed method approach to testing was developed that employs both representative and directed testing methods. Each type of testing method is used when it is most efficient to do so in order to accelerate the reliability testing process. Using the mixed method approach to testing required far fewer test cases than were required by representative testing.

Methods were developed to allow data from the mixed method approach to testing to be used for making reliability estimates. A software reliability growth model based on order statistics was developed to provide these estimates.

This dissertation is arranged as follows:

Chapter 2 presents a survey of the field of software reliability. General software reliability concepts are discussed and several existing software reliability models are described.

Chapter 3 presents a survey of the field of software testing. Some of the more popular testing methods are explained, and the relative strengths and weaknesses of representative and directed testing methods are discussed.

Chapter 4 defines the goals of the research by defining the problems are being solved and the methods used to solve them.

Chapter 5 discusses related work in the fields of software testing and software reliability.

Chapter 6 describes a mixed method approach to testing that involves a combination of both representative and directed methods that will allow for more efficient use of testing resources. This testing method is compared to existing testing methods, and its relative strengths and weaknesses are discussed.

Chapter 7 discusses the relationship between the set of faults found when a software system is tested using representative methods, and the set of faults found when the same software system is tested using directed methods. In this

chapter, the Ordered Directed Testing Property that is believed to describe this relationship is presented. Techniques that can be used to estimate the fault failure rate for a given fault, regardless of the testing technique used to find that fault are also presented.

Chapter 8 presents a reliability model based on order statistics that will make it possible to use failure data obtained from the mixed method testing technique to make accurate reliability estimates for a software system.

Chapter 9 describes the techniques that were used during this research to analyze the results of each experiment.

Chapter 10 describes the results that were obtained when the Order Statistics based model was applied to data sets gathered using representative methods. Comparisons are made to the results obtained by applying existing software reliability models to these data sets.

Chapter 11 describes the results that were obtained when the Order Statistics based model was applied to data sets obtained during a study that involved generating various debugging sequences for a set of failure rate data. The data sets consist of a mixture of representative testing data and directed testing data.

Chapter 12 describes the results that were obtained when an existing software system used for alarm tracking in an industrial enviorment was subjected to testing and reliability estimation.

Chapter 13 summarizes the research presented in this dissertation, and suggests directions for future research.

Chapter 2

An Overview Of Software Reliability

A generally accepted definition of software reliability is given by Musa et al. [12], who state that "...software reliability is the probability of failure-free operation of a computer program in a specified environment for a specified time." They then go on to state that a software reliability model "usually has the form of a random process that describes the behavior of failures with time."

This chapter provides a survey of some of the more popular software reliability models. The models discussed in this chapter are divided into two groups: Time Between Failure Models and Fault Count Models. Following the survey, the assumptions of these models are discussed.

2.1 Time Between Failure Models

As the name implies, time between failure models use interfailure times of a system under test to make reliability estimates. Some of the more popular time between failure models are discussed in this section.

2.1.1 The Jelinski-Moranda De-Eutrophication Model

Jelinski and Moranda [13] present what has come to be probably the most famous software reliability model. In this work, the process of fault removal is referred to as a de-eutrophication process. Thus, their model is known as the Jelinski-Moranda De-Eutrophication model.

The model developed by Jelinski and Moranda assumes that failures occur randomly as a software system is exercised. The model also assumes that the program failure rate between failures is constant, and that when a failure occurs and the corresponding fault is fixed, the failure rate of the system decreases by a constant amount. The latter assumption implies that all faults are the same size[1]. Experience shows that this assumption is not an accurate depiction of the real world.

The Jelinski-Moranda De-Eutrophication model assumes that the failure rate at any point in time is proportional to the current error content of the program. The failure rate of a program after i failures is given by:

$$\lambda_i = (N - i)\phi. \tag{1}$$

where N is the initial error content (number of errors) in the program and ϕ is a proportionality constant representing the step size of the decrease in the program failure rate when a fault is fixed. A possible realization of the Jelinski-Moranda De-Eutrophication process is shown in Figure 1.

2.1.2 The Moranda Geometric Model

In later work, Moranda [18] proposes a variation of the Jelinski-Moranda De-Eutrophication process. The so-called Geometric De-Eutrophication Process, though similar in spirit to the original model, has several important differences. First, the geometric model assumes an infinite number of faults in a program, where the original model assumed a finite number. Second, the geometric model

[1]The size of a fault refers to how often a given fault will manifest itself as a failure during system operation. Thus, the size of a fault refers to the extent that system reliability is affected by the existence of the fault in question.

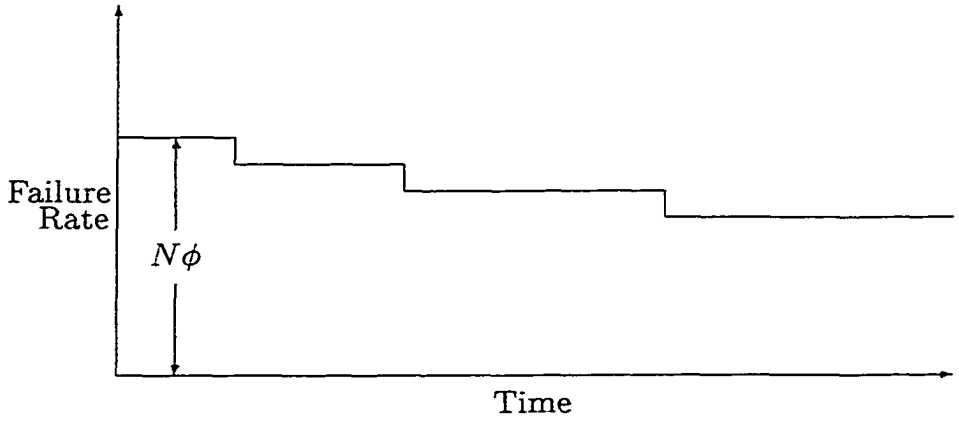

FIG. 1. The Jelinski-Moranda De-Eutrophication Process

assumes that when a failure occurs, then the system failure rate decreases by a geometrically varying amount. The parameters that describe the behavior of the geometric model are D, the initial failure rate of the system, and k $(0 < k < 1)$, the failure rate decay parameter. At the start of testing, the failure rate of the system is D. After one failure occurs (and is fixed), the failure rate of the system becomes kD. After the second failure is detected, the failure rate becomes k^2D. After i failures the system failure rate is given by:

$$\lambda_i = k^i D. \tag{2}$$

A possible realization of the Moranda Geometric De-Eutrophication process is given in Figure 2.

The change in the program failure rate in the geometric model seems to be more realistic than the change in failure rate of the De-Eutrophication model because one would expect that the faults found first under representative testing would be the same faults that would occur most often during actual system use. Therefore, removing these "large" faults will have more of an impact on the program failure rate than the removal of faults that manifest as failures less often.

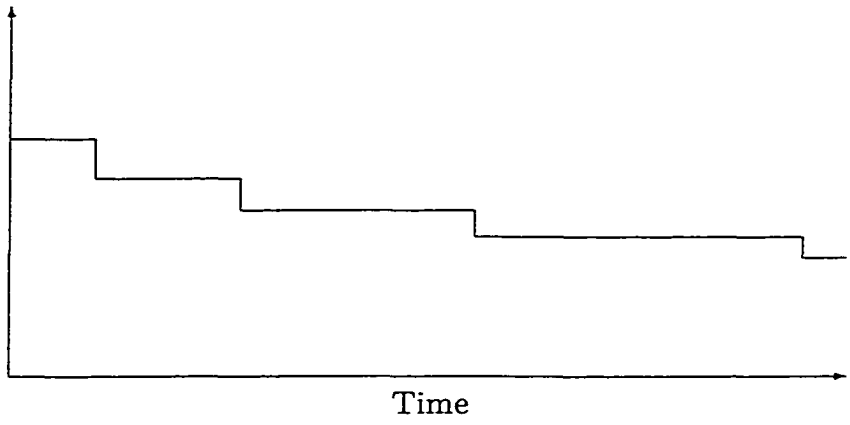

FIG. 2. The Moranda Geometric De-Eutrophication Process

2.1.3 The Musa Basic Model

Musa [12] develops a model that he refers to as the Basic Execution Time Model. This model is very similar to the Jelinski-Moranda De-Eutrophication Model. As in the Jelinski-Moranda Model, the per fault failure rate is assumed to be constant. The failure intensity for this model after μ faults have been removed is:

$$\lambda(\mu) = \lambda_0[1 - \frac{\mu}{v_0}] \tag{3}$$

where λ_0 is the initial program failure rate at the beginning of testing and v_0 is the total number of errors present in the software at the beginning of testing.

2.1.4 The Musa Log Poisson Model

Musa [12] develops a second model to address his concern that the operational profile for most software systems is not uniform. This model, referred to as the Logarithmic Poisson Execution Time Model, accounts for this non-uniformity by modeling the fault failure rates as decreasing with time. Therefore, removal of a fault early in testing will have a greater impact on the program failure rate than

removal of a fault later in testing. This behavior is more intuitively satisfying than the constant decrease in program failure rate in the Musa Basic Model and in the Jelinski-Moranda De-Eutrophication Model. The failure intensity for the Musa Log Poisson Model after μ faults have been removed is:

$$\lambda_\mu = \lambda_0 \exp(-\theta\mu) \tag{4}$$

where λ_0 is the initial program failure rate at the beginning of testing and θ is a decay parameter.

2.1.5 The Littlewood Model

Littlewood [15] proposed a refinement of the Jelinski-Moranda De-Eutrophication Model. As discussed above, the Jelinski-Moranda model assumes that all faults contribute the same amount to a program's failure rate. Littlewood argues that all faults in a system do not contribute equally to system failure rate. He argues that when testing is conducted in a representative fashion, the faults with high occurrence rates will cause failures before faults with low occurrence rates. Therefore, the largest faults in the system will manifest first and will be removed from the program first. Thus, a program's failure rate will decrease more when a failure is removed early in testing than it will when a failure is removed later in testing.

Like the Jelinski-Moranda De-Eutrophication model[13], Littlewood's model assumes that there are a finite number of faults in a program at the beginning of testing. The interfailure times of the faults are assumed to be exponentially distributed. Littlewood also assumes that when a failure occurs, the fault is immediately removed with probability 1. Littlewood's model assumes that each fault in a program has a failure rate that is independent of the failure rates of the other faults in the system. The failure rate of a program after i faults have been removed is given by:

$$\Lambda = \Phi_1 + \ldots + \Phi_{N-i}. \tag{5}$$

where each of the of the Φ's is has the probability density function $\beta gamd(\beta\phi;\alpha)$ at the beginning of testing.

If a program P has been tested for some time τ and that i faults have been detected and removed, then if we consider one of the remaining $N-i$ faults in the program, then its failure rate Φ is drawn from a probability density function (pdf) with the following form:

$$pdf\{\phi\} = (\beta+\tau)gamd([\beta+\tau]\phi;\alpha) \tag{6}$$

The program failure rate is

$$pdf\{\lambda\} = (\beta+\tau)gamd([\beta+\tau]\lambda;[N-i]\alpha). \tag{7}$$

The reliability of the program is

$$f\{t\} = ((\beta+\tau)/(\beta+\tau+t))^{(N-i)\alpha} \tag{8}$$

and the failure rate function is

$$\lambda(t) = ((N-i)\alpha)/(\beta+\tau+t)). \tag{9}$$

2.2 Failure Counting Models

Failure Counting Models use the number of failures that occur during a certain time interval as the basis for making estimates and predictions. Most models of this type use a Poisson distribution to describe failure activity. The Poisson distribution is simply a special case of the binomial distribution where the number of trials, n, is very large and the probability of success, p, for each trial is small. The expected number of successes in n trials is given by: $m(t) = np$.

The Goel And Okumoto NHPP Model (1979)

Goel and Okumoto [6] model the debugging process of a software system as a nonhomogenous Poisson process (NHPP). Several assumptions about the failure process are made by the authors. These assumptions are:

1. $\{N(t), t \geq 0\}$ is a counting process representing the number of failures by time t.
2. $N(0) = 0$
3. The counting process has independent increments, which means that the number of failures during an interval is independent of the number of failures in all other intervals which do not overlap with the interval in question.
4. The number of software failures in an interval is proportional to the expected number of undetected errors at the beginning of the interval.
5. The probability of more than one failure during a small interval is negligible.

The authors let $m(t)$ represent the expected number of software failures by time t. The authors assume a finite number of faults in the software system, so $m(t)$ is bounded in the following way:

$$m(t) = \begin{cases} 0 & t = 0 \\ a & t \to \infty \end{cases}$$

The authors then use assumption 4 above to state that the expected number of failures in an interval $(t, t + \Delta t)$ is described by:

$$m(t + \Delta t) - m(t) = b\{a - m(t)\}\Delta t + o(\Delta t). \tag{10}$$

where

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

which follows from assumption 5. Then, letting $\Delta t \to 0$, and dividing through by Δt yields a differential equation that can be solved to give the mean value function of the process. This function is:

$$m(t) = a(1 - \exp(-bt)) \tag{11}$$

The intensity function is simply the derivative of the mean value function. Specifically:

$$\lambda(t) = m'(t) = ab\exp(-bt) \tag{12}$$

Given the intensity function, λ, the authors describe the probability of a certain number of failures occurring during a given interval as follows:

$$N(t+\Delta t) - N(t) = \begin{cases} 0 \text{ with probability} & 1 - \lambda(t)\Delta t + o(\Delta t) \\ 1 \text{ with probability} & \lambda(t)\Delta t + o(\Delta t) \\ 2 \text{ or more with probability} & o(\Delta t) \end{cases}$$

Now, since the expected value of this Poisson process is $m(t)$, the Poisson distribution of the number of failures at time t is given by:

$$Pr(N(t) = y) = ([m(t)]^y / y!)\exp(-m(t)) \tag{13}$$

The Modified NHPP Model Of Yamada, Ohba, And Osaki

Yamanda et al. [22] present a modification of the NHPP model proposed by Goel And Okumoto. The reader will recall that the authors in [6] proposed a mean value function which was characterized by exponential growth. Yamanda et al. [22] suggest that the software failure process would be modeled better with an S-shaped growth curve. The reasoning behind this assertion is that the test team will undergo a "learning" period at the beginning of testing, and will not be very successful at uncovering faults until they are familiar with the testing environment. The mean value function is then given as:

$$M(t) = a[1 - (1 + bt)exp(-bt)], \tag{14}$$

where a is the total number of failures to be detected in infinite time and b is a constant of proportionality describing the error detection rate per error.

2.3 A Discussion Of The Assumptions Made By Existing Software Reliability Models

The research presented in this dissertation was at least partly inspired by the shortcomings of existing reliability models. This section discusses the assumptions made by these models that are problematic. Future chapters discuss how the model developed in this dissertation addresses each of these problems.

2.3.1 Assumption: All Faults Are Created Equal

Of the above models, the Jelinski-Moranda De-Eutrophication Model and the Musa Basic Model assume that all faults in the program contribute equally to the program failure rate. In other words, these models assume that all faults are the same size. This assumption is probably not accurate for most programs. For example, consider a program that allows the user to enter commands that update a database system. Suppose that there are only two commands in this system: an ADD command to add a new record to the database and a DELETE command to delete a record from the database. Let the input distribution be such that for every 100 commands entered by the user, ninety will be ADD's and ten will be DELETE'S. Now, let us assume that the code for each command has one fault in it that will always cause a failure when executed. Obviously then, the fault in the ADD command will cause more problems than the fault in the DELETE command. Accordingly, removal of the fault in the ADD command will have more of an impact on the program failure rate than removal of the fault in the DELETE command. Therefore, it is plain to see that the assumption made by these two models that all faults are the same size is not an accurate depiction of the real world.

2.3.2 Assumption: Testing Is Conducted Using Representative Methods

All of the models presented in this survey provide reliability estimates and predictions based on failure times observed during testing. In order for this type of measure to provide accurate predictions for system behavior during actual use, it is necessary that the system be exercised during testing the same way it will be exercised during actual use. Therefore, these models required that testing be conducted using representative methods.

If testing is conducted using non-representative methods, then the reliability models discussed in the chapter will not provide accurate estimates or predictions of program behavior. This fact is especially problematic since it seems that most testing that is currently being conducted is non-representative.

Even if testing is conducted using representative methods, the use of data observed during testing is problematic because it represents only one possible instantiation of a random process, and is therefore subject to a high levels of statistical variance. Because of the large potential for randomness in the data observed during testing, reliability estimates that rely on this data may not be accurate. This problem is discussed in more detail in the work by Hoppa and Wilson[9] and the work by Mitchell and Zeil [17].

2.3.3 Assumption: Faults Are Found In Perfect Order

With the exception of the Jelinski-Moranda Deeutrophication Model and the Musa Basic Model, all of the models discussed above assume that fault failure rates decrease with every fault that is found. There is no provision in the models for finding a fault that is bigger than a fault that was previously found. As discussed in the previous section, however, it is likely (even under representative testing) that faults will be found "out of order". Imperfectly ordered fault observations could cause existing models to provide inaccurate estimates and predictions.

Chapter 3

Overview Of Testing Methods

The previous chapter examined software reliability models which are used in conjunction with failure data from a software system to make estimates about the current reliability of the system. The failure data is obtained when the system is tested according to some criterion, called a *testing method.*

Most traditional software reliablility models assume that reliability data is given in terms of program failure rates (interfailure times). The failure rate data available for reliability estimation depends greatly on the way that testing is conducted.

For example, suppose we have a program P which has 100 possible input values, and suppose that P always fails for 10 of these possible values, but never fails for any other input. Now, if P is tested using some criterion A which causes P to never be executed using any of these 10 input values, then the perceived program failure rate will be 0. On the other hand, if P is executed using some criterion B which does test P using some of these 10 input values, then the perceived current program failure rate will be non-zero. Therefore, it is plain to see that the testing method can have a large effect on the estimates and predictions made by software reliability models.

For these reasons, an understanding of software reliability requires a basic understanding of software testing methods. This chapter provides a survey of software testing methods. The first part of this chapter discusses representative

testing, which is the only form of testing that has traditionally been used for making reliability estimates. The second part of this chapter discusses directed testing methods. The relative advantages and disadvantages of each type of testing method with respect to software reliability will be discussed.

3.1 Representative Testing Methods

The goal of representative testing is to exercise a program during testing the same way that it will be exercised during deployment. More formally, consider a program P with input domain D. Assume that once P is implemented in its actual operating environment, inputs will be chosen from D according to some pdf $f(d), d \in D$. Representative testing dictates that P be tested in such a way that the inputs to P are chosen from D according to f.

As mentioned in the previous section, existing software reliability models assume that failure data is given in terms of program failure rates. Obviously then, in order for reliability estimates to be accurate, the program should fail during testing the same way that it will fail during actual system use. Therefore, it follows that if we want the program to fail during testing that same way that it will fail during actual system use, then the program should be exercised during testing the same way it will be exercised during actual system use. Using representative testing methods, we achieve both of these goals.

When representative methods are used to test a program, P, no information about the structure of P is needed in order to conduct testing. For this reason, representative testing is sometimes referred to as a black-box testing method. The test team does not have to be familiar with the inner workings of the program in order to conduct representative testing. This fact removes a substantial learning curve for the test team. Black box testing is in keeping with the idea of information hiding which has become popular with the increasing use of object oriented programming techniques.

```
main:%{coin} "," %{die} "\n";
coin:6:H,4:T;
die:[1-6];
```

FIG. 3. A Sample DGL Grammar For Simulating The Toss Of An Unfair Coin And The Roll Of A Fair Die

3.1.1 Generating Representative Test Cases

Generation of test cases for representative testing is often inexpensive. Tools are available that, when given a grammar that defines the operational profile of a system, will automatically generate inputs for the system according to the operational profile. As an example of such a tool, consider the Data Generation Language (DGL) described by Maurer [16]. DGL allows users to specify a context free grammar (CFG) that describes how input items are chosen from a set of possible inputs. The user can associate probabilities for each possible input, which makes it possible to construct a grammar that allows inputs to be automatically chosen from a set according to an operational profile. Invocation of DGL for a given CFG causes a C program to be created that, when executed, generates the output specified by the grammar. Figure 3 gives an example of a simple grammar that generates inputs to simulate the toss of an unfair coin (weighted so that the probability of heads is 0.6) and the toss of a fair die.

By using automated tools, a large number of test cases can be quickly generated with no need for human intervention. Thus, human resources are freed to work on other tasks, such as insuring the accuracy of the operational profile and checking output to verify correctness. Further, if representative testing is used and a new system is being designed to replace an older system, large quantities of real input data from the older system may be available for use during testing.

3.1.2 Operational Profiles

Before representative testing can be conducted, the operational profile of the system must be specified. Hamlet [7] argues that there are many systems for which an operational profile does not exist or is not known.

Even if an operational profile exists for a program, discovering this distribution will probably not be an easy task. Musa [20] outlines a manual technique for determining operational profiles, but working through the construction of the sample profile given in [20] impresses the reader with how time-consuming and difficult such a construction can be.

Even if it is possible to develop the operational profile for a system, the operational profile may change over time. For example, if a system is moved into a new operating environment, or modifications are made to the system, the operational profile of the system is likely to change. When the operational profile of a system is changed, it is necessary to regenerate the test sets according to the new operational profile and to perform testing according to the new profile.

3.1.3 Testing For Ultra-Reliability Using Representative Methods

Life-critical or mission-critical software often requires that extremely high levels reliability be obtained. It is not uncommon for such software to be required to have a failure rate of less than 10^{-9} before it can be released. Butler and Finelli [2] state that reliability of this magnitude could require several years or more of testing when representative methods are used. This amount of time obviously cannot be devoted to testing the program.

Even if we were able to execute 10^9 test cases to insure ultra-reliability, we would still be faced with other problems. When a program is being tested, the inputs to the program cause output to be produced that must be checked for correctness. The problem of how to verify the correctness of output from a program being tested is known as the *oracle problem*. No matter what testing method is used, the oracle problem is an issue that must be addressed. Because of the very

large number of test cases required by representative testing, the oracle problem is especially problematic. Musa observed that in many projects a previous version of the software can be used as a partial oracle, and Weyuker [28] offers some suggestions about how to ease the oracle problem for certain types of programs, but no general solution exists.

3.2 Directed Testing Methods

Where representative testing seeks to exercise a program during testing the same way that it will be exercised during deployment, directed testing seeks to intentionally exercise the software in a non-representative manner according to some coverage critera in order to find faults more quickly and to accelerate the testing process. More formally, consider a program P, and suppose that the input domain of P is some set D. Let C be some coverage criteria for P. Directed testing involves testing P in such a way that the inputs to P are chosen from D in order to satisfy C without regard to the operational profile of P. This section will first examine several different types of directed testing techniques and will then discuss the implications of using data from directed testing for reliability estimation.

3.2.1 Implementation-Based Techniques

Implementation-based testing techniques require knowledge about the structure of the program to be tested. Test cases are constructed in order to satisfy some coverage criterion with respect to the structure of the program. Implementation-based techniques are commonly referred to as white-box techniques.

Structural Testing

Statement coverage and branch coverage are two examples of structural testing methods. Statement coverage requires that every statement in the program under test be executed at least once. Branch coverage requires that every conditional statement in the program under test be executed at least once for each of the two

possible truth values of that statement. Branch coverage is a stronger coverage criteria than statement coverage, because in order to achieve branch coverage, statement coverage must also be achieved.

Several variations of branch coverage and statment coverage have been employed to improve the fault finding ability of the testing process. For example, condition coverage is a variation of branch coverage that attempts to overcome one of the shortcomings of branch coverage. White [29] gives the following example of a conditional statement:

```
IF (A>0) AND (B<5)
```

Branch coverage will insure that this statement will be executed twice, once for each truth value. However, branch coverage does not take into account that the truth value of this statement is based upon the truth value of two separate predicates. Condition coverage, on the other hand insures that this conditional statement is executed at least once for each possible truth value of each of the component predicates. Thus, this conditional statement will be executed at least four times by a test set that satifies the condition coverage criterion.

Data Flow Testing

Rapps and Weyuker [25] outline a technique for selecting test cases for a program by using data flow testing. The program to be tested is viewed as a flow graph, with nodes in the graph representing blocks of statements that are sequentially executed. The edges of the graph represent transfer statements (either conditional or unconditional) between nodes. The selection of test data is driven by the requirement that some criterion that deals with the way in which values are bound to program variables be satisified.

Let P be a set of complete paths through a flow graph G that represents some program. Along such a path, a def of a variable x reaches a use of x if there are no subsequent redefinitions of x between these two points. Now, suppose that the execution of the set of paths P results from the use of a set of test cases S with some program T. The criterion known as *all-defs* requires that for each variable

definition in each node of G, then there must be a path $p \in P$ that includes that definition and a subsequent use reached by that definition.

Another criterion suggested by the Rapps and Weyuker is the *all-uses* criterion. The all-uses criterion is similiar to the all-defs criterion, except that the all-uses criterion requires that for each variable definition in each node of G, then the paths in P must include *all* subsequent uses reached by that variable definition.

After the work in [25], several papers were published suggesting new data flow path selection criteria. Clarke et al. [14] present an evaluation of these suggested criteria. The various criteria are analyzed and ranked according to their subsumption relationships.

Fault-Based Testing

Strong Mutation Testing Mutation analysis is a directed testing technique that was suggested by Demillo, Lipton, and Sayward [24]. In mutation testing, a number of mutant programs are created by injecting small changes in a program P that is being tested using a certain test set. If the output of one of the mutant programs is different from the output of P for at least one of the test cases in the test set, that mutant is said to be killed. The number of mutants that are killed by a test set can be used as a measure of how well that test set exercises the program being tested. If a large number of mutants survive the testing process, then the test set is probably not adequately exercising the program. The mutants that survive the testing process can be the basis for further development of the test set being used.

Weak Mutation Testing A variation of strong mutation testing, known as weak mutation testing, was suggested by Howden in [10]. The basic difference between strong and weak mutation testing is the way in which it is determined if a mutant has been killed. In strong mutation testing, a mutant is killed if the output from the mutant program is different from the output of the original program. In weak mutation testing, a difference in program output is not required in order to kill a mutant. A mutant is killed under weak mutation testing if the

internal data state of the mutant becomes infected, making it different from the internal data state of the original program. This requirement for killing a mutant is a weaker requirement than requiring outputs to be different which is why this technique is referred to as weak mutation testing.

Pertubation Testing Zeil [32] proposes a fault based testing method called Pertubation Testing. This testing method is based on a model that describes the conditions under which a domain error can be caused by a fault in an arithmetic or relational expression. This set of conditions defines a set of possible faults left undetected by a given test set. This information can be used to guide subsequent testing.

As testing is conducted, each execution of a predictate that does not cause an incorrect path to be taken imposes a constraint on the geometric space of possible faults for this statement. This constraint divides the space of possible faults into tested and untested regions. By keeping track of the linear inequalities that define the untested region, one can choose test cases to impose borders within this untested region to further enclose the untested region in any direction. By enclosing the untested region in this manner, testers are able to control the type of faults that remain undetected in the software.

Error-Based Testing

Domain Testing White, et al. [30] propose a testing method that they refer to as the Domain Strategy Testing Method. The primary focus of this testing method is to detect errors in predicates that could cause a wrong path to be taken during system execution.

To test a given predicate for correctness in a program, each variable in the predicate is replaced by its symbolic value, given in terms of input variables. After performing this substitution, the resulting predicate is given in terms of only input variables, and is therefore simply a set of constraints on input selection.

The authors note that if one considers the execution of a complete path through a program, then the path condition for this path is simply the conjunction of the

predicates along the path. By performing the symbolic substitution described above, a set of constraints that define the input space domain required to execute the selected path is obtained. This conjunction of constraints on the input space of the program defines a geometric area of the input space referred to as the domain of the path condition. Any input values selected from this geometric area of the input space will satisfy the path condition for the path under consideration, and will cause the path to be executed.

As stated earlier, the purpose of domain testing is to detect predicate errors that could cause the incorrect path to the executed. Another way of viewing this type of error is that an incorrect predicte causes a shift in the geometric area defining the input domain for the path condition. Then, due to this shift in the border of the geometric input domain, it is possible that input values will be incorrectly included in or left out of the input domain in question. Domain testing was created to detect this type of error.

When using this testing method, each border segment of the input domain is considered. Test points are generated for each border segment in such a way that correct processing of the test points implies that the predicate defining the border is correct. Test selection consists of the selection of a series of ON-OFF-ON data points, where ON points are located on the border and OFF points are located on the open side of the border. The distance of the OFF points from the border defines the minimum border shift that this testing method will detect. This testing method will detect any border shifts with a magnitude larger than this distance.

Figure 4 shows the selection of three test points meeting the ON-OFF-ON requirements for a closed border segment. The dotted line indicates the correct border segment, while the solid line indicates the given border. In this case, the error would be detected because the OFF lies on the opposite side of the correct border from the given border.

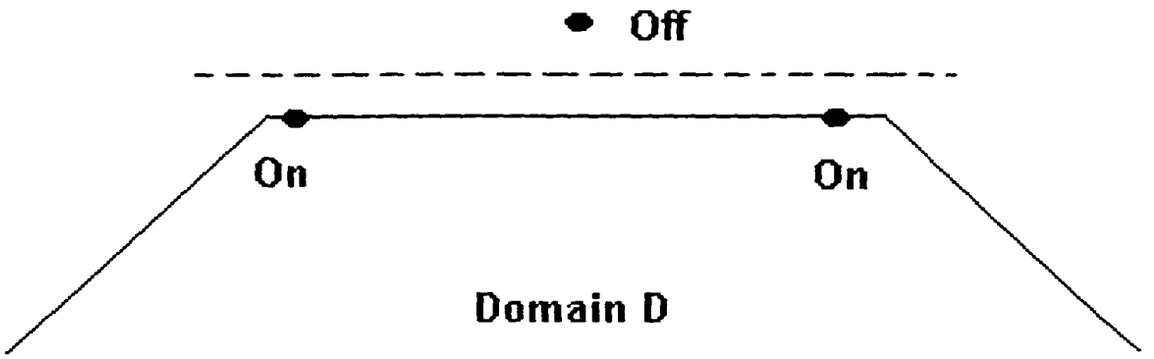

FIG. 4. Domain Testing Example

3.2.2 Specification-Based Techniques

Specification-based testing techniques do not require knowledge about the structure of the program to be tested. All knowledge required for driving the testing process is obtained from the program specification. For this reason, specification-based techniques are known as *black-box* techniques.

Equivalence Partition Testing

Equivalence partition testing is a directed testing method that was proposed by Howden [11]. When equivalence partition testing is conducted, for each function that is to be tested, the input domain of that function is partitioned into equivalence classes that are assumed to be homogeneous. In other words, if one element of an equivalence class is processed correctly by the program, then all of the elements of that equivalence class will be processed correctly by the program. Testing is conducted by choosing one input point from each equivalence class for each function in the program.

The major problem with partition testing is the question of whether is it is possible to divide the input domain for each function into homogeneous equivalence classes. If it is not possible to do this, then partition testing basically reduces to random testing with relatively few test cases. A major problem with most testing

methods that involve partitioning the input domain of a program or function is that no systematic process exists for conducting the partitioning process .

As a solution to this problem, Ostrand and Balcer [23] proposed category-partition testing. Category-partition testing attempts to provide a systematic method for constructing the test sets to drive equivalence partition testing. Category-partition testing consists of several steps. The first step involves involves identifying all of the functional units in the program. In the second step, each of the functional units is examined, and the variables that affect the operation of each functional unit are identified. The third step of the category partition method attempts to divide the variables for each functional unit into categories to allow input points that are likely to cause faults in the software to be chosen. Ostrand and Balcer [23] state that "Tests [should] be designed to to maximize the chances of finding errors in the software." The final step of category-partition testing is to partition each category into choices. The choices in category-partition testing play the same role as the partitions in partition testing. Testing is conducted by choosing inputs points from each choice within each category for each functional unit in the program.

Boundary-Value Analysis

Myers [21] states that test cases that explore boundary conditions are generally more successful at finding faults than test cases that do not. For this reason, he suggests a black-box method called boundary-value analysis. Boundary-value analysis utilizes many of the ideas of equivalence partition testing, but instead of allowing the tester to select any of the possible inputs from a given equivalence class to generate a test case, boundary-value analysis dictates that the tester should select test cases from the equivalence class to insure that each edge of the class is the focus of at least one test.

Intuitively, this testing method makes sense because all software developers have probably created test cases to verify the behavior of their software for "special" input values. For example, suppose that a given program is supposed to read a text file and determine how many times the word "computer" appears in

the file. Several obvious test cases a tester might include in his test suite involve "special" input values. Some of the possible choices for these test cases are:

- An empty input file.
- An input file with no occurrences of the word "computer".
- An input file with only the word "computer".
- An input file the contains substrings of the word "computer" (e.g. "compute").

Cause-Effect Graphing

Most of the testing methods discussed in this section only use individual system input values to generate test cases. Errors caused by interactions between input values are not given special attention. Cause-effect graphing [21] is a black box testing method that provides a systematic method for identifying and testing interesting combinations of input conditions.

When conducting cause-effect graphing, high-level system specifications are used to partition system output values into classes called *effects*. Each of these classes is analyzed to determine the input states responsible for creating the effect. These input states are referred to as *causes*. The program input space is partitioned using the cause-effect relationship and the cause-effect relationships are translated into a boolean lookup table that is used to generate test cases.

3.3 Directed Testing And Reliability Estimation

As mentioned earlier, existing reliability models use program failure rate information to make software reliability predictions. Since directed testing techniques seek to accelerate the rate of fault detection during testing, the program failure rate data found during testing with directed methods will not be the same program failure rate data that would be seen if the system were in actual use. Therefore, program failure rate data gathered using directed testing methods cannot be used

by existing reliability models for making reliability estimates. However, if this basic problem can be solved (or worked around), then there are certain properties of directed testing that make it desirable for use with software reliability estimation.

One advantage of directed testing is that since test cases are designed manually to exercise a particular area of the program, a single test case designed for use in directed testing may be more effective at finding faults than a single test case generated for use in representative testing. Therefore, directed testing may require fewer test cases than representative testing to achieve similar levels of program reliability. This fact is especially important when faced with the task of testing for ultra-reliability. For example, suppose that we are testing a program P for which a failure rate of no more than 10^{-7} is required. Further, suppose that there is a fault, e, in a segment of infrequently executed code. Assume that the probability that the faulty code will be executed is 10^{-6}, and that if the code is executed then the probability of failure is 1. Therefore, e has a failure rate of 10^{-6}. If P is to achieve the required reliability of 10^{-7}, then the fault e must be found. If representative testing were conducted, we would expect to have to generate, execute, and check about 10^6 test cases before the fault will be found. On the other hand, if directed testing were used, this fault would probably be exposed much sooner, since a test case would probably be designed specifically to exercise this area of the program. Therefore, directed testing can deal very well with one of the main problems faced by representative testing, namely the need to execute an enormous number of test cases to uncover faults with small failure rates.

A second advantage of directed testing is that an automated oracle is not as essential to directed testing as it is to representative testing. Since directed testing usually requires far fewer test cases than representative testing, directed testing causes much less output to be generated. When directed testing is used, a fully automated oracle may not be required. A much simpler, partially automated oracle that makes use of human intervention may be sufficient. Future chapters will discuss a testing framework that combines the representative and directed testing methods to allow them to complement each other in a more robust and efficient testing process than would be possible if only one of the methods was

used.

3.4 Summary

This chapter has focused on software testing techniques. This dissertation divides testing techniques into two groups: representative methods and directed methods. Existing software reliability models only allow predictions to be made when failure data is gathered using representative methods. However, directed testing methods have several advantages that would make their use attractive to software reliability practicioners, if some fundamental problems can be solved. The remainder of this dissertation focuses on how these problems can be solved, and how representative and directed testing methods can be used together to improve the software reliabililty testing process.

Chapter 4

Goals Of This Research

This chapter outlines the goals of this research, and will summarize the steps that were taken to reach these goals.

4.1 Development Of A Mixed Method Testing Process

One of the goals of this research was to develop a software testing method that incorporates both representative and directed testing methods at various points in the testing process. Each component testing method is used during the time when it is most efficient to do so.

4.2 Development Of Techniques To Allow Reliability Estimation Regardless Of The Testing Method Used

As stated in the previous chapter, failure data obtained using directed testing methods cannot be used with existing software reliability models because existing models base their predictions on the observed program failure rate, which varies

depending on the testing method being used.

One of the goals of this research was to develop techniques for obtaining failure rate data from a program that does not depend on the way that the program is tested. For this reason, the emphasis of data collection is switched from quantities observed during testing to quantities obtained during debugging.

Specifically, the random variable of interest for reliability estimation is switched from observed program failure rates to individual fault failure rates. To determine the fault failure rate for a given fault, one must answer the question: "If we were testing this program using representative testing, how often would we expect this fault to fail?". If this question can be answered, then testing can be conducted using any method, because we will have obtained a quantity that is independent of the testing method being used.

4.3 Development Of A Software Reliability Model Capable Of Using Directed Testing Data

Even if the random variable of interest was switched from program failure rates to fault failure rates as described above, no software reliability model existed that could use this data for making predictions. Therefore, a software reliability model that could deal with data in the form of fault failure rates, program failure rates, or any mixture of the two was needed.

4.4 Validation Of The Developed Model

Once the model was developed, it was neccesary to evaluate its performance relative to the performance of existing software reliability models . Several experiments were conducted to achieve this goal.

4.5 Evaluation Of The Suitability Of Time Based Data For Reliability Estimation

As stated earlier in this dissertation, existing software reliability growth models use failure data based on time-based quantities observed during testing. However, several researchers [9] [17] have observed that such measures are very prone to statistical noise which may compromise model results. One of the goals of this research is to investigate this claim and to suggest types of data that can be used as input into reliability models to provide more stable and accurate results.

Chapter 5

Related Work

The research presented in this dissertation takes a different approach to software reliability than most previous work. However, several authors have recently addressed issues that are related to this research. This chapter presents a survey of the most relevant of these papers.

5.1 The Effects Of Fault Recovery Order On Software Reliability Models

Hoppa and Wilson [9] investigate an inherent problem with the way that most software reliability testing is conducted. They state that when software reliability testing is conducted, the failures that are observed represent only one possible realization of the debugging process. The authors state that repeating the debugging process will probably yield a different order of failure detection (and different interfailure times) each time the process is repeated. The focus of the Wilson's and Hoppa's research [9] was to determine if this different ordering (and the change in observed interfailure times) has any effect on the predictions provided by software reliability models.

The results of the research indicate that existing software reliablity models are

sensitive to different fault recovery orders. The model developed in this dissertation is specifically designed to minimize the error due to varying fault recovery order because the model focuses on fault failure rates instead of interfailure times (program failure rates).

5.2 PIE: A Dynamic Failure-Based Techniqe

Voas [27] proposes a framework for estimating whether faults are likely to be uncovered by testing. Each section of code in a program is analyzed by considering three component probabilities. These probabilities are the exection probability, the infection probability, and the propagation probability. The execution probability for a given segment of a program is the probability that the segment of code will be executed. The infection probability is the probability that execution of the given segment of code will result in an incorrect program data state. The propagation probability for a given segment of code is the probability that an infection of the data state at the code location will cause visible incorrect behavior of the software. These three probabilities combine to give the effective failure rate for the segment of code. One of the problems that needed to be solved in this research was how to determine the actual fault failure rate for a given fault. The framework proposed by Voas provides a possible solution to this problem. In fact, the problem we are trying to solve (determining the failure rate for one fault in one segment of code) should be simpler than the problem that Voas was trying to solve (determining the failure rates for all faults in all segments of code).

5.3 The Relationship Between Test Coverage And Reliability

Malaiya, et al. [26], propose a model that allows coverage information from directed testing methods to be used to estimate the reliability of the software being tested. The authors note that a basic problem with existing software reliability

models is that they require data from representative testing, and that representative testing is seldom used in practice. They state that directed testing is a faster and more effective than representative testing, and a method is needed for using data from directed testing for making reliability estimates.

Malaiya, et al. propose that a relationship exists between test coverage during directed testing and defect coverage in the program, and that a relationship exists between test coverage during directed testing and the subsequent reliability of the software. The authors develop a model that allows software reliability predictions to be made based on an estimated initial number of faults in the program being tested and the coverage achieved during directed testing.

The problem that Malaiya, et al. try to solve is largely the same as one of the problems this research solves. We also want to use data from directed testing to make reliability estimates for most of the same reasons. However, instead of relying solely on the coverage obtained during testing, our research takes the extra step of considering the fault failure rates of the faults uncovered during directed testing which should provide a more "customized-fit" of our model to each program being tested.

Another similarity between Malaiya's research our research is the proposed nature of the fault sets found by directed testing. Malaiya, et al. postulate that given a certain level of test coverage during directed testing, then a certain set of faults will be exposed thereby achieving a certain level of reliability. There are certainly similarities between this idea and the Ordered Directed Testing Property presented later in this dissertation.

5.4 Software Testability

Voas [8] [5] discusses the concept of *software testability*. Voas defines the testability of a program to be the probability that if a fault exists in that program, then that fault will be detected by any testing scheme. In the words of Voas and Hamlet, the testability of a given program defines the degree to which that program "wears its faults on its sleeve".

As discussed in previous sections (and to be discussed in more detail later), the software reliability model proposed in this dissertation uses individual fault failure rates as a basis for making estimates and predictions for a system under test. Before this use of fault failure rates can be justified, one question that needs to be answered is "How can you expect to use the fault failure rates observed in the past to predict the fault failure rates that you will see in the future?"

By using the concept of software testability, we argue that a given program has certain attributes that determine its ability to hide faults from the testing process. Then, since software testability is a property inherent to the program, we would expect that the failure rates for all of the faults in the program are constrained to some extent by this level of testability. Therefore, we can view the testing process as a walk through the structure of the program. The fault failure rates we obtain are indicative of this structure. Thus, we have a basis for arguing that the faults failure rates we have observed in the past share a common distribution with the faults we expect to see in the future.

Chapter 6

A Mixed Method Approach To Testing

Previous chapters discussed representative and directed testing methods and examined the advantages and disadvantages of each method. The applicability of data obtained from each of these testing methods towards reliability estimation was discussed as well. This chapter outlines a testing method that utilizes both directed and representative testing methods. This testing method is organized so that the advantages of each component method are utilized, while the effects of the disadvantages of each component are minimized.

6.1 Overview

It is useful to view the debugging process as consisting of two phases. During the first phase of the debugging process, a relatively large number of faults remain in the program, so the program failure rate is relatively high and interfailure times are relatively short. As more and more faults are removed from the program, the program enters the second phase of the debugging process. During the second phase of the debugging process, relatively few faults remain in the program, so the program failure rate is low and interfailure times tend to be very long. The

following sections examine these two phases of the debugging process. The properties of each phase are used to determine the testing method that will be the most efficient during that phase.

6.2 Phase 1: Testing When A Large Number Of Faults Remain

Consider a program, P, to be tested. During the beginning of testing, there will be a relatively large number of faults in P, so the program failure rate will be relatively high and the interfailure time of P will be short.

Since the faults found early in testing will have short interfailure times, then the failure rate of these faults will be estimated to be large. Therefore, the removal of faults early in testing has more impact on system reliability than the removal of faults later in testing. For this reason, it is important that the faults removed early in testing are the same faults that would have occurred first if P were implemented.

Additionally, since interfailure times are relatively short at the beginning of testing, valuable resources can be saved by using a testing method that offers significantly less expense per test case, rather than a more expensive testing method that may be more effective at finding failures on a per test case basis.

The above observations support using representative testing methods at the outset of testing.

6.3 Phase 2: Testing When A Small Number Of Faults Remain

Consider a program P that has been tested for some period of time using representative methods. As faults are removed from P, the interfailure times will become very long if representative testing is continued. Eventually, the interfailure times will become so long that conducting representative testing, which had previously

been the most practical choice for testing, will become more expensive than other testing methods. At this time, a switch to another testing method is warranted.

Consider the cost of generating a test case for use with representative methods. Let this cost be denoted by c_r. Let the cost of generating a test case for use with directed testing by c_d. When representative testing is conducted, the interfailure time of P will increase quickly. As the interfailure time increases, the number of test cases required to uncover a failure using representative testing will also increase quickly. However, the number of test cases required to uncover a failure using directed methods should not grow as rapidly. Let n_r represent the current number of test cases required to expose a fault in P if representative testing is used. Let n_d represent the current number of test cases required to expose a fault in P if directed testing is used. Now, as noted above, $c_d > c_r$. At the beginning of testing, we will have $c_d n_d > c_r n_r$. However, as testing continues, and the interfailure times for representative testing grow rapidly, we will eventually have $c_r n_r > c_d n_d$, at which time representative testing has become more expensive than directed testing. At this point, the testing process will become more efficient if we switch from representative testing methods to directed testing methods.

6.4 Advantages of Mixed Method Testing

By using a mixed method approach to software testing, we are able to take advantage of the strengths of each of the component testing methods. Further, we will not be subject to the biggest disadvantages of the component methods. The main advantages of representative testing are that test cases are relatively inexpensive to generate and that the behavior of the software during testing reflects how the system will behave once it is implemented. The biggest advantage of directed testing is its ability to detect errors with small failure rates without requiring an enormous number of test cases. By utilizing representative testing during the first stage of testing when the program failure rate is highest, we can use inexpensive test cases to find faults. Also, by conducting representative testing at the beginning of testing, the large faults present in the program will be detected early in

testing and the expected large reliability growth at the beginning of testing can be achieved. Once all of the large faults have been removed from the system (and the program failure rate decreases accordingly), the interfailure time for the system will become very large if the test team continues to use representative testing. By switching to directed testing once it becomes cheaper to do so, we will be able to continue to achieve higher reliability with a reasonable amount of effort. Thus, a mixed method approach to testing benefits from the advantages of both of the component methods.

6.5 Disadvantages of Mixed Testing

Despite its advantages, the mixed method approach to testing outlined above does inherit some problems from its component methods. First, because representative testing is being utilized, a knowledge of the operational profile of the system is required. Second, because directed testing is being used, a knowledge of the structure of the program may be required, depending on the coverage criterion employed.

Chapter 7

Directed Testing And Reliability Estimation

Previous chapters have discussed the fact that existing software reliability models require that all data gathered for purposes of prediction be gathered using representative testing methods. These chapters also discussed how it would be advantageous to conduct reliability testing using a combination of representative and directed testing methods.

Three obstacles that prevent directed testing data from being used to make reliability predictions are (1) program failure rates observed during directed testing are not indicative of the program failure rates that would be observed during actual system use, (2) faults will be found in a non-intuitive order (not largest to smallest) by directed testing, and (3) there is no assurance that the set of faults found by directed testing for a given program will be the same set of faults found by representative testing.

The software reliability model developed in this dissertation overcomes the first problem by shifting the emphasis from program failure rates to individual fault failure rates which do not depend on the method used to conduct testing. Additionally, the proposed model solves the second problem through the use of Order-Statistics. Therefore, if problem three can be solved, then we will be able to use directed testing data to make software reliability estimates and predictions.

The following sections provide evidence that the set of faults found by directed testing for a given program will be the same set of faults found by representative testing, and discuss possible methods for determining the fault failure rate for a given fault.

7.1 The Nature Of Fault Detection Under Directed Testing

Cobbs and Mills [3] assert that directed testing methods find faults in random order, with small faults being as likely to be found as large faults. Specifically, they state: "coverage testing is as likely to find a rare execution failure as it is a frequent one". At first, this claim may seem reasonable, because directed testing techniques do not exercise software by choosing input points according to to the operational distribution that will be used once the system is implemented. However, with further thought, it becomes apparent that the distribution used to choose input points is not the only factor that plays a part in determining whether or not a fault manifests as a failure during program execution.

Voas [27] maintains that the manifestation of a fault as a system failure during program execution depends on three factors. His PIE (Propagation, Infection, Execution) model states that for a system failure to occur, three things must happen. First, the location in the program containing the associated fault(s) must be executed. This requirement is the Execution component of the PIE model. Second, once the location in the program containing the fault has been executed, it is necessary that the fault cause a change in the data state of the program. This requirement is the Infection component of the PIE model. Third, the incorrect data state caused by the fault must be propagated to the output. This requirement is the Propagation component of the PIE model. More formally, let ε_{lPD} represent the execution rate for a given location l in a program P when inputs are drawn randomly from a distribution D. Let ν_{mlPD} be the infection probability for a fault m at location l in program P. Let φ_{mlPD} represent the

propagation probability for fault m at location l in P. Then the failure rate, λ_{mlPD} of m at location l in program P when inputs are chosen according to distribution D is:

$$\lambda_{mlPD} = (\varepsilon_{lPD})(\nu_{mlPD})(\varphi_{mlPD}) \tag{15}$$

Of course, the above equation assumes the independence of the three component probabilities.

The input distribution used to conduct testing directly affects only one of these factors, the execution component. Therefore, in addition to the input distribution used to drive testing, structural components of a program also play a hand in determining the order in which faults manifest themselves during program use. These structural components are common to the fault manifestation process whether directed testing is conducted or representative testing is conducted. Therefore, the sets of faults found when directed testing is conducted should be approximately the same as the set of faults that would be found if representative testing were used.

Voas' testability research [8][5] provide additional support for this claim, by providing evidence that the set of faults found for a given program under any testing method is affected largely by the structure of that program.

With these observations in mind, the following property is now proposed:

Ordered Directed Testing Property: For a given directed testing method, as we approach coverage of the method, the set of k faults revealed will be the k faults with the largest individual operational failure rates.

Now, this property probably does not hold exactly. However, it is claimed that this property is closer to the truth than the assumption that directed testing will uncover faults with no regard to fault failure rate. Preliminary support for the Ordered Directed Testing Property was given in previous research conducted by Mitchell and Zeil [17].

If the assumptions of the Ordered Directed Testing Property hold, then directed testing can be used to conduct reliability testing without missing a substantial number of faults that would have been found by representative testing, and without finding a substantial number of faults that representative testing would have missed. The worst thing that has happened during the testing process is that the discovered faults may have been found in a different order than if testing had been conducted using representative methods.

As it turns out, the order of fault detection is not important to the software reliability model developed in this dissertation. Therefore, if the individual fault failure rates can be determined for the discovered faults, then data from directed testing can be used with the proposed model to make reliability estimates.

7.2 Estimating Fault Failure Rates

Traditional software reliability models depend on time based measures, such as time between failures, as a basis for estimation and prediction. It is exactly this dependence on such measures that requires the use of representative testing with these models. When directed testing is conducted, the time between failures during testing will not necessarily reflect the time between failures during actual system use. In addition to using measures based on observed program failure times for making estimates and predictions, the model presented in this dissertation uses a measure that is independent of the method used to conduct testing.

This observed quantity used by the model in this dissertation is an estimate of the operational fault failure rate for a given fault. Basically, when a fault is found during the directed testing process, instead of recording the time of the failure (or other artifact of the testing process), the human debugger will use some technique to determine how often the given fault would manifest as a failure during actual system use. Thus, the responsibility for determining the quantities used for reliability estimation is taken away from the testing process itself and is placed with the human debugger. Such a shift should allow for a more robust estimation process where human intuition can play a role.

In addition, by gathering failure information in this manner, the model estimation problems associated with making estimates from a single realization of a random process described by Hoppa and Wilson[9] and Mitchell and Zeil [17] are avoided. These problems are avoided because the observed quantities of this random process are mapped to the *expected* value of the observed quantities.

The next section describes how the PIE Method of Voas [27] can be used to estimate fault failure rates. Section 7.2.2 then describes how a debugging team can use less formal methods to estimate fault failure rates.

7.2.1 The PIE Method For Determining The Fault Failure Rate

Consider a program P being tested using directed methods. Now, suppose that testing uncovers a fault, z, located at location l in P. Let this location be denoted as P_l. Now, according to the Voas, the probability of failure of z (the fault failure rate) is based on three component probabilities. These component probabilities are (1) the probability of execution of location P_l during actual system use, (2) the probability of data state infection when location P_l is executed, and (3) the probability of incorrect output when the data state is infected due to execution of location P_l. If these three component probabilities can be estimated, then their product provides an estimate of the fault failure rate for z.

The first component, the execution probability, is fairly easy to estimate. Suppose testing is being conducting for some program P and a failure is observed because of a fault z at location P_l. To estimate the execution probability for location P_l, the human debugger would stop the testing process, and run n representative test cases for P. Instrumentation in P will count the number of times n_e that P_l is executed. It is not neccesary to check the output of these representative tests for correctness, because the focus here is on the execution component of the PIE Model. The ratio $\frac{n_e}{n}$ provides an estimate of the execution probability for P_l. Tools commonly used for conventional performance profiling can be used by the human debugger to help make this estimate.

The second component, the infection probability, can be estimated by re-running the n_e test cases that reached location P_l when the execution probability was being estimated. Additional instrumentation at location P_l will be required to count the number of times n_i that the data state immediately following P_l is infected because of z. The ratio $\frac{n_i}{n_e}$ provides an estimate of the infection probability at location P_l.

The third component, the propagation probability, can be estimated by mutating the program state at P_l for each of the n_e test cases that reached this location. The number of times, n_p, that incorrect output results from the infection at P_l is counted, and the ratio $\frac{n_p}{n_e}$ provides an estimate of the propagation probability for z.

By combining these three components, an estimate of the fault failure rate for this fault is obtained that does not depend on the testing method being used.

7.2.2 Using Debugger Estimates To Determine Fault Failure Rates

Two more possible methods for determining fault failure rates are proposed in this section. These methods rely on the debugging team to estimate the fault failure rates. In both methods, when a failure occurs during testing, the debugging team will find and fix the associated fault. The team will then determine the fault failure rate.

In the first method, the debugging team determines the inputs that cause the fault to manifest as a failure. Once these inputs have been identified, the debugging team goes back to the operational profile of the system and estimates the fault failure rate. In the absence of an operational profile, the test team can estimate how often they feel the failure will occur based on the inputs that cause the failure. This type of estimation is probably not much different than what already goes on in practice during many test-debug sessions.

Another possible method that debuggers can use to determine the fault failure rate for the fault is to run representative tests for two versions of the program.

The first version of the program includes the fix that was made to remove the fault. The second version of the program does not include the fix that was made to remove the fault. The debuggers can then estimate the fault failure rate by running representative tests for both of these versions of the program and watching for differences in output.

Chapter 8

A Software Reliability Model Based On Order Statistics

Conventional software reliability models assume that the failure data to be used to make reliability estimates is obtained by representative testing methods. As a result, conventional software reliability models may not provide accurate reliability estimates when failure data is gathered using directed testing methods. This chapter presents a software reliability model that will allow failure data obtained by using either representative testing methods or directed testing methods (or both, as in the case of our mixed method approach to testing) to be used to make accurate reliability estimates.

8.1 Order Statistics Basics

This section presents a very brief discussion of the order statistics that will be needed for an understanding of the presentation of the proposed model. A more extensive treatment of the subject of order statistics appears in the text by H.A. David [4].

David [4] defines the distribution of an order statistic. If n random variables, $X_1, X_2, \ldots, X_n$ are arranged in ascending order, then $X_{r:n}$ denotes the r^{th} order statistic out of n. If we assume that the X_i are continuous with pdf $f(x) = F'(x)$,

the distribution of the $x_{r:n}$ is given by:

$$f_{r:n} = \frac{n!}{(r-1)!(n-r)!} F(x)^{r-1}[1 - F(x)]^{n-r} f(x) \tag{16}$$

where $f_{r:n}$ denotes the pdf of $X_{r:n}$.

8.2 Model Basics

Suppose that a program P has n faults $z_1, z_2, \ldots, z_n$ drawn from a set Z. Let $f(\phi) = F'(\phi)$ denote the pdf describing the distribution of the failure rates associated with the faults in Z. Let $\phi_1, \phi_2, \ldots, \phi_n$ represent the failure rates of the n faults in P ordered arbitrarily.

Now, suppose that P is tested until k faults have been removed. Let $\psi_1, \psi_2, \ldots, \psi_k$ denote the operational failure rates of these k faults, where the index of each $\psi_i, 1 <= i <= k$ reflects the order in which the fault was discovered. Further, suppose that these operational failure rates are independent.

If we consider the remaining $n - k$ undetected faults in P and denote them by $\psi_{k+1}, \psi_{k+2}, \ldots, \psi_n$, we can approximate the program failure rate of P after k faults have been removed by using the sum of the operational failure rates of the $n - k$ undetected faults. Specifically,

$$\lambda_k = \sum_{i=k+1}^{n} \psi_i \tag{17}$$

Use of this approximation assumes that there are a finite number of faults in the program, and that the faults failure rates are independent. These are common assumptions made by many existing software reliability models [13][12][15].

This equation is not particularly useful for making reliability estimates about P, because the current program failure rate is given in terms of the faults that have not yet been discovered. Therefore, we need another technique that can be used to estimate λ_k. The use of order statistics is at the heart of this proposed technique.

Suppose that we sort the $\phi_i, i \in 1 \ldots$, into ascending order. Let $[\phi_{i:n}]_{i=1}^{k}$ denote the resulting sequence. The pdf of the $\phi_{i:n}$ can then be described by the function

given in (16). If we can determine the nature of the $f_{r:n}, r \in 1 \ldots n,$) then we will be in a position to determine the expected values for operational failure rates of the remaining $n - k$ undetected faults in P. Specifically,

$$E(\phi_{r:n}) = \int_0^\infty \phi f_{r:n}(\phi) d\phi \tag{18}$$

The expected value of the program failure rate of P, λ_k is then given by:

$$E(\lambda_k) = \sum_{j=1}^{n-k} \int_0^\infty \phi f_{j:n}(\phi) d\phi \tag{19}$$

and the reliability of P is:

$$R(t) = exp(-\lambda_k t) \tag{20}$$

Thus, we have a model that makes use of fault failure rates as the random variable of interest. Further, the order in which the faults are discovered by testing does not affect the estimates made by this model. Therefore, this model can be used regardless of the methods used to conduct testing.

Chapter 9

Data Evaluation Plan

Before any experiments could be conducted, it was necessary to decide how the results of these experiments would be analyzed. This chapter describes the models involved in the experiments, the analysis methods used by the experiments, and the tools that were developed to aid in the analysis process.

9.1 Selection Of Models

For comparison purposes, three other software reliability models were selected to be compared to the Order Statistics Model. These models were selected because they seem to be the most often cited models in software reliability research, and are often used by other researchers to assess their proposed models.

The models that were selected are:

- Jelinski-Moranda Deeutrophication Model
- Musa Basic Model
- Musa Log Poission Model

9.2 Techniques For Comparing Model Performance

During the course of the experiments, the performance of each model was measured in three different ways.

First, the predictive accuracy of each model was measured. The predictive accuracy of a model specifies how closely the model's predictions match observed quantities. The need for this measure is obvious because the reason for using software reliability is to be able to make estimates about a program's current failure rate and to be able to predict the future performance of a program.

Second, the best fits of each model for each data set were compared to determine which model provided the best overall fit to the observed data. The results of this type of analysis provides information about the ability of each model to account for all of the observed values using a single set of parameters. High error values for the best fit of a given model may indicate that the model may not be flexible enough to account for the range of observed failure rates.

Third, the stability of the model parameter estimates was compared over multiple iterations of the fitting process, where each iteration introduced additional failure rate data. The following sections describe how each of these three types of analysis was conducted.

9.2.1 Traditional Means Of Measuring The Predictive Accuracy Of Software Reliability Models

Brockhurst and Littlewood present a statistical tool called a U-Plot that measures the predictive accuracy of a software reliability model [1]. The U-Plot uses the cumulative distribution function (*cdf*) of the program failure rates. Brockhurst asserts that given a set of observed program failure rates and the (actual) cdf that describes their distribution, then the cdf values associated with each of the program failure rates will be uniformly distributed on the interval $(0, 1)$.

The first step in constructing a U-Plot for a set of data is to estimate the model

parameters based on the observed data. Once these estimates are made, the cdf value associated with each of the observed program failures rates is calculated. This sequence of values is sorted in ascending order and plotted in a step-wise fashion along the X axis of an X-Y plot. If the cdf based on the estimated parameters is close to the actual cdf of the program failure rates, then all of the points in the U-Plot should fall near the line defined by the equation $y = x$. In general, the closer the plotted points are to the line $y = x$, then the better the model's predictive ability.

From the above description of the U-Plot, it is obvious that the use of this tool assumes that all of the program failure rates are drawn from one distribution that is described by the model parameters. The Order Statistics Based Software Reliability Model proposed in this dissertation assumes that each program failure rate is equal to the sum of the fault failure rates for the faults remaining in the software. Further, each of these fault failure rates is assumed to be drawn from a different distribution, as described in the previous chapter. Therefore, the U-Plot is not suited for use with the Order Statistics Based Software Reliability Model presented in this dissertation.

In order to measure the predictive accuracy of a software reliability model an OP Plot was used instead of a U-Plot. The OP Plot is described in the next section.

9.2.2 The OP Plot

The term OP Plot is short for Observed vs Predicted Plot. A single OP Plot shows the performance of a single model with respect to a single data set. The OP plot is an X-Y plot showing the relationship between the observed failure rates and the predicted failure rates for a given run. Observed quantities provide the values for the independent (X) variable and the model's predictions provide the values for the dependent (Y) variable. If the model in question provides perfect predictions, then all of the points in the the OP Plot will lie on the line defined by the equation $x = y$. The closer the resultant plot is to this ideal, then the better

the model's predictive ability. Since the failure rates plotted on the OP Plot will span mulitiple orders of magnitude, a log scale was used for the plot axes.

The OP Plot provides a qualitative evaluation of model prediction, which is useful in detecting patterns of strength and weakness. For example, by using the OP Plot, it is easy to determine if a given model appears to be overly optimistic or overly pessimistic about the program failure rate.

The sum of the squared error for each (observed,predicted) pair in the OP Plot is used to compare the performance of the models and to provide a quantitative evaluation of model prediction.

9.2.3 Comparing The Best Fit Of Each Model

Another metric for evaluation of model performance is derived from the best fit for each model for each data set in its entirety. The sum of the squared error for the fit of each model to each data set was recorded and compared. A high error value for a software reliability model for a given data set may indicate that the distribution used by reliability model is not able to adequately span the full range of the observed failure rates.

9.2.4 Comparing Model Stability

During the course of generating the OP Plot, predictions were made by each model as additional data points (failure rates) were incrementally added to the data set. The stability of the model parameter estimates during the incremental predictions is an indicator of how stable the model is. The main concern here is that a software reliability model should not be overly sensitive to fluctuations in the data set. It is not desirable, for example, for a software reliability model to abruptly and drastically change its estimate for the total number of faults in a program when new data is introduced into the failure set. The underlying distribution being used by the model should be flexible enough to accomodate new data as part of a distribution instantiation with parameters similar to the one estimated before the new data was introduced.

9.3 Development Of Analysis Tools

During the course of this research, a program was developed to implement all of the required software reliability models. This software was developed in C++ and the fitting routines utilize the least squares approach for estimating model parameters. We selected the least squares approach to parameter estimation instead of the more frequently used maximum-likelihood method because of the nature of the expected inputs to our model. Obtaining a solution to a maximum-likelihood method system usually requires that all observed values will be of the same type. However, in the order statistics model a combination of program failure rates and fault failure rates will be used as input. By using the least squares approach, we were able to combine these different types of input by weighting them. This combination would probably be mathematically intractable under maximum likelihood.

This software includes functionality to output the data required to conduct each of the three types of analysis described above.

This program takes a text file containing a set of failure rate data as input and makes software reliability predictions for a specifed model. This tool was used to provide the results presented in later chapters.

Chapter 10

Applying The Model To Representative Data

This chapter details the initial validation of the Order Statistics Based Software Reliability Model. The basic purpose of this experiment was to put the OS-Model through a "sanity-check" to make sure that its predictions were at least in the ballpark with existing models. This chapter discusses the experiment and presents the results that were obtained.

10.1 Data Set Selection

This experiment was intended to perform initial validation of the Order Statistics model. As such, the intention was to use pre-existing failure data, since the use of such data would save the significant overhead of performing testing and debugging at this point in the research process. However, it is rarely easy to find such data, a problem documented by other researchers [12] [13]. To a large extent, the scarcity of appropriate data can be attributed to the requirements that failure data must meet before being deemed suitable for use for reliability modeling. The following sections detail some of these requirements and describes the data that was ultimately selected for use in this experiment.

10.1.1 The Need For High Quality Data

The first requirement that potential data sets have to satisfy before being selected is that the failure data represented in the set has to have a high level of quality. That is to say, the data should have been meticulously compiled with application to software reliability in mind. Data sets that omit data or include incorrect data are of little use for validating a software reliability model. In fact, inaccurate data may do more harm than good if it leads researchers to draw the wrong conclusions about the models being tested.

There are incidents reported in the literature where shortcomings in existing data sets have compromised the results that the authors wanted to obtain. Jelinski and Moranda [13] report that the vague nature of the trouble reports they were using as a data sources forced them to change the way that their experiment was designed. Musa, as well, reports [19] that ambiguity in fault reports that he had encountered forced him to make simplifying assumptions when several failures were found grouped together in the data instead of being reported individually.

For these reasons, the data selected for this experiment needed to be failure data that had been carefully compiled with software reliability estimation in mind.

10.1.2 The Need For A Substantial Number Of Failures

The second requirement that a potential data set had to satisfy before being selected is that the data set had to consist of a substantial number of failures. This requirement is necessary to insure that the results that are obtained will be statistically sound. It is not realistic to believe that valid statistical results can be obtained from data sets that consist of only a few faults. Small samples are notorious for providing misleading statistical results, because of increased sensitivity to flucuations in individual values.

Another problem with small-sized failure sets is that a small number of failures may indicate that the software system being testing was not an actual large-scale system, but was a toy system written exclusively for research purposes. It is not clear that results from such systems can be directly applied to real-world

situations.

10.1.3 The Need For Failure Data From Representative Testing

At the outset of this experiment, is was obvious that determining how well the model fit the failure data would be difficult to assess in absolute terms. For example, if the model was applied to the data and the best fit had an error measure of some number, E, then how would we know whether this fit was a good fit, or not so good? For this reason, it was necessary to compare the results from the model developed in this dissertation to the results from existing models. Since the selected data sets have to be used with existing models, data obtained during representative testing had to be used.

10.1.4 The Selected Data Sets

The data sets used in this experiment were first published by Musa[19]. These failure sets were gathered from several real-life software systems and were compiled with software reliability application in mind. In addition to the data sets, Musa [19] describes how the data was gathered and what assumptions were made.

10.2 Experiment Design

Four models and four data sets were used in this experiment. The desired analysis was performed for each model for each data set, for a total of 16 runs. Each run was an iterative fitting process consisting of the multiple iterations necessary to generate an OP-Plot.

For each run, the iterative process was started by setting the number of program failure rates used for the fitting process on the first iteration to be half of the total number of observed program failure rates. The software reliability tool was used to predict the current program failure rate. This predicted value was compared to the next program failure rate that was actually observed in the data

set. This process was repeated by adding one additional value to the observed quantities, until the entire data set had been considered. The resulting sequence of observed and predicted values was used to generate the OP-Plot.

The performance of each model on an OP Plot is very important because the iterative process involved in generating an OP Plot mirrors the process that a test team will follow when conducting reliability testing. The team would typically find a fault, fix it, and obtain a program reliability estimate based on the available data. This process would be repeated until testing is completed.

A OP Plot was generated for each of the 16 runs in the study. In addition to the OP Plot information, data about the best fit and progressive parameter estimates for each model for each data set was recorded.

10.3 Results

This section presents the results obtained for each of the three types of analysis performed in this study.

10.3.1 The Results Of Analyzing The Predictive Accuracy Of The Models

The sixteen OP Plots generated during this experiment are shown in Figures 5 to 20. Just by glancing at the four plots generated for each data set, it is apparent that all of the models seemed to have some trouble generating predictions that were close enough to the observed values to create an OP Plot close to the $x = y$ ideal.

Perhaps the first questions that comes to mind are "Why are all of the models having trouble?" and "Aren't any of them any good?". The following paragraphs will attempt to provide answers to the these questions.

Figure 21 gives the average of the fitting error for each OP Plot. The value in each cell in Figure 21 represents the average of the squared error for each X-Y pair on the corresponding OP Plot. For example, an entry of .000564 is entered

in the table for the JM Model for Set 2. The presence of this number means that if the difference of each X-Y pair on the OP Plot for the Jelinski-Moranda Model for Data Set 2 is squared,normalized, and summed, then the resulting value that is obtained is .000564.

Looking across the rows of this table, it is apparent that all of the software reliablity models performed about same for any given data set. There are no cases where one model performed significantly worse or significantly better than the other models.

A possible explanation for the poor performance of all of the OP Plots is the nature of the data itself. By looking at the observed data series in Figures 22 through 37, the scattered nature of the observed failure rates is obvious. Although the observed values do tend to decrease as the failure number increases, it is obvious that the failure rates are far from strictly decreasing. Since all of the models under consideration in this study (and probably all models in existence) model program failure rates as strictly decreasing, then any exceptions will contribute to poor estimates and predictions. In Figures 22 through 37, it seems that an increase in observed program failure rate from point to point seems as much the norm than the exception. From these observations, it seems that the poor results in the OP Plots have more to do with the nature of the data than the models themselves.

These observations support the discussion in Chapter 2 about the amount of statistical noise inherent in data based on quantities observed during testing, and provides more evidence that the emphasis of reliability data acquisition needs to be changed from quantities derived during testing to quantities derived during debugging whenever possible.

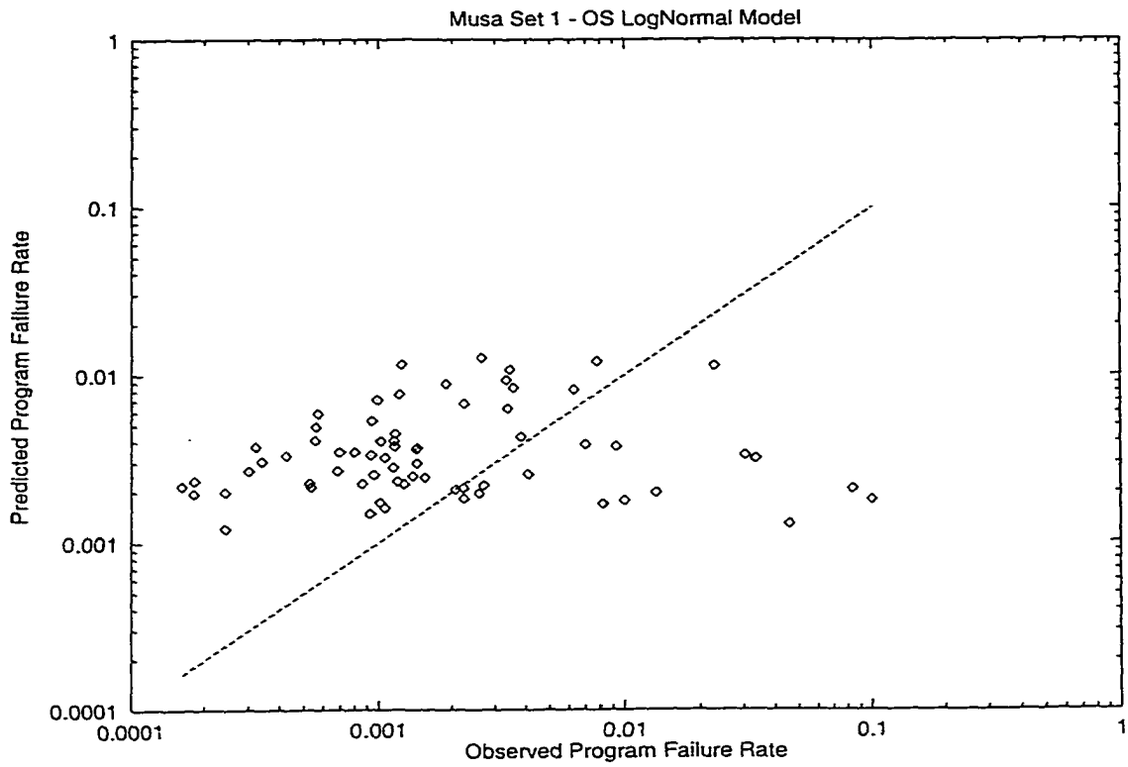

FIG. 5. OS Model OP Plot For Data Set 1

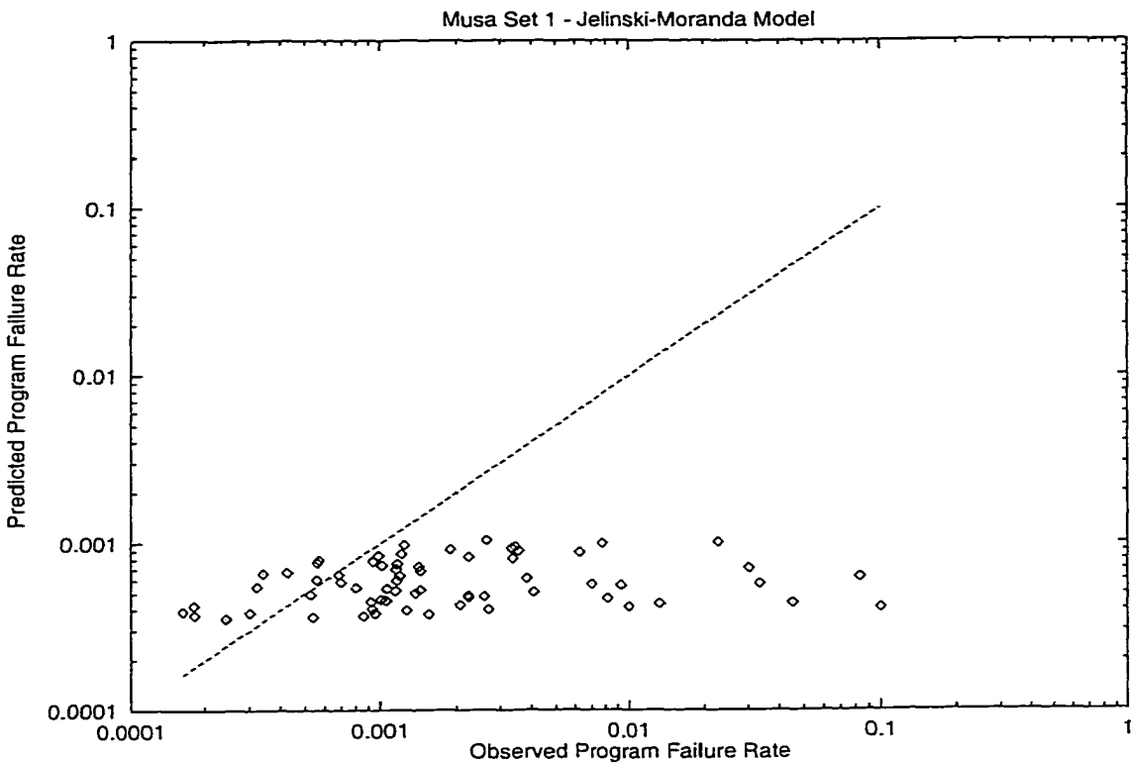

FIG. 6. JM Model OP Plot For Data Set 1

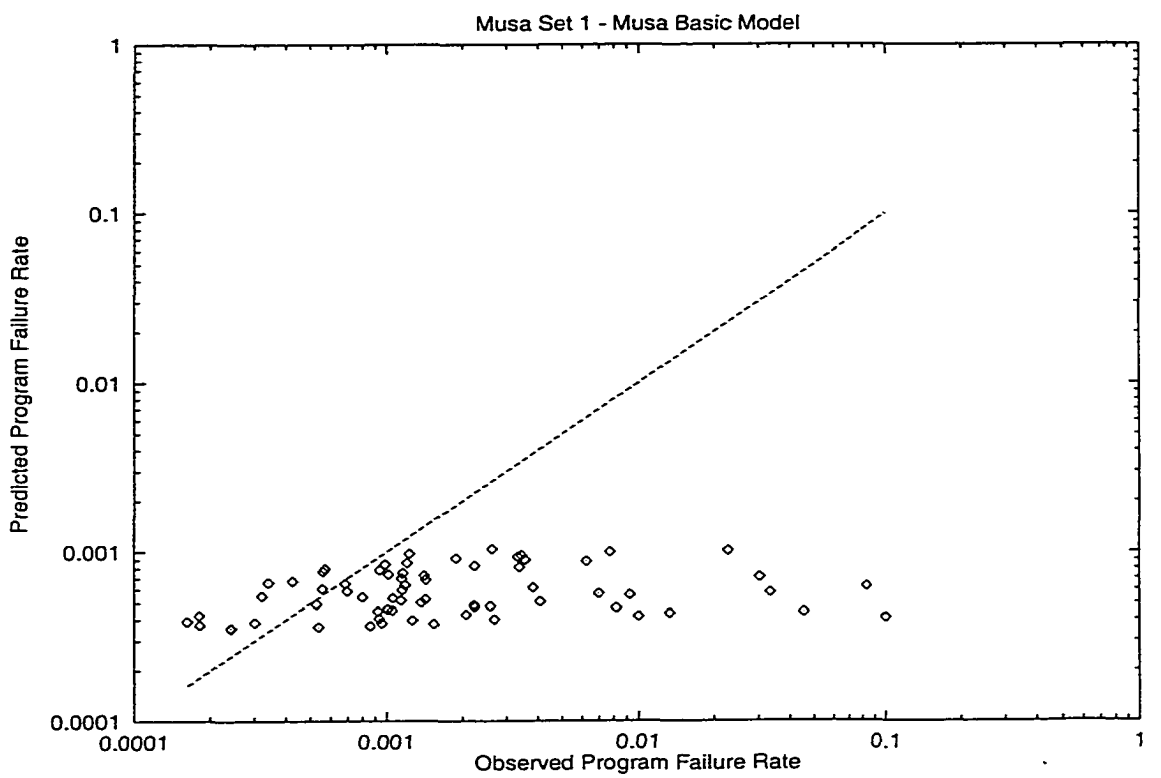

FIG. 7. Musa Basic OP Plots For Data Set 1

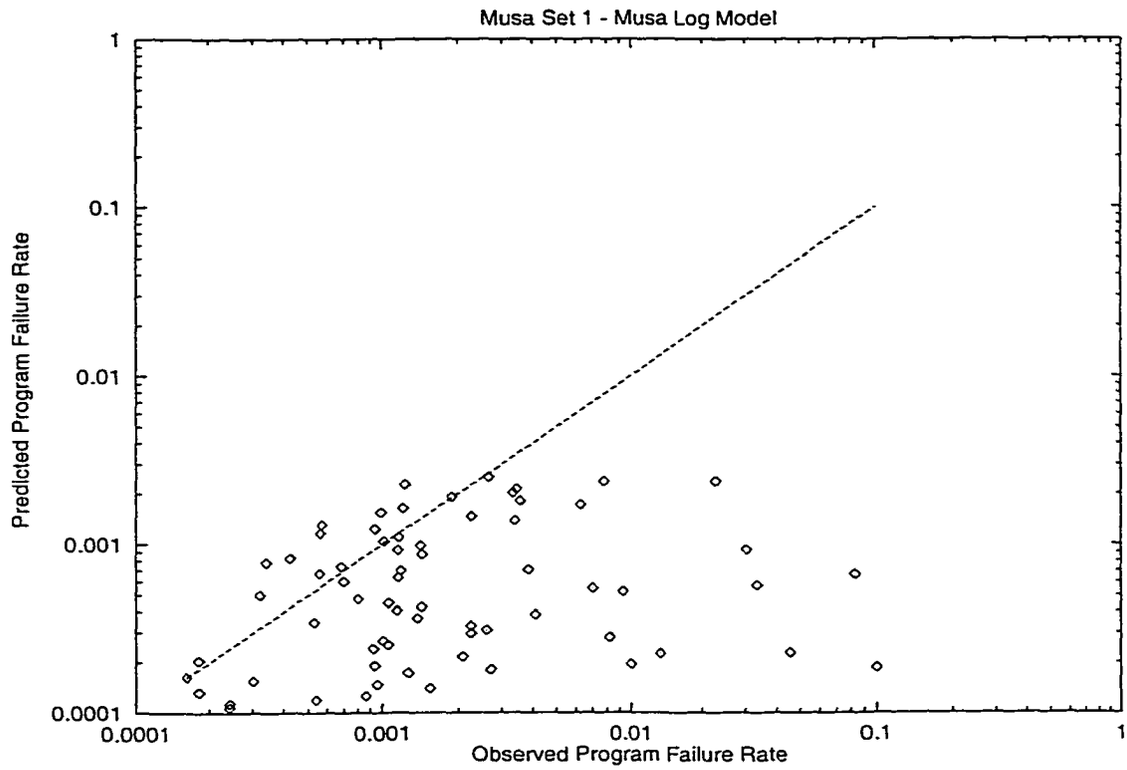

FIG. 8. Musa Log OP Plots For Data Set 1

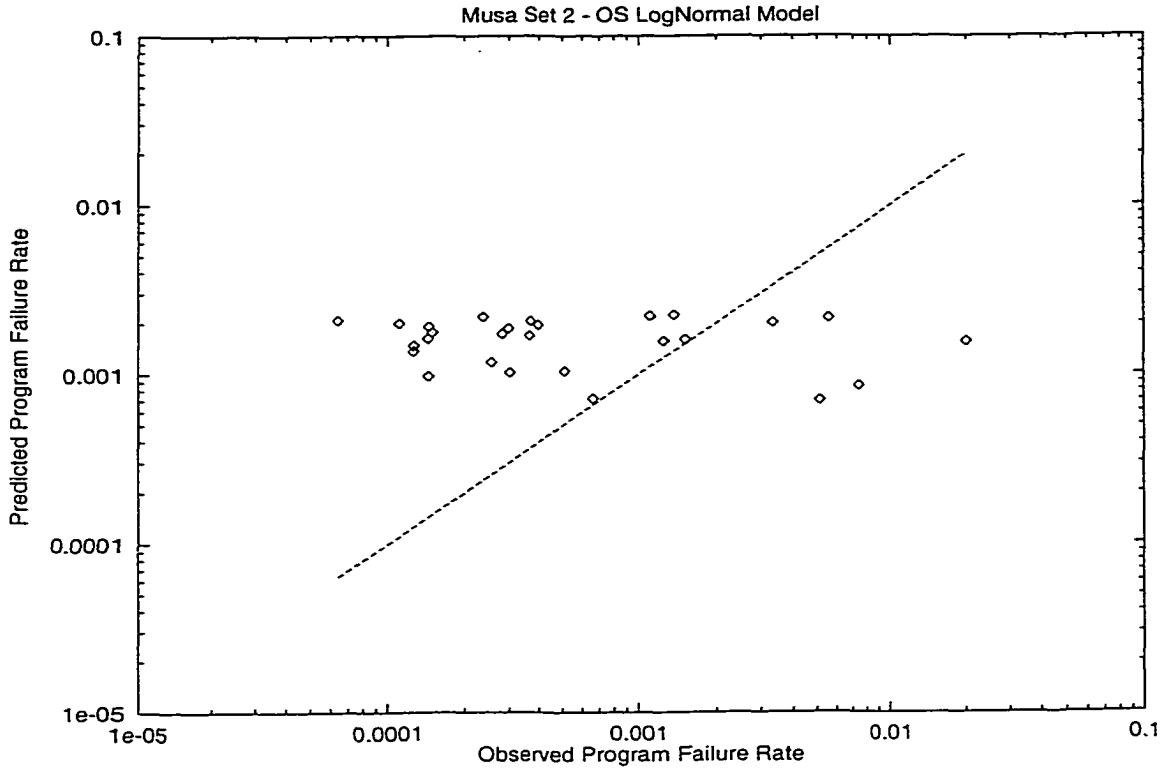

FIG. 9. OS Model OP Plot For Data Set 2

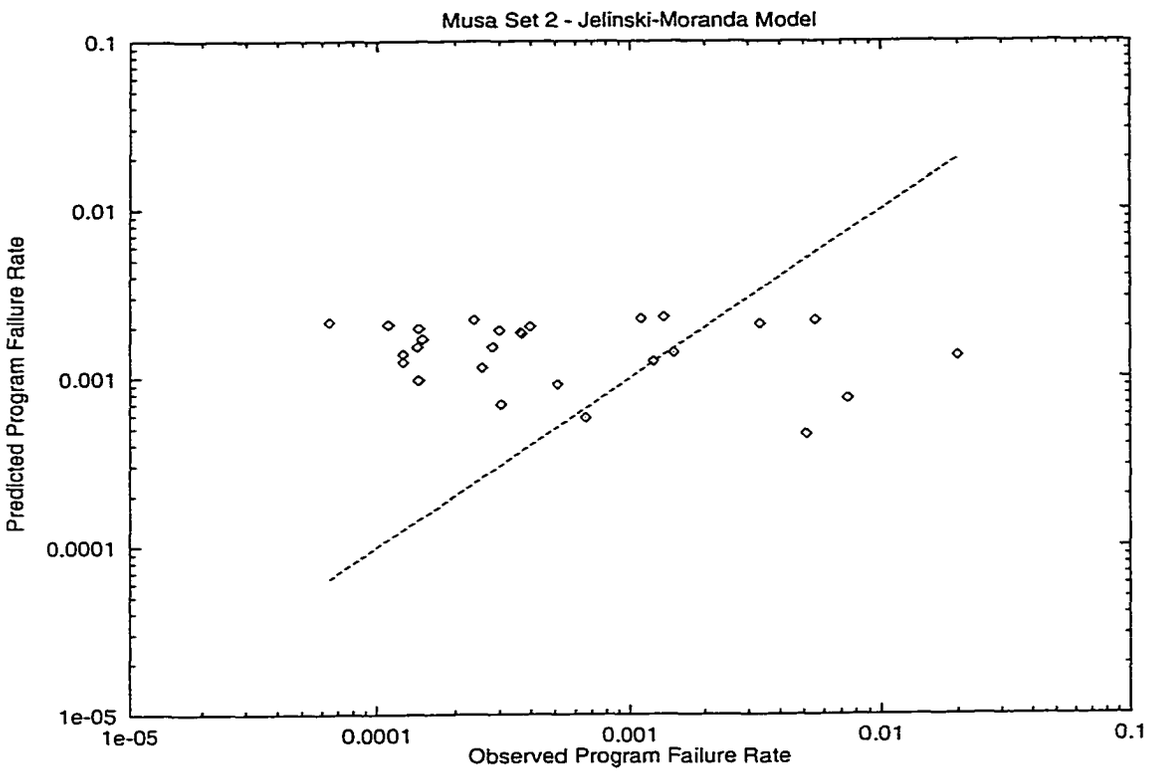

FIG. 10. JM Model OP Plot For Data Set 2

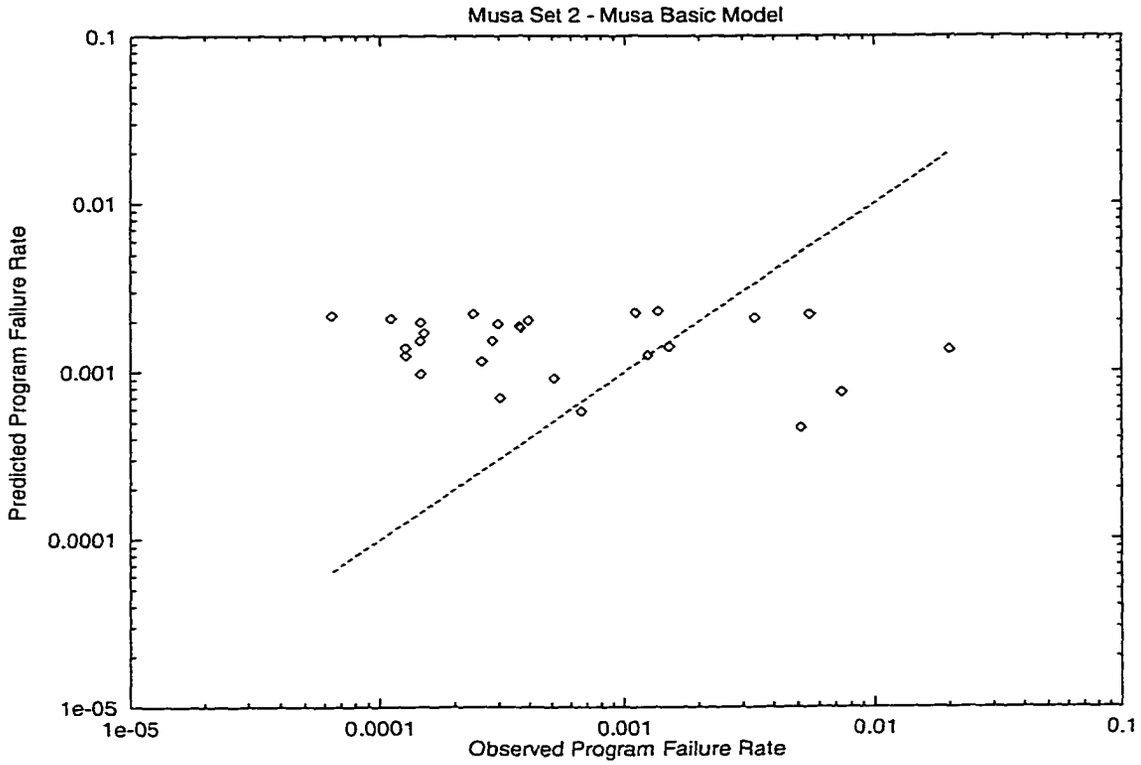

FIG. 11. Musa Basic OP Plots For Data Set 2

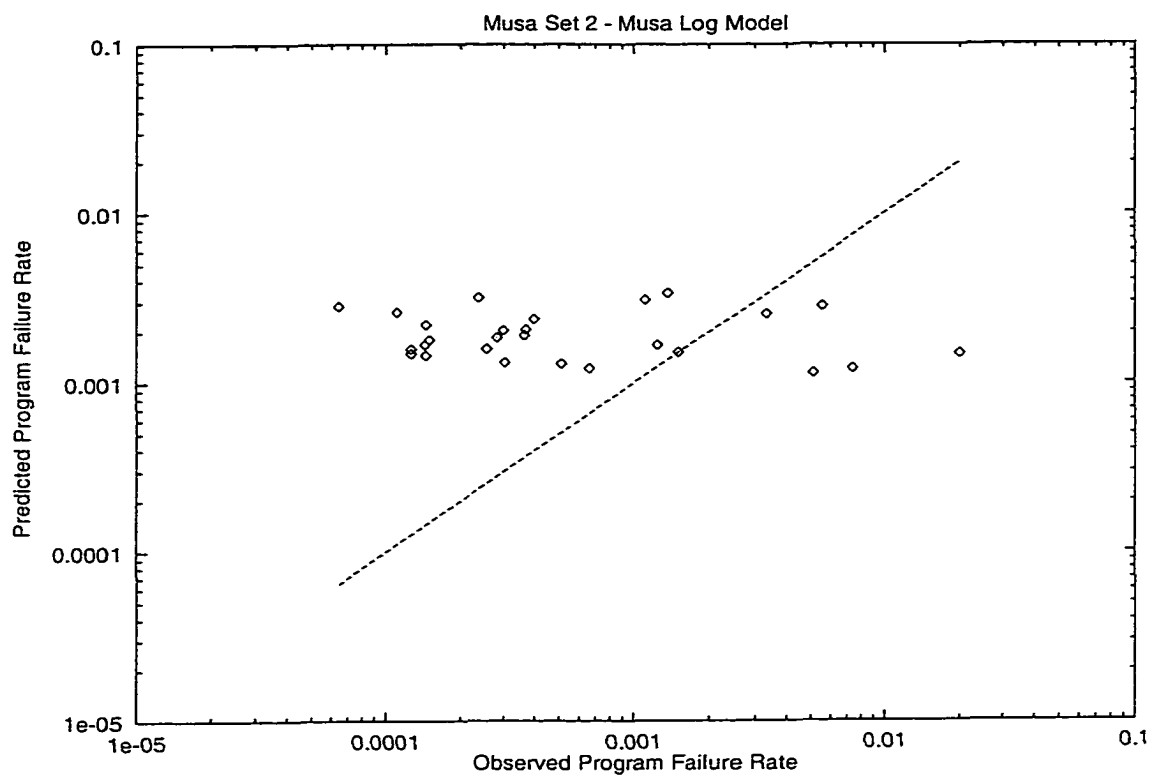

FIG. 12. Musa Log OP Plots For Data Set 2

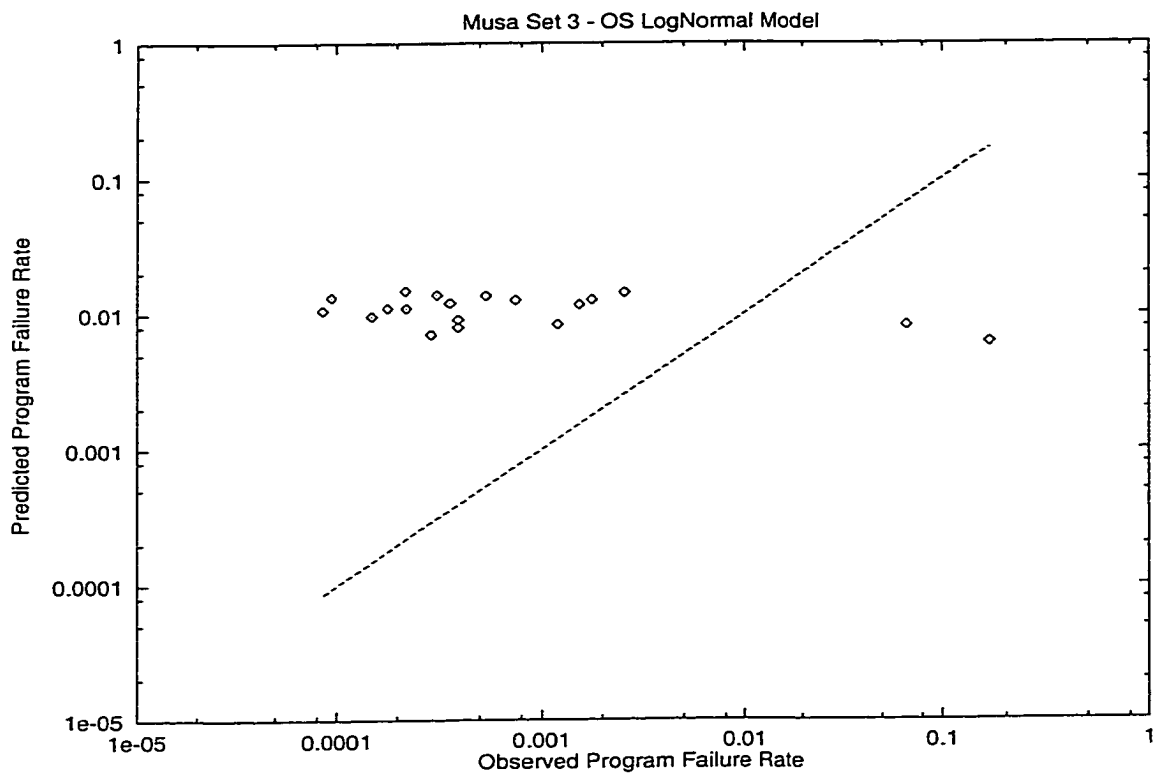

FIG. 13. OS Model OP Plot For Data Set 3

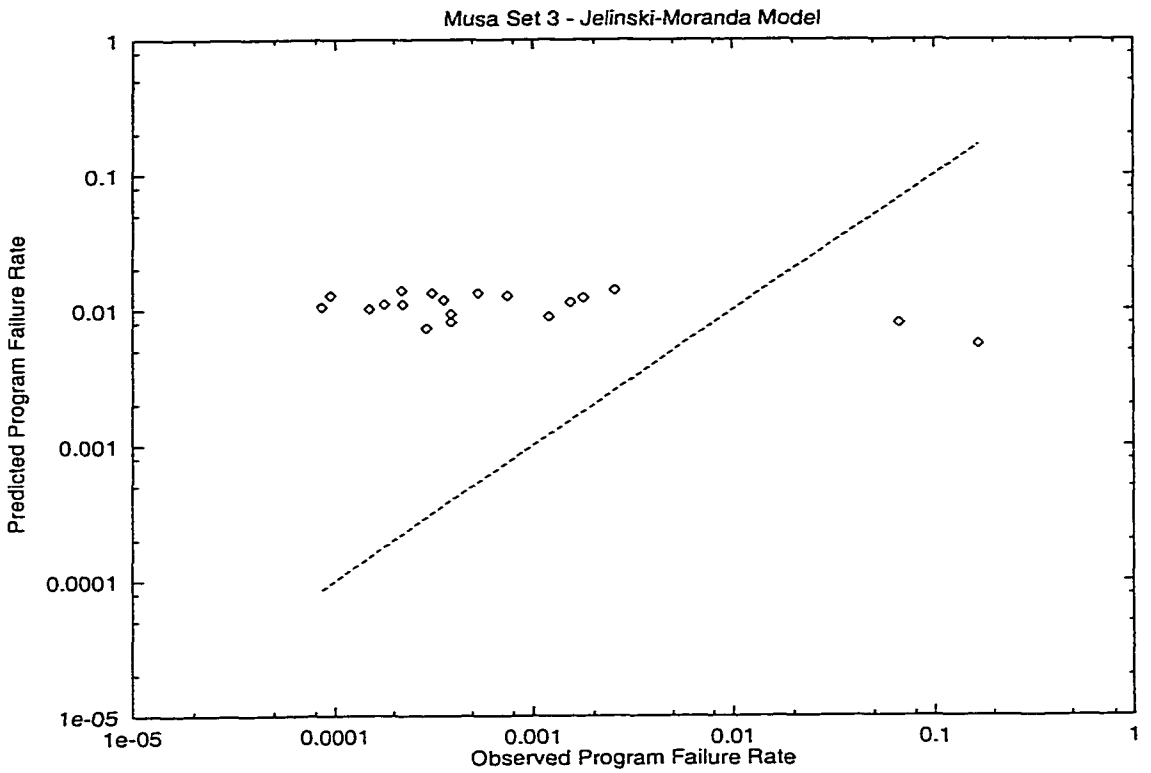

FIG. 14. JM Model OP Plot For Data Set 3

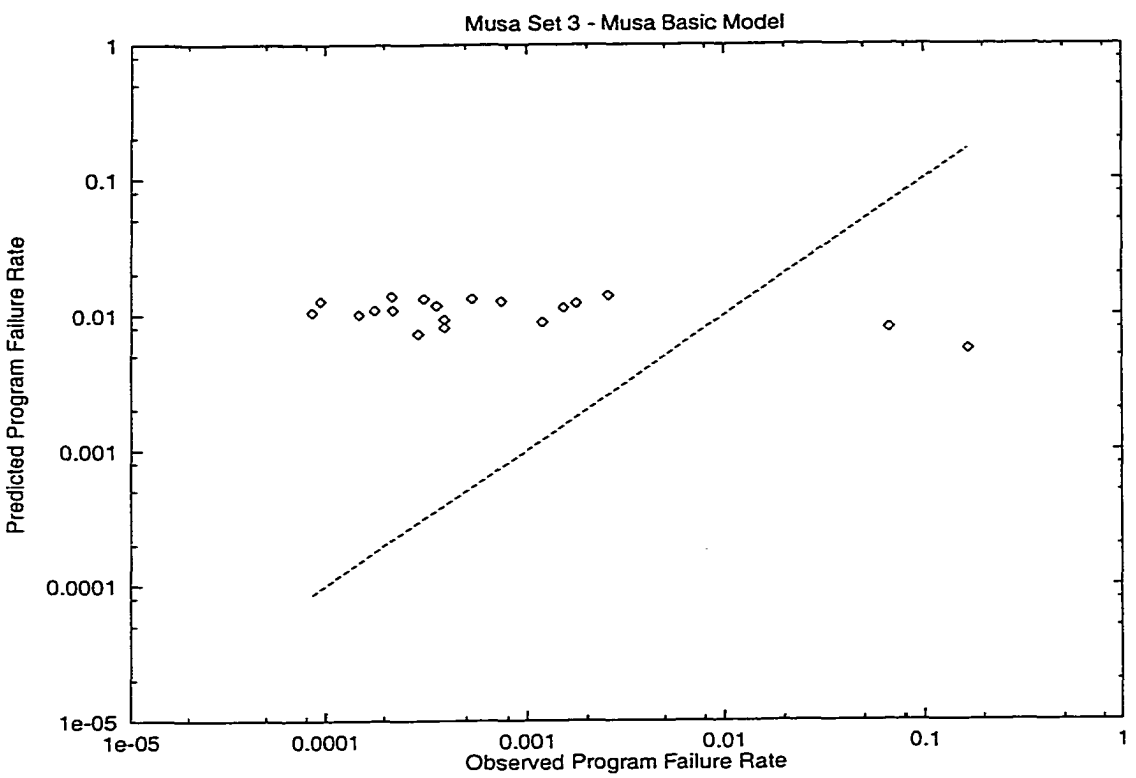

FIG. 15. Musa Basic OP Plots For Data Set 3

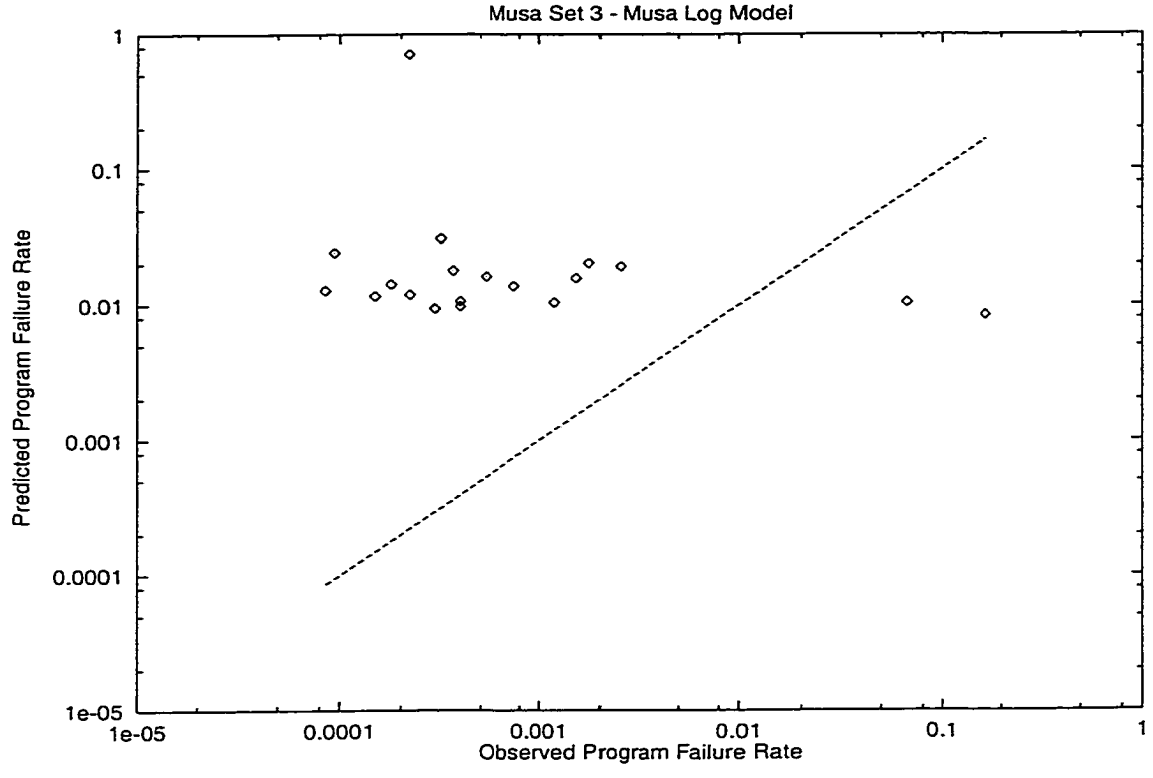

FIG. 16. Musa Log OP Plots For Data Set 3

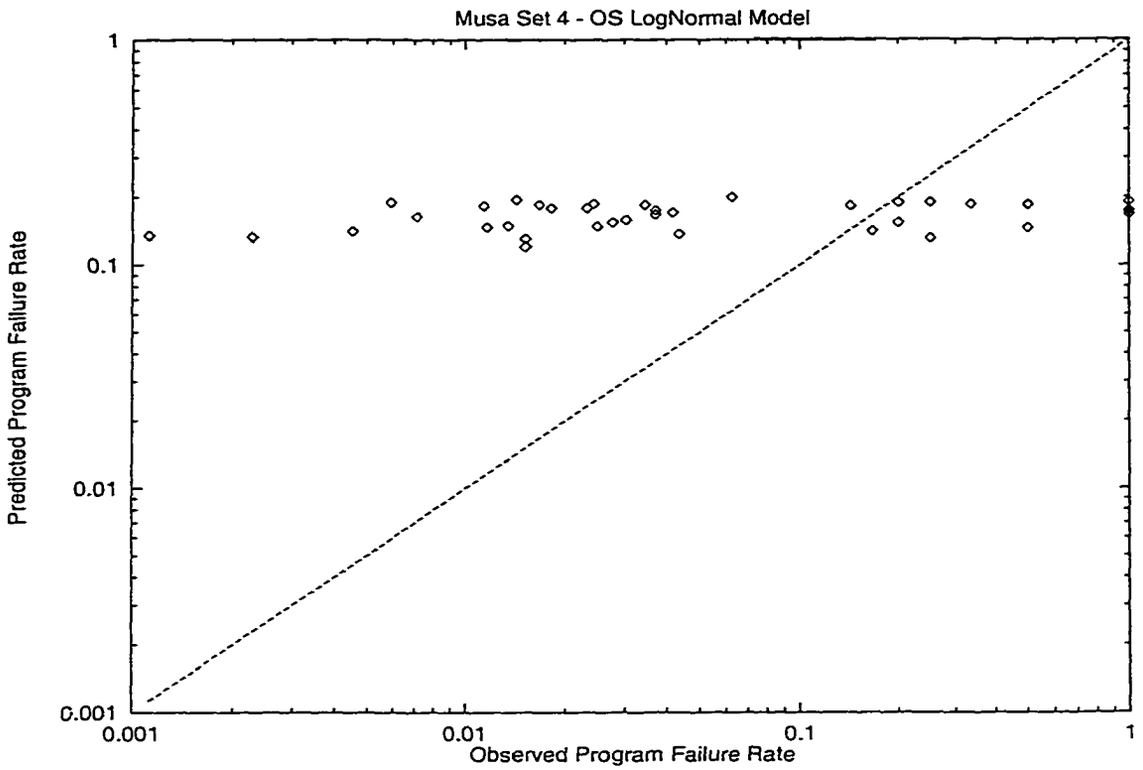

FIG. 17. OS Model OP Plot For Data Set 4

10.3.2 Comparing The Best Fit For Each Model

The sixteen plots of the best fits for each run made during this study are shown in Figures 22 to 37. As pointed out earlier, these plots show the noisy nature of the observed data. Given the observed data in these plots, it is difficult to imagine any distribution that would provide a better fit to the observed data than the fits provided by the models used in this experiment. The data provided in Figure 38 shows that for any one of the data sets, all of the models performed about the same. These observations suggest that no software reliability model (or any other statistical method) would be able to provide really accurate estimates and predictions for this data.

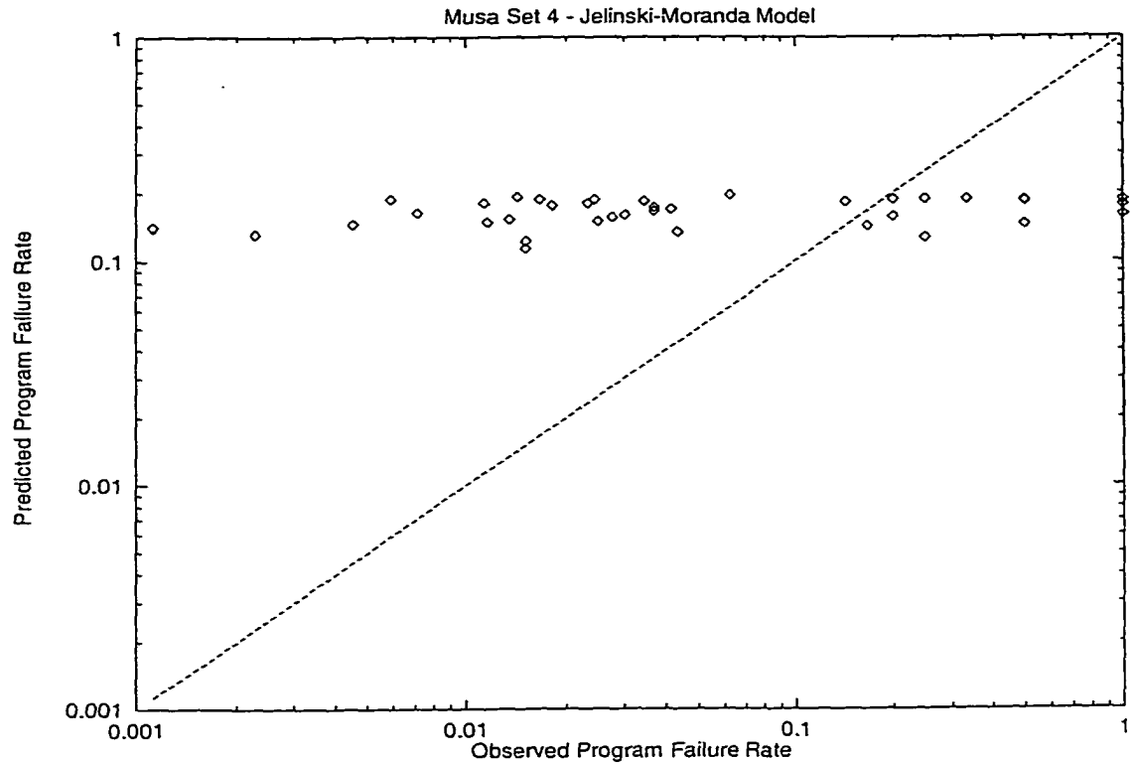

FIG. 18. JM Model OP Plot For Data Set 4

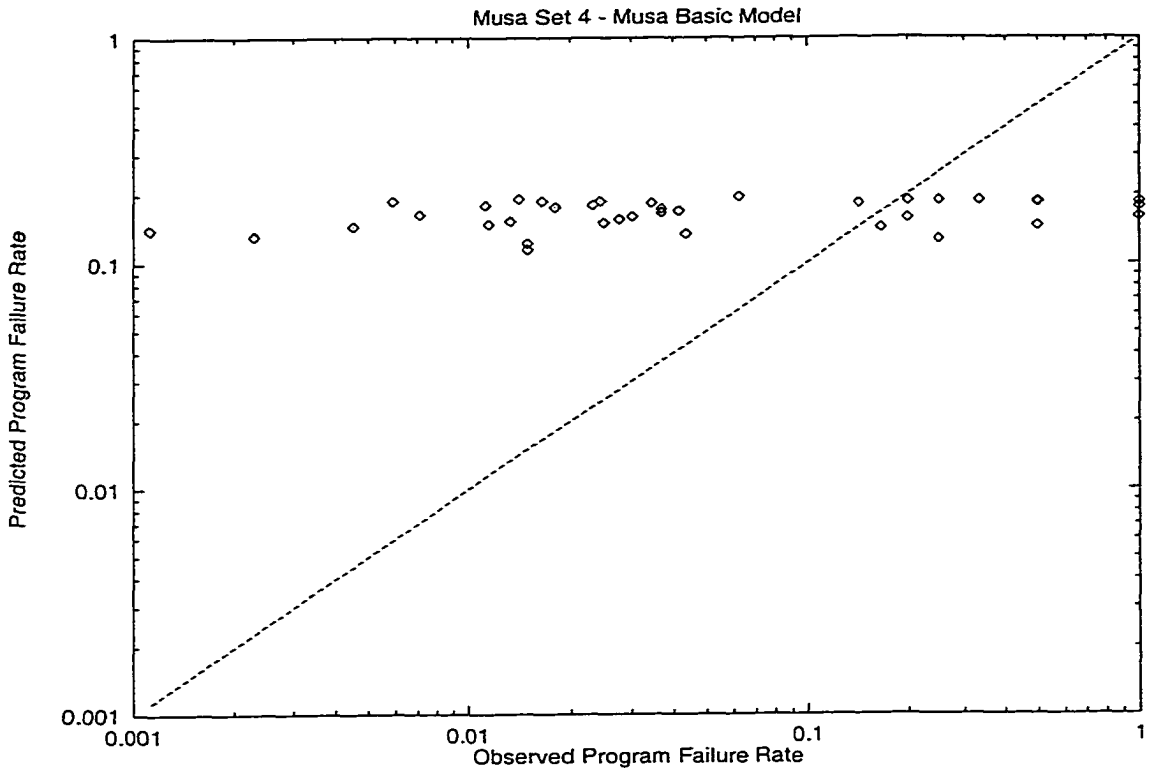

FIG. 19. Musa Basic OP Plots For Data Set 4

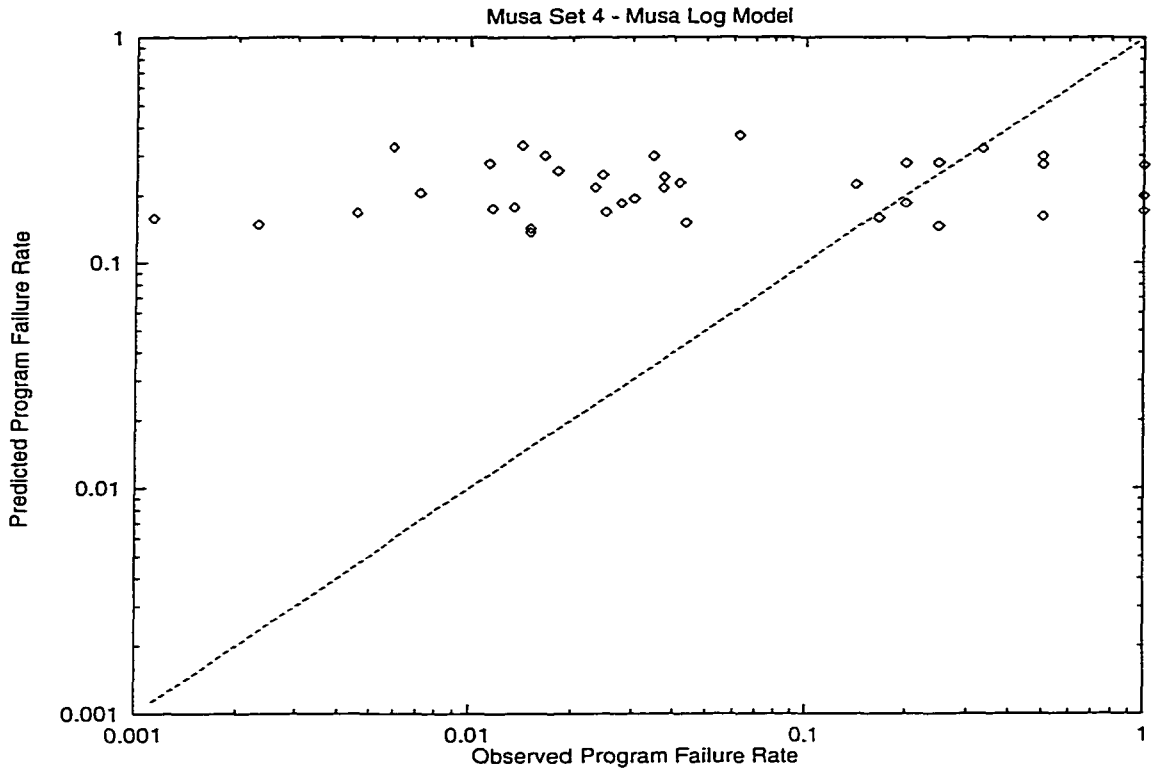

FIG. 20. Musa Log OP Plots For Data Set 4

Model	JM	MB	ML	OS
Set 1	0.424323	0.424323	0.400713	0.452195
Set 2	0.000564	0.000564	0.000563	0.000589
Set 3	0.024337	0.024337	0.023249	0.024303
Set 4	5.621282	5.621283	5.563041	5.733988

FIG. 21. Average Error For Each OP Plot

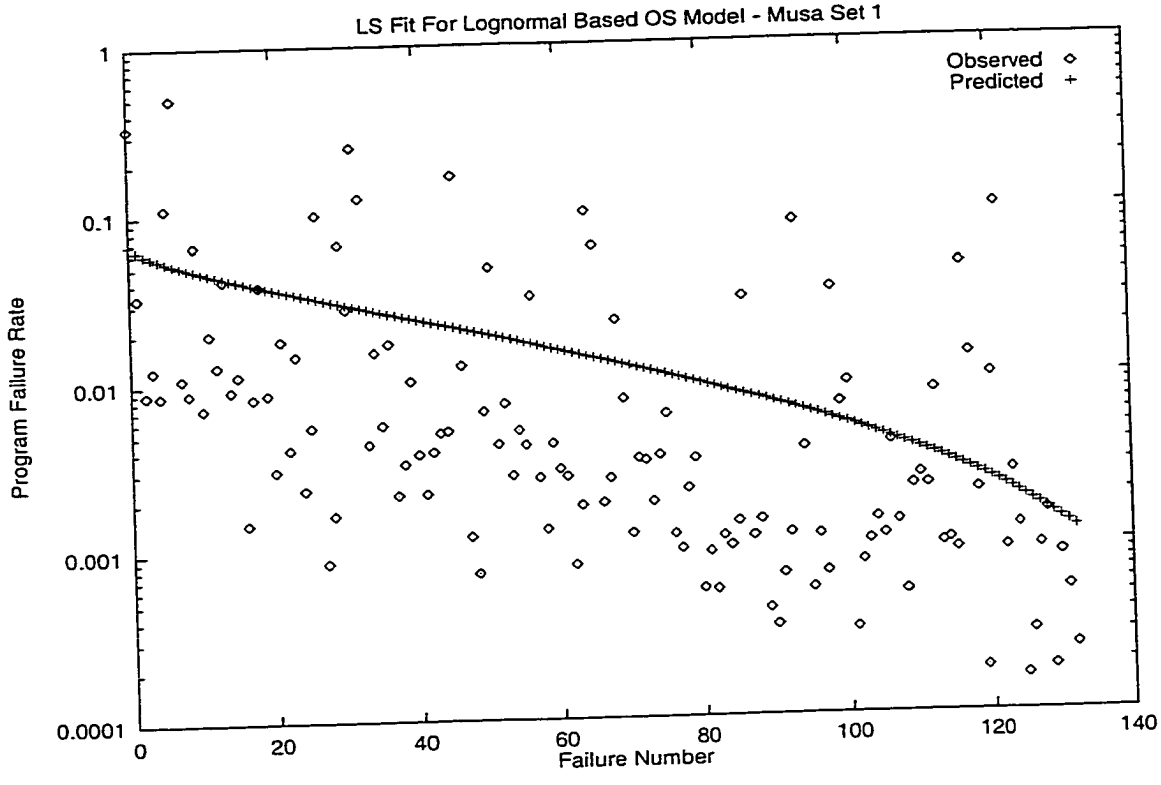

FIG. 22. Best Fits For OS Model Data Set 1

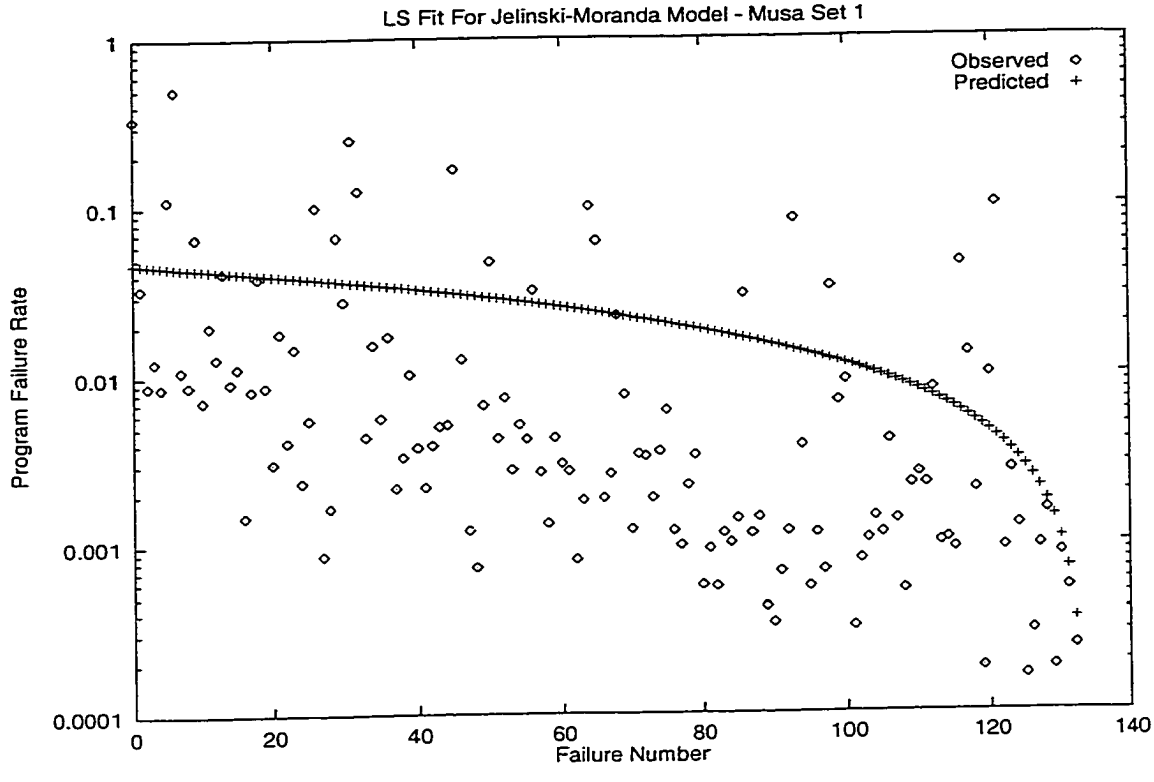

FIG. 23. Best Fits For JM Model Data Set 1

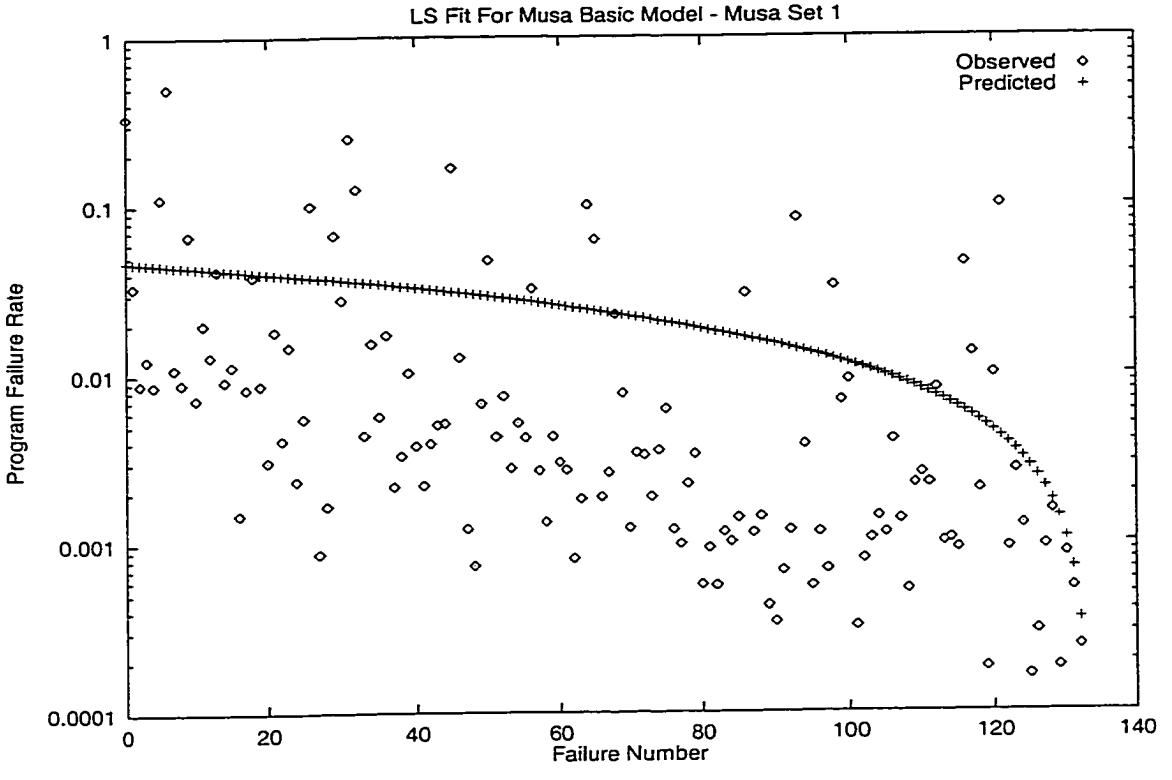

FIG. 24. Best Fits For MB Model Data Set 1

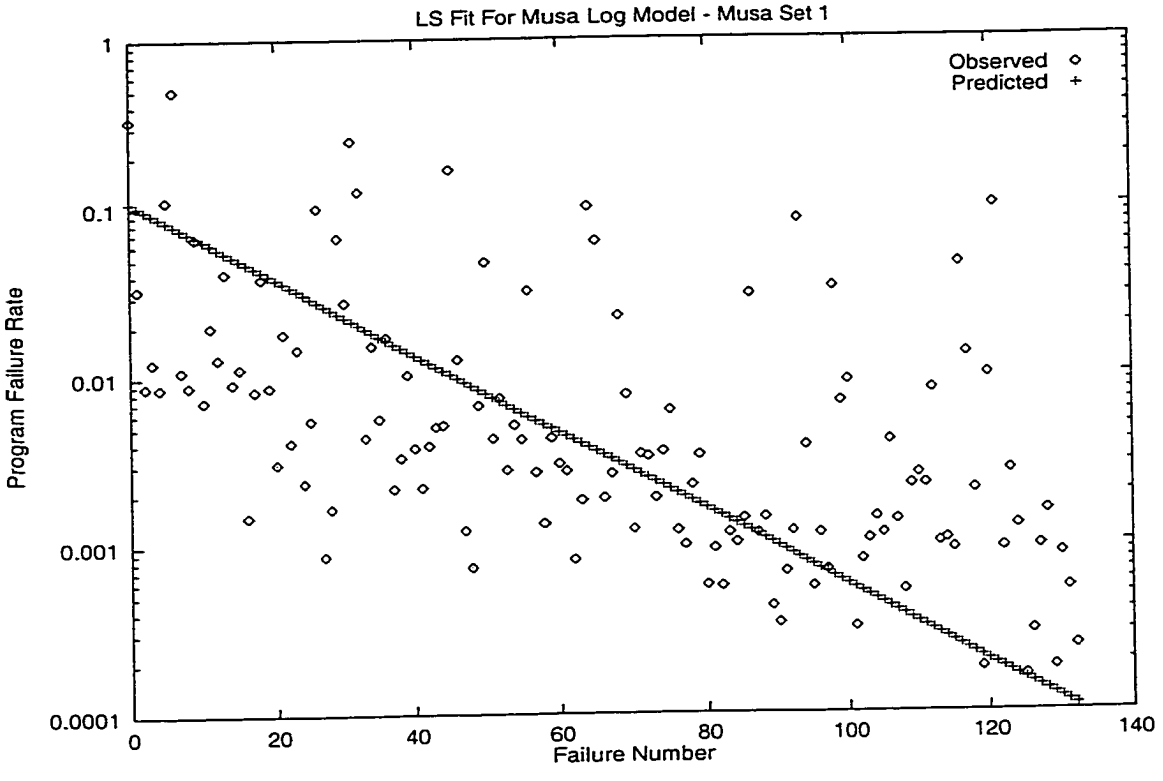

FIG. 25. Best Fits For ML Model Data Set 1

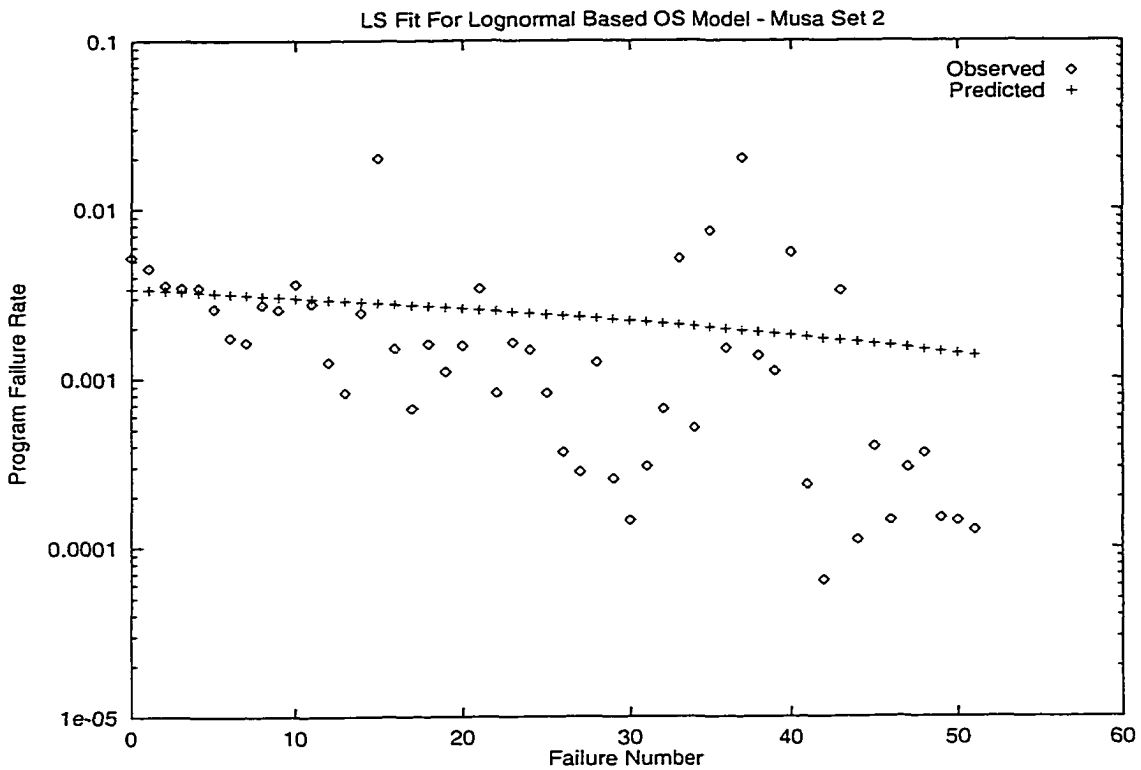

FIG. 26. Best Fits For OS Model Data Set 2

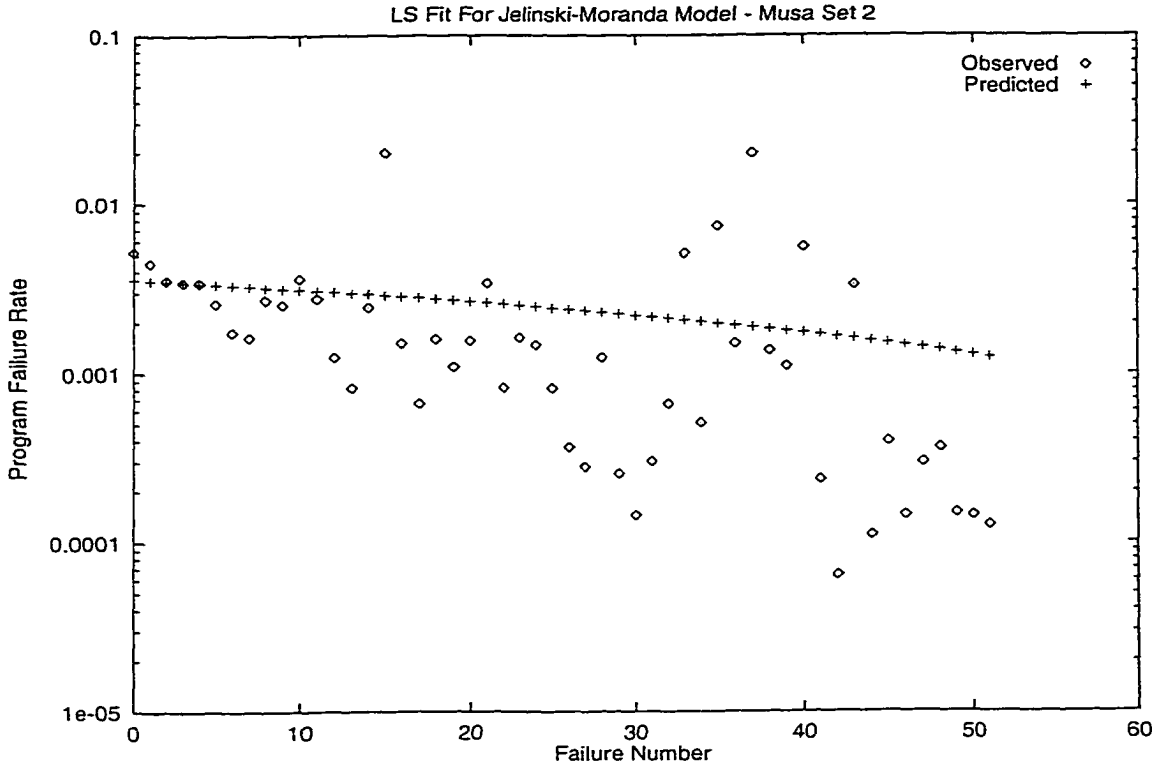

FIG. 27. Best Fits For JM Model Data Set 2

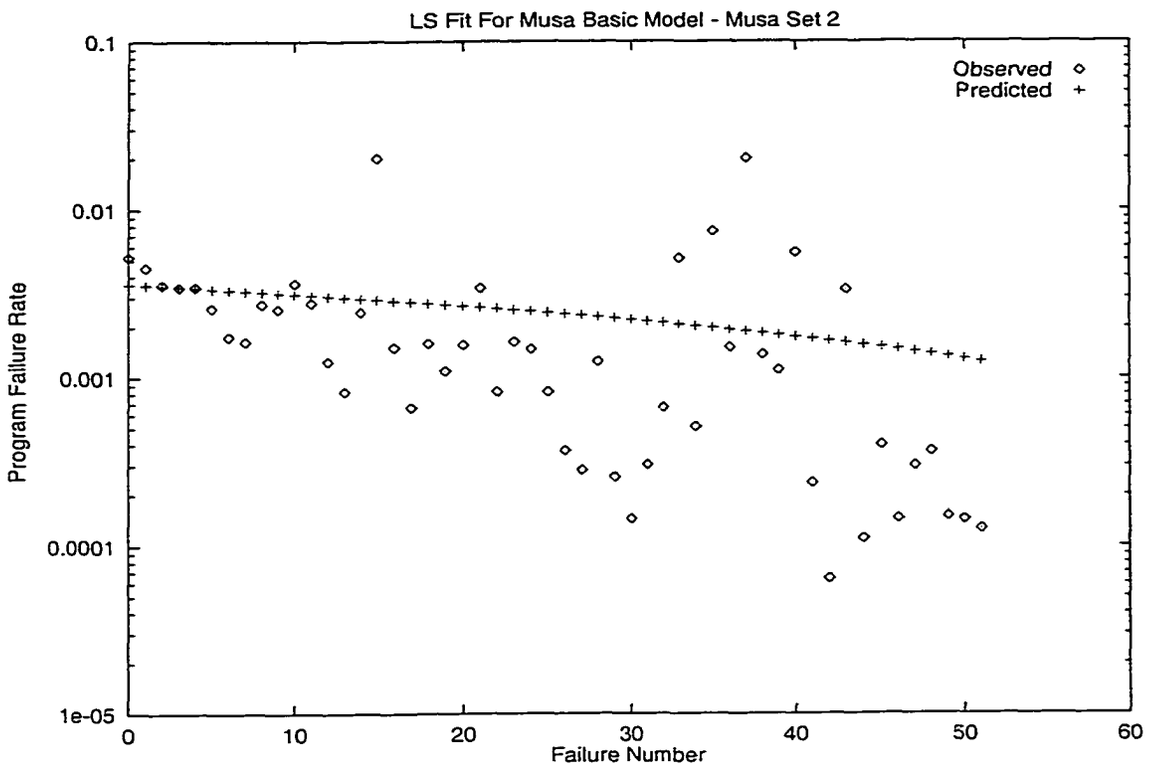

FIG. 28. Best Fits For MB Model Data Set 2

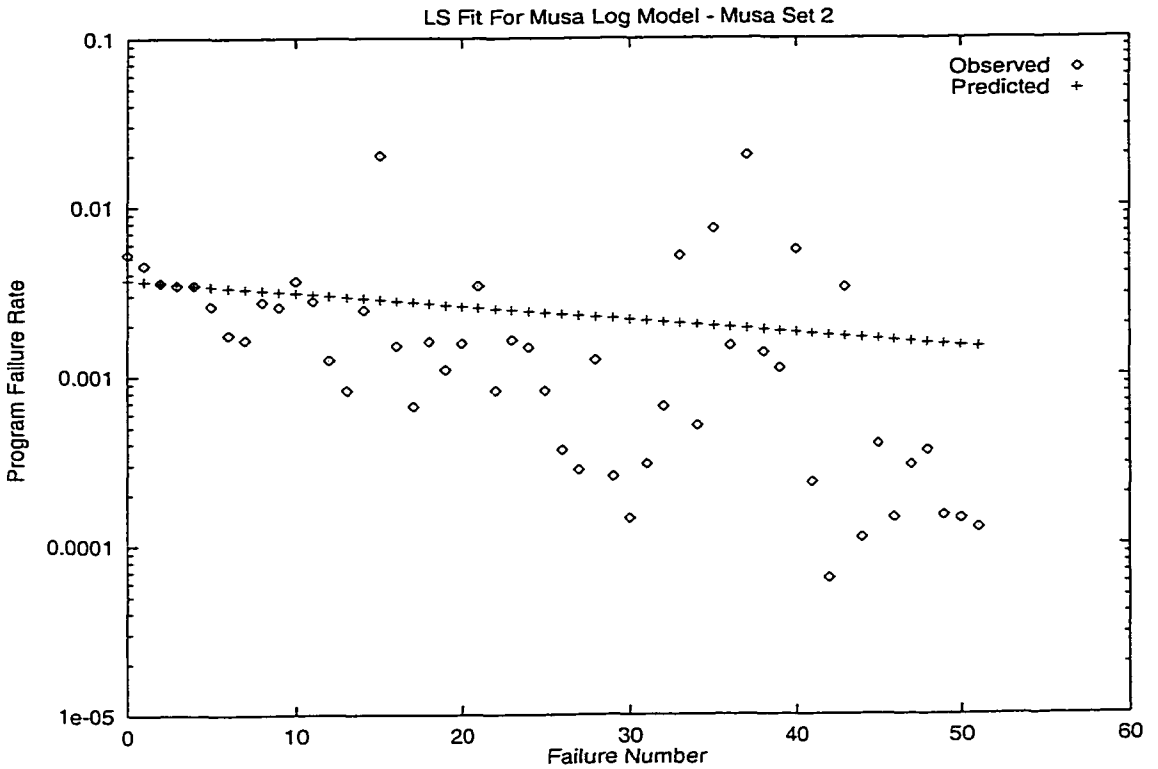

FIG. 29. Best Fits For ML Model Data Set 2

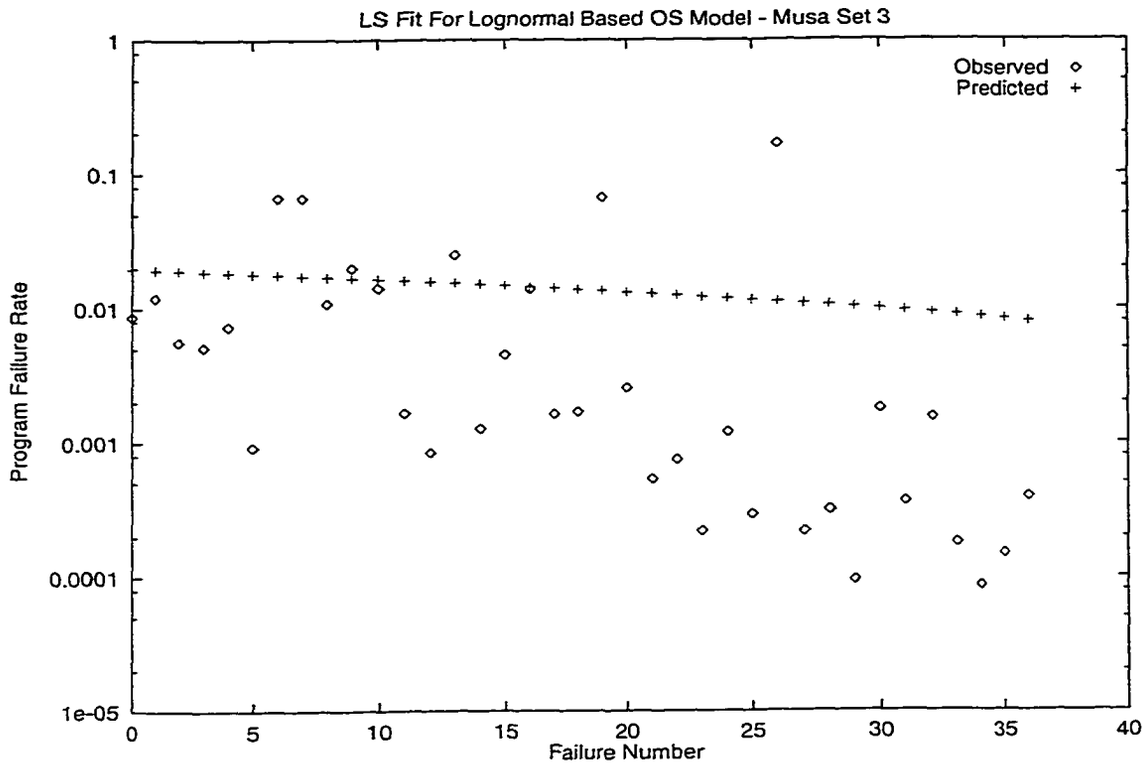

FIG. 30. Best Fits For OS Model Data Set 3

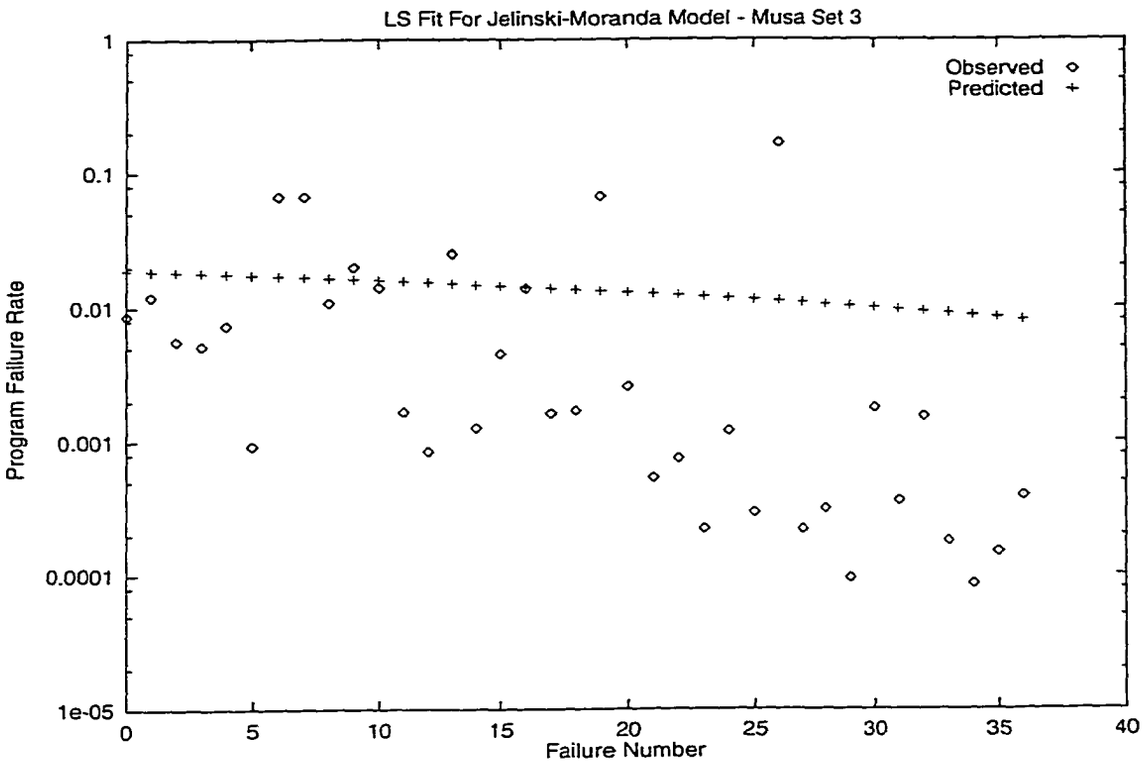

FIG. 31. Best Fits For JM Model Data Set 3

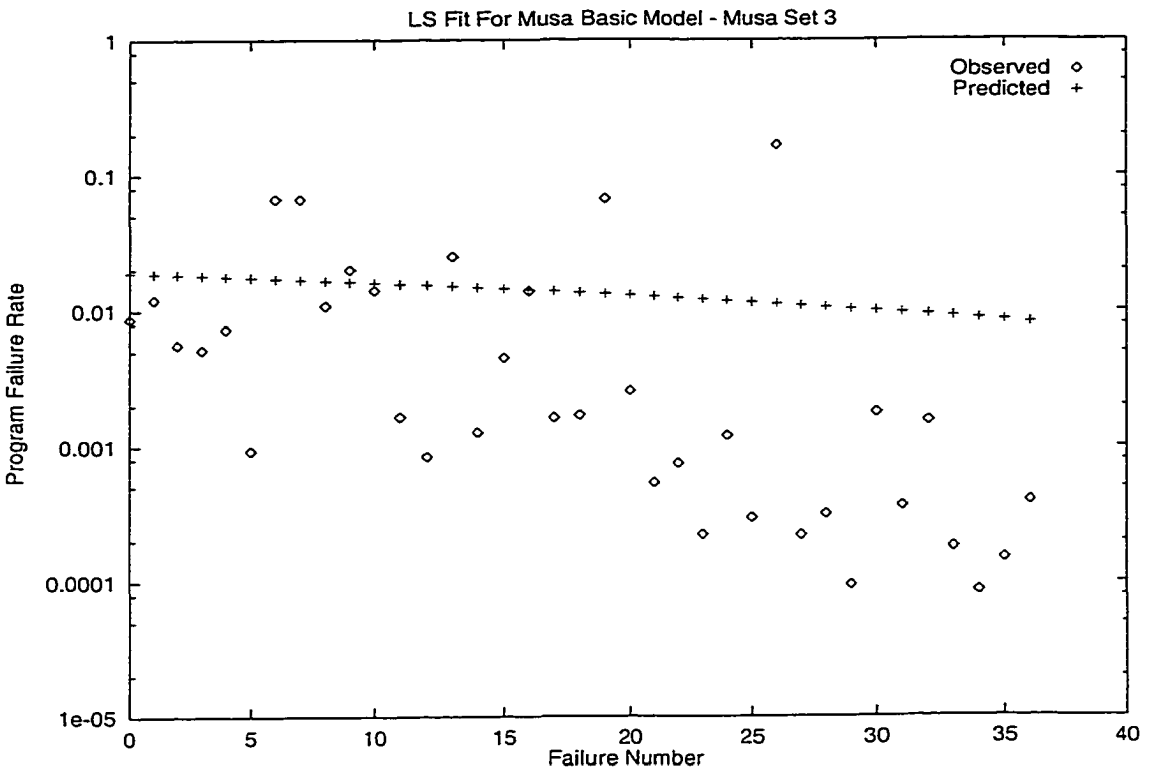

FIG. 32. Best Fits For MB Model Data Set 3

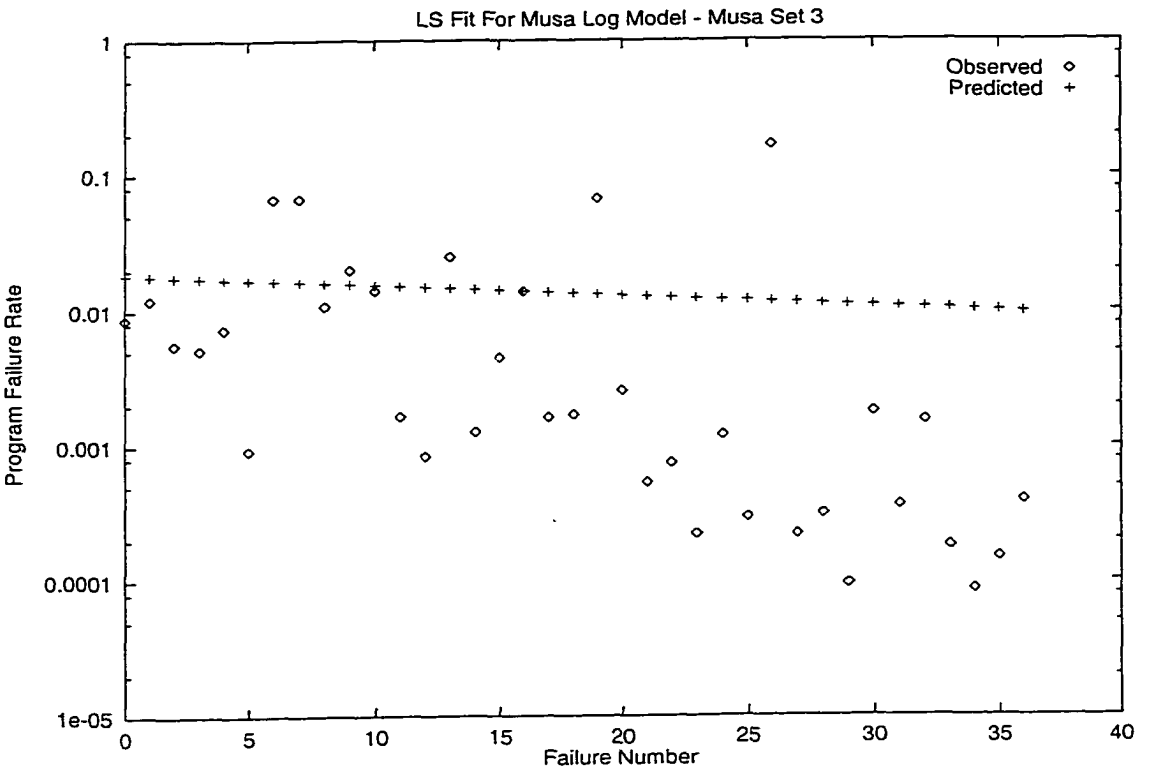

FIG. 33. Best Fits For ML Model Data Set 3

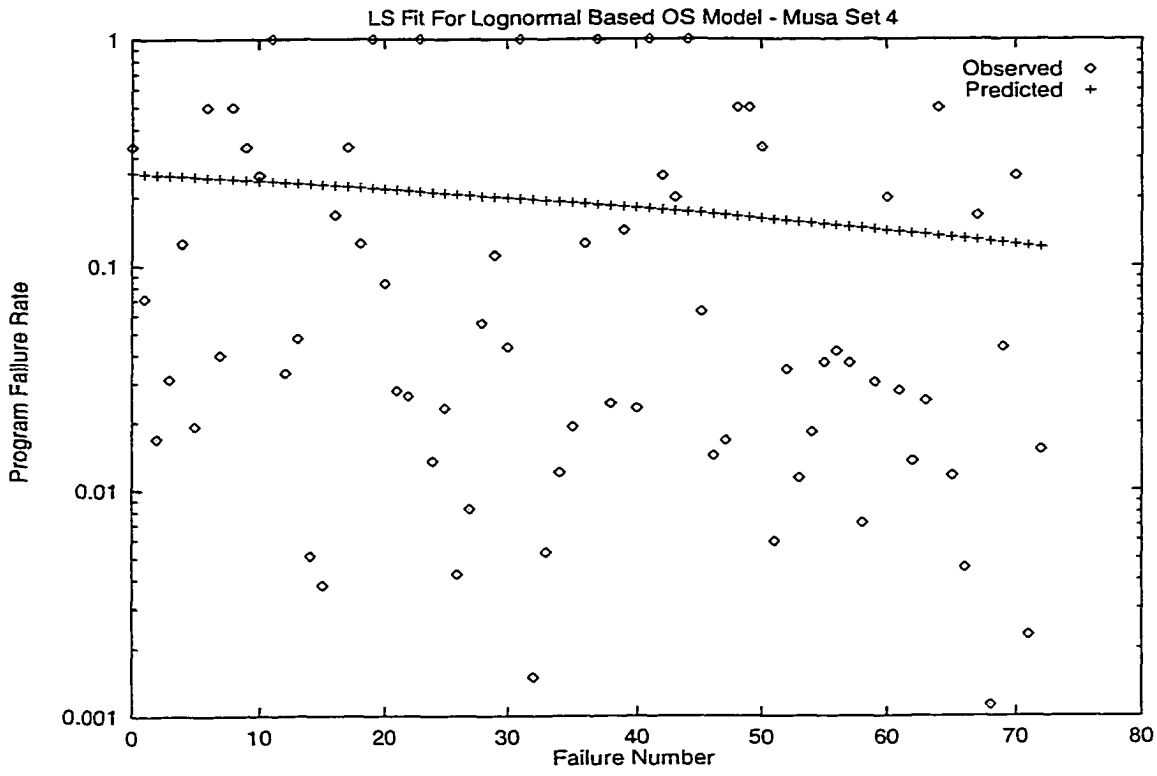

FIG. 34. Best Fits For OS Model Data Set 4

10.3.3 Comparing The Parameter Progression For Each Model

The plots of the parameter progression for each parameter for each model during the generation of the OP Plot is shown in Figure 39 to 47.

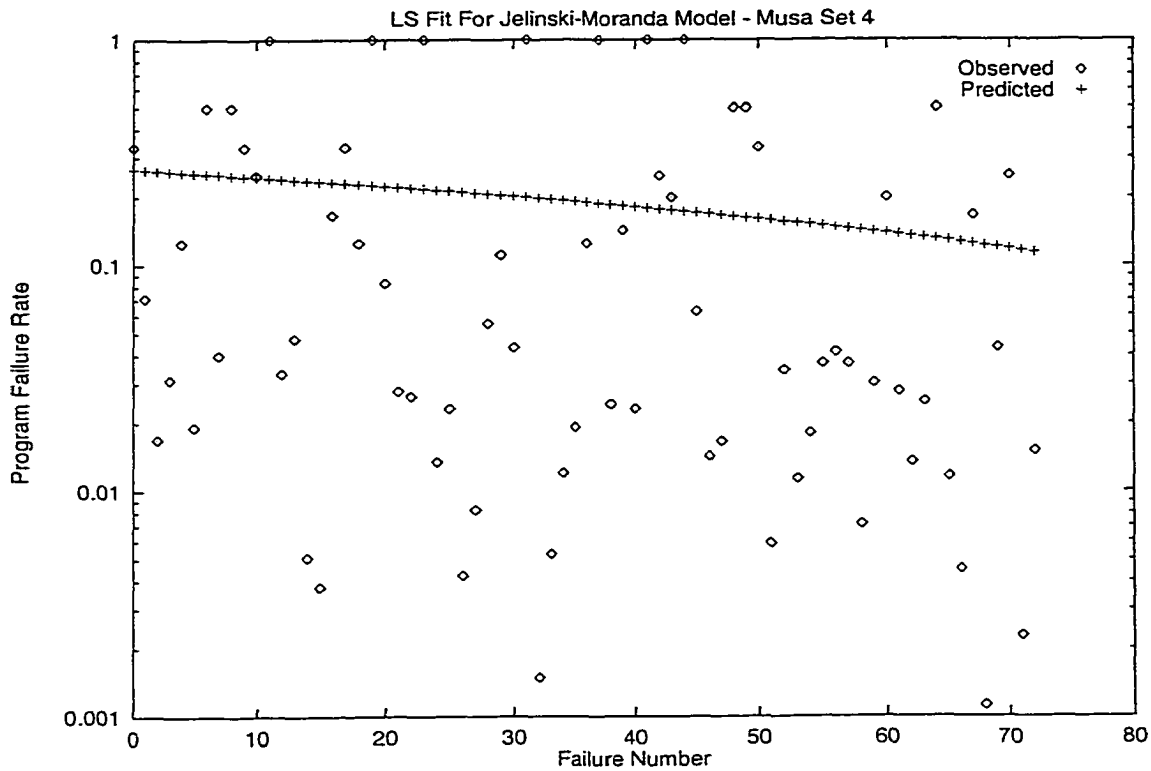

FIG. 35. Best Fits For JM Model Data Set 4

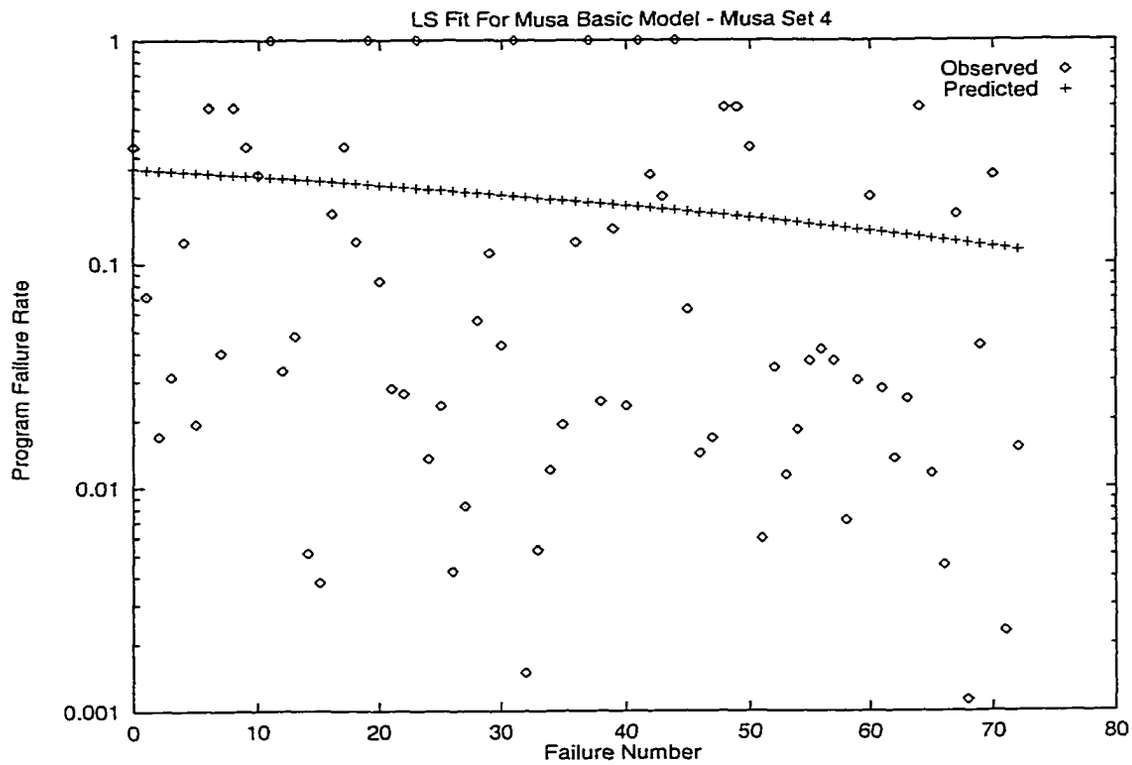

FIG. 36. Best Fits For MB Model Data Set 4

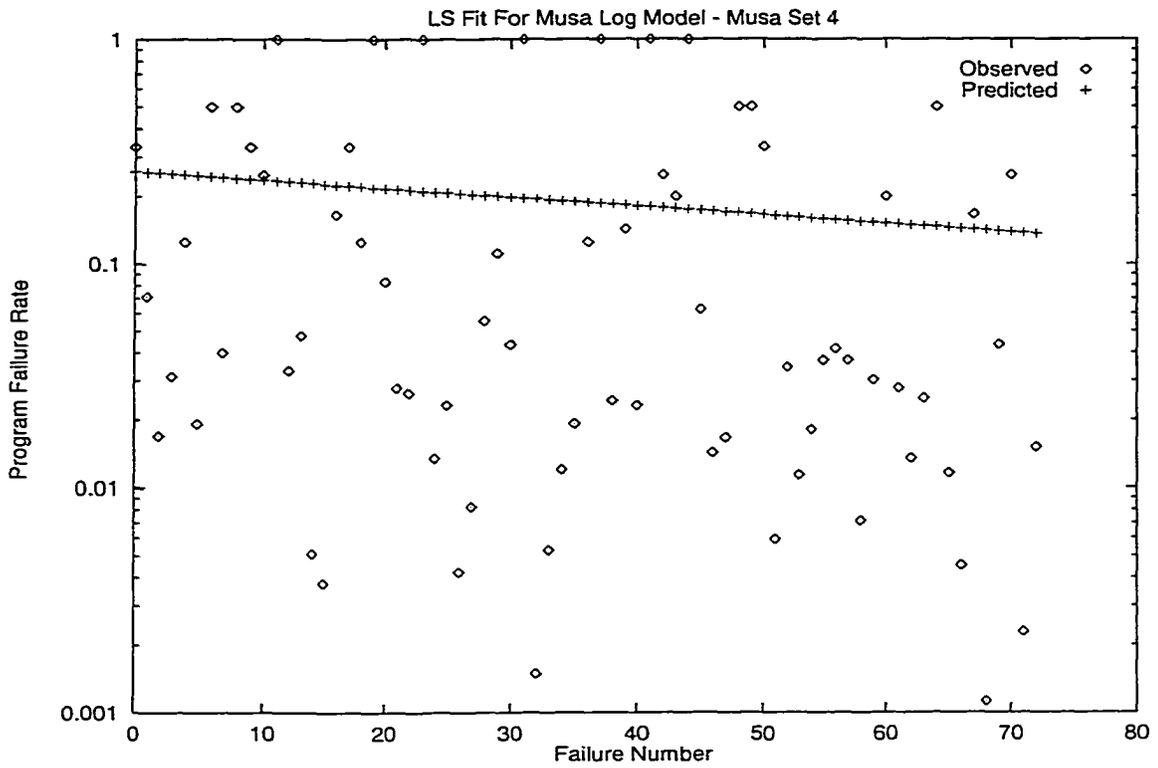

FIG. 37. Best Fits For ML Model Data Set 4

Model	JM	MB	ML	OS
Set 1	0.45232	0.445232	0.414309	0.465994
Set 2	0.000757267	0.000757267	0.000758417	0.000781957
Set 3	0.035536	0.035536	0.0356122	0.0354903
Set 4	6.29474	6.29474	6.31988	6.40239

FIG. 38. Error For Each Best Fit

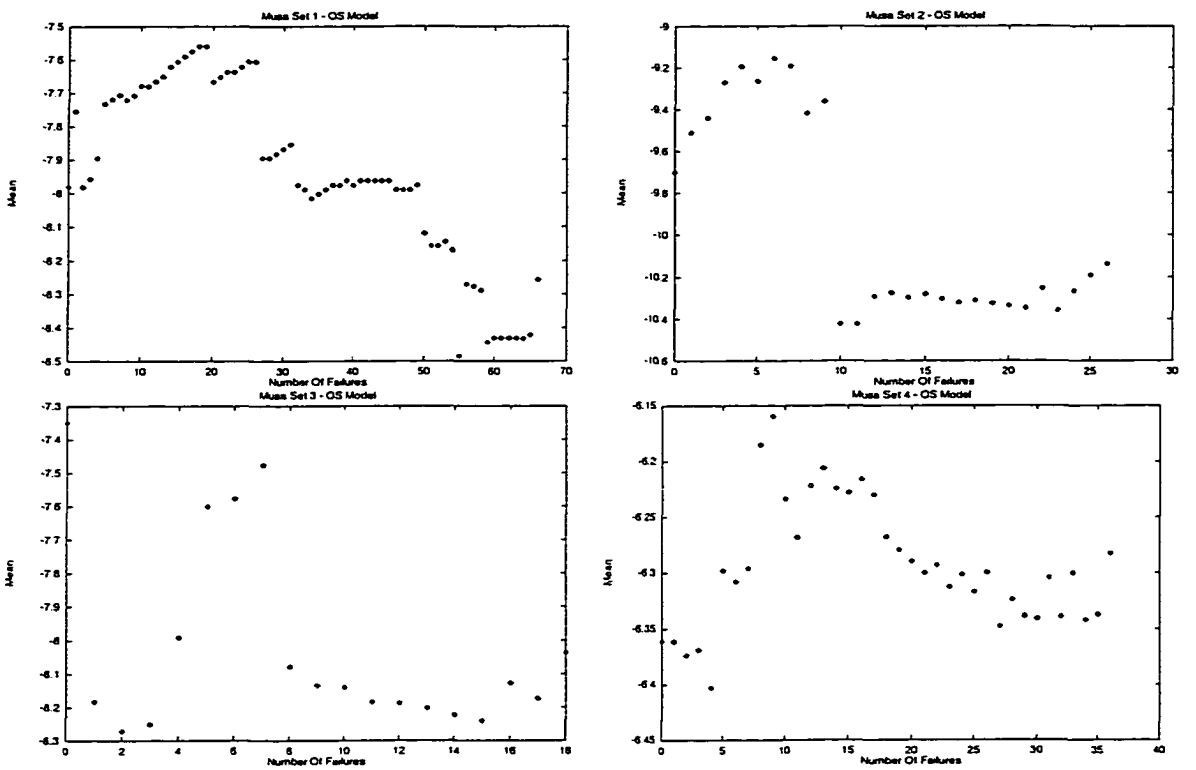

FIG. 39. OS Model Mean Progression For All Data Sets

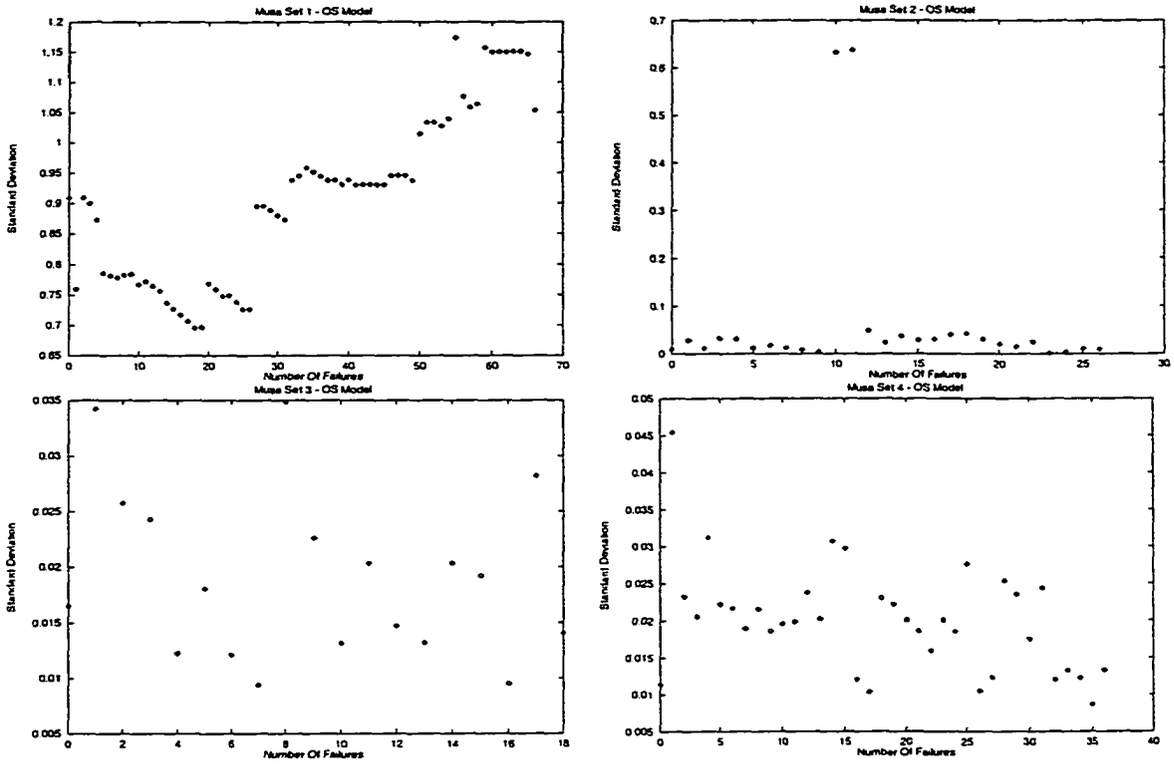

FIG. 40. OS Model Standard Deviation Progression For All Data Sets

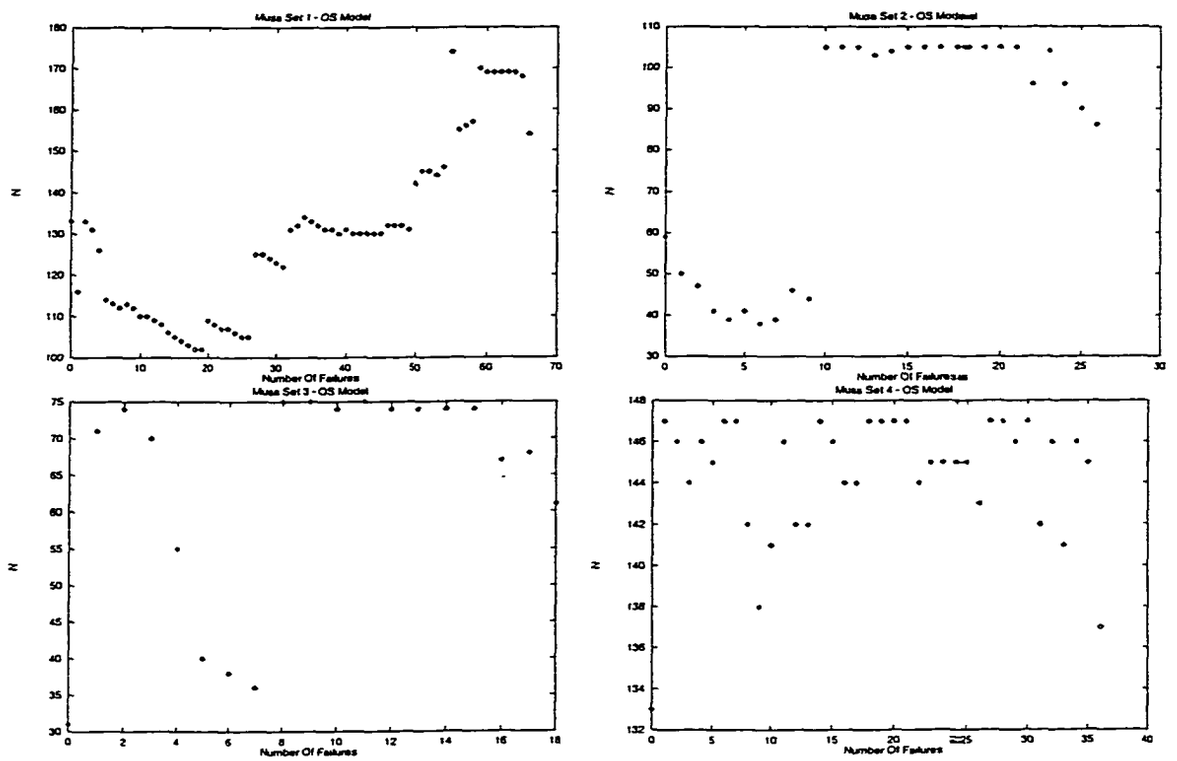

FIG. 41. OS Model N Progression For All Data Sets

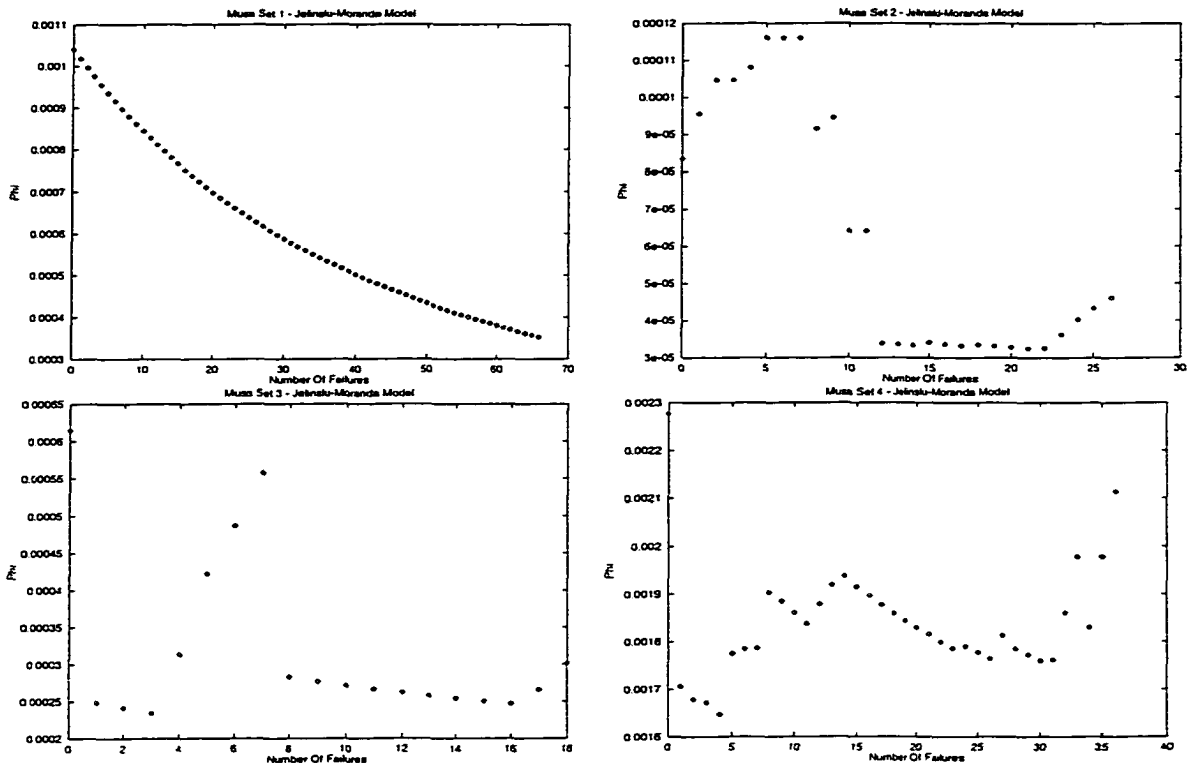

FIG. 42. Jelinski-Moranda Phi Progression For All Data Sets

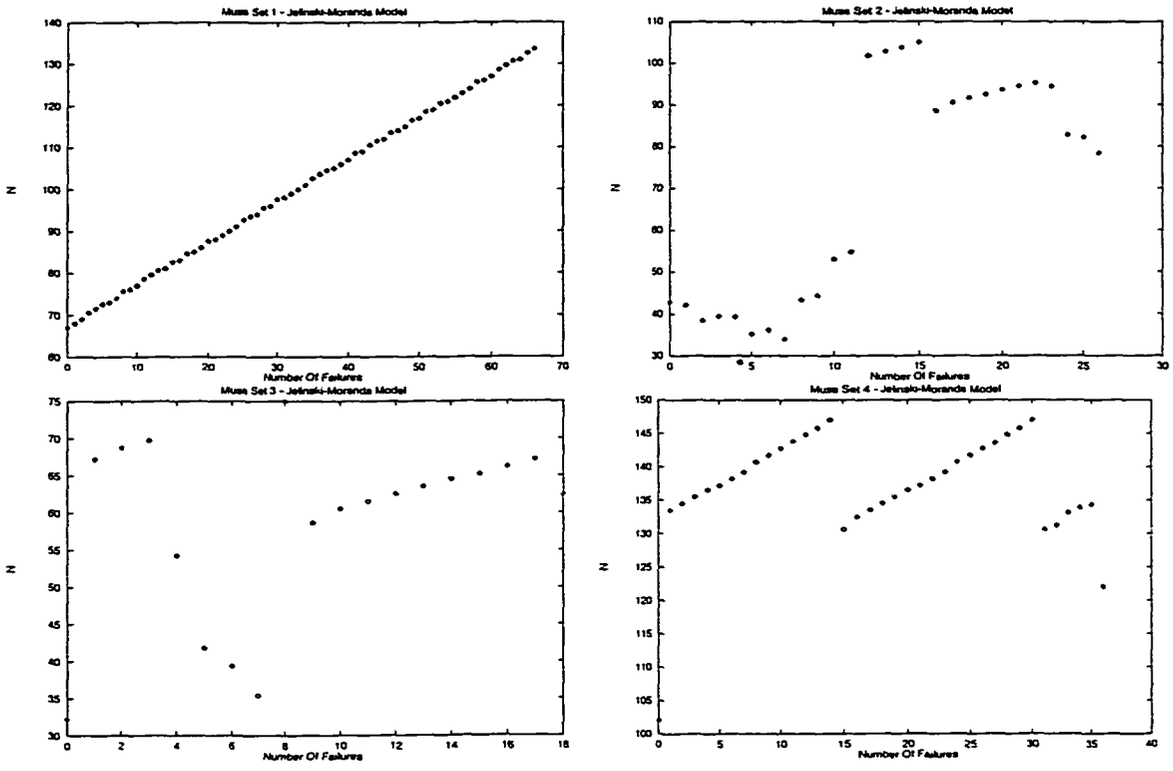

FIG. 43. Jelinski-Moranda N Progression For All Data Sets

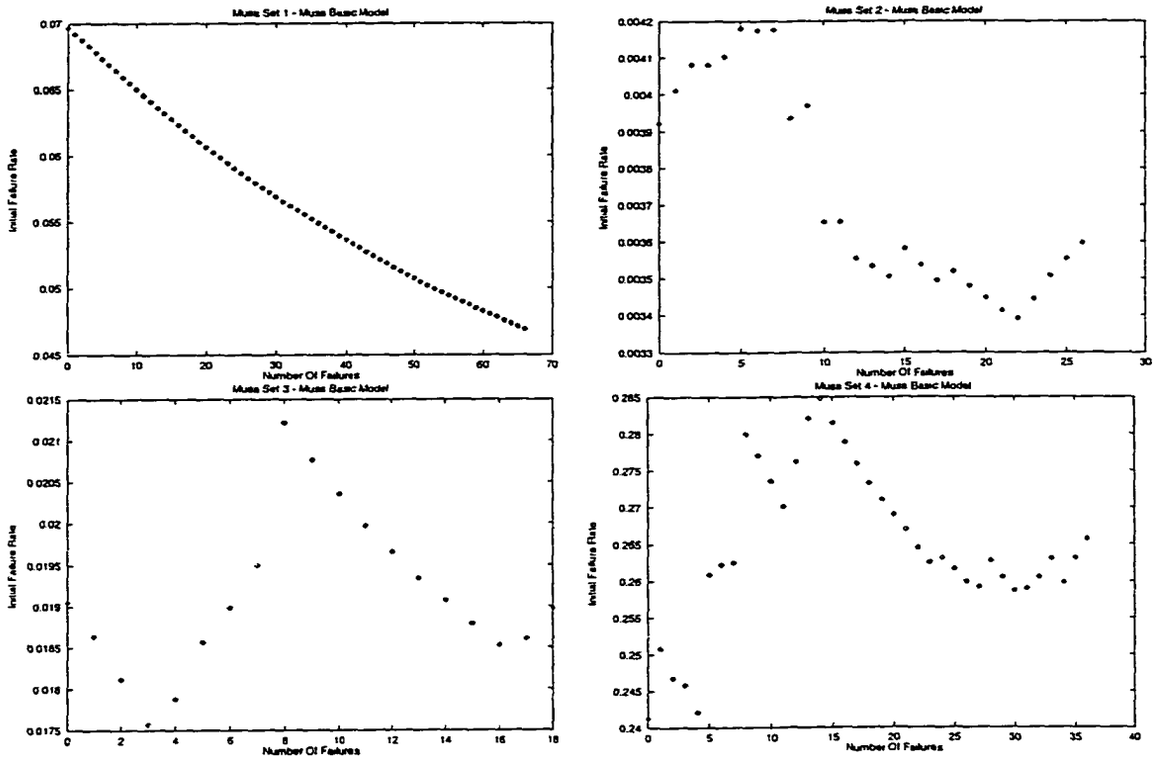

FIG. 44. Musa Basic Initial Program Failure Rate Progression For All Data Sets

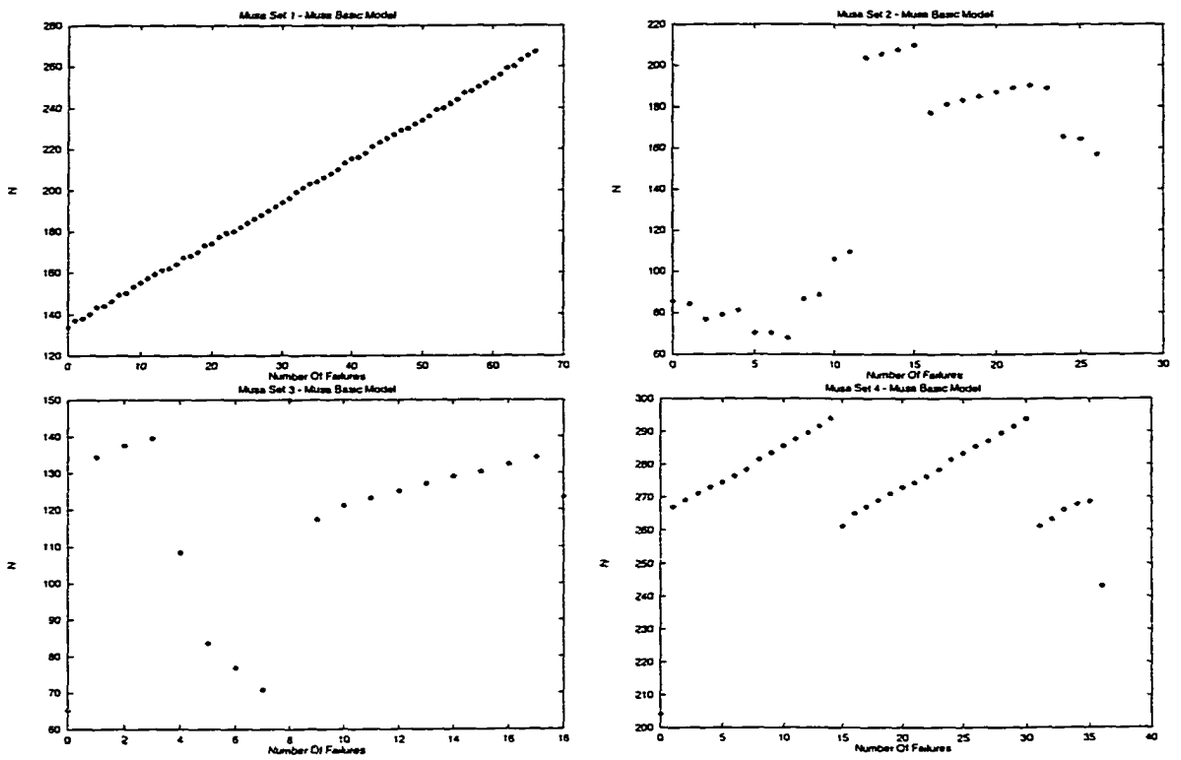

FIG. 45. Musa Basic N Progression For All Data Sets

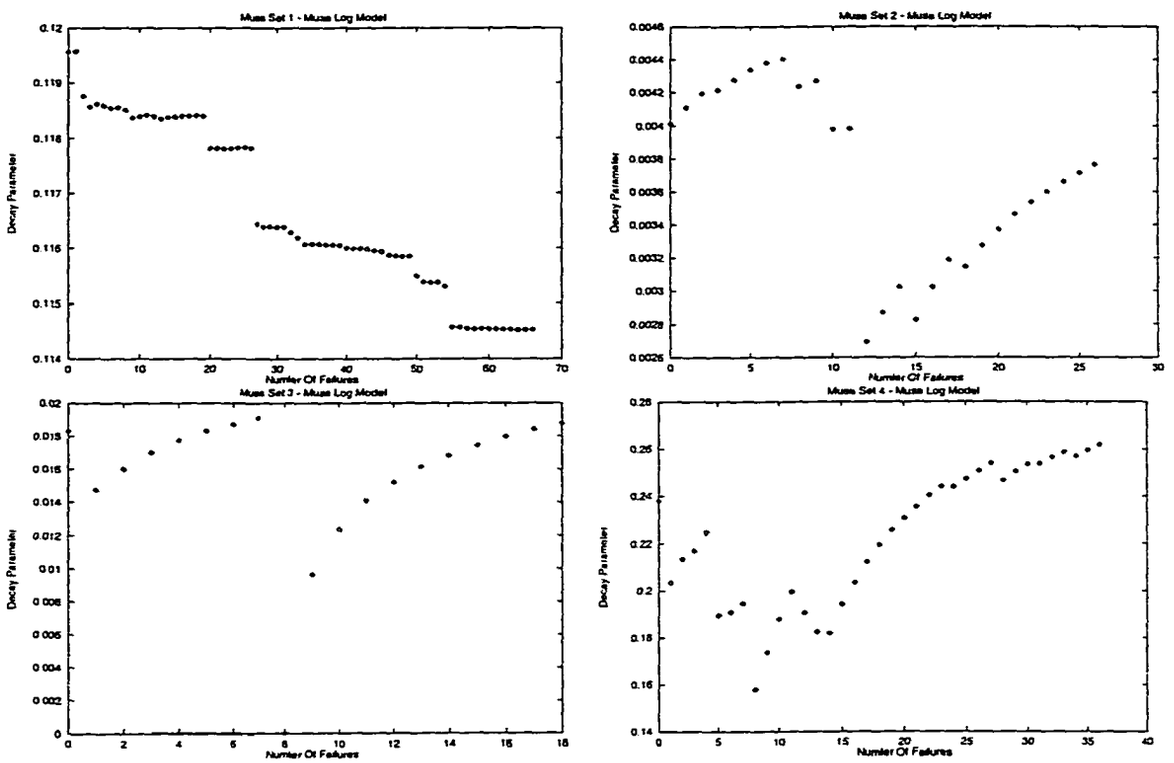

FIG. 46. Musa Log Decay Parameter Progression For All Data Sets

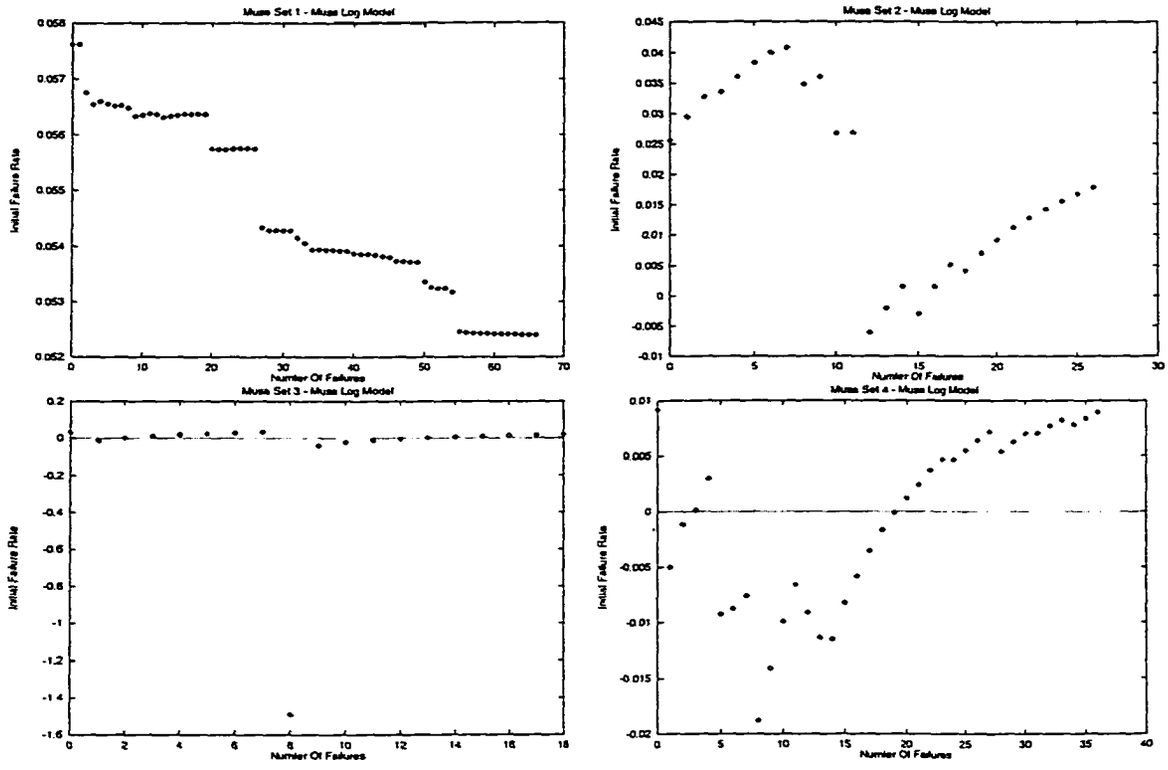

FIG. 47. Musa Log Initial Program Failure Rate Progression For All Data Sets

10.4 Conclusions

In this chapter, the initial verification of the model proposed in this dissertation was presented. The biggest problem that surfaced during this experiment afflicted all of the software reliablity models being used. It seems that data based on observations made during testing is simply too noisy to be of use for making accurate reliability estimates.

During this experiment, all of the models performed similarly, providing initial verification that the Order Statistics Model is at least as good as the existing models. In addition, this experiment provided further evidence that time based measures observed during testing (such as interfailure times) are simply too prone to statistical noise to provide a good basis for making accurate reliability estimates and predictions.

Chapter 11

Data From Generated Debugging Sequences

This chapter describes the second experiment that was conducted during the course of this research. The data for this experiment was generated by simulating debugging sequences based on existing failure rate data for an actual software system.

11.1 The Data Set

The basis for this study was a data set obtained in prior research conducted by Wild et al.[31], where the authors tested the Launch-Intercept-Control (LIC) software system using both representative methods and a directed testing method known as Knowledge-Driven Functional Testing. During Wild's experiment, the operational failure rate and directed failure rate for each discovered fault was recorded. This failure rate information is summarized in Figure 48.

Debugging sequences were generated from this data and the resulting failure rate data was used as input into selected reliability growth models. The process of generating the debugging sequences is described in the next section.

Fault #	# Failures / # Tests	
	Best Refined (100 tests)	Random (10^6 tests)
1.1	0	.000002
3.1	1.0	.000135
3.2	1.0	.000195
3.3	1.0	.000537
3.4	0	.000006
6.1	1.0	.000607
6.2	.89	.000511
6.3	1.0	.000032
7.1	0	.000071
8.1	1.0	.000225
8.2	1.0	.000098
9.1	.71	.000047
9.2	.15	.000006
11.1	.46	.000554
12.1	1.0	.000356
12.2	0	.000071
13.1	0	.000004
14.1	1.0	.001297
14.2	0	.000071
16.1	1.0	.000028
16.2	0	.000034
17.1	0	.000201
17.2	0	.000076

Fault #	# Failures / # Tests	
	Best Refined (100 tests)	Random (10^6 tests)
18.1	0	.000008
19.1	1.0	.000264
20.1	1.0	.000323
20.2	.46	.000697
21.1	1.0	.000085
21.2	0	.000007
22.1	1.0	.006551
22.2	1.0	.001735
22.3	1.0	.001735
23.1	1.0	.000072
23.2	0	.000008
24.1	1.0	.000260
25.1	1.0	.000014
25.2	1.0	.000080
25.3	.19	.000003
26.1	1.0	.000140
26.2	1.0	.000009
26.3	0	.000001
26.4	1.0	.000006
26.5	1.0	.000004
26.6	.15	.000368
26.7	.15	.000243

FIG. 48. Directed Fault Failure Rates (KDFT) versus Representative Fault Failure Rates [31]

11.2 Experiment Design

During this experiment, debugging sequences for both representative testing and the mixed method approach to testing were generated from the failure rate data provided by Wild et al [31]. The mixed method approach to testing was modeled as using representative methods at the start of testing and switching to directed methods later. The mixed method data was used as input into the Order Statistics model. The representative testing data was used as input into the Jelinski-Moranda model, the Musa Basic model, and the Musa Log model.

A total of four data sets of each type were generated for this experiment.

11.2.1 Generating Representative Testing Data

The generation of failure data from representative testing involved simulating and executing 100,000 representative test cases. This number of representative test cases is the same number of cases generated and executed in Wild's research. For each test case in this experiment, the probability of exposing a given fault was dictated by the operational failure rate of that fault in Wild's table.

For example, if given fault in the program had an operational failure rate of .000002, then on each iteration (1-100,000), a random number between 1 and 1,000,000 was generated. If the value of this random number was less than or equal to 2 ($1,000,000 * .000002 = 2$), then this fault was considered to be found by this simulated test case, and the failure time (iteration number) was recorded.

Simulating 100,000 test case iterations resulted in a set of system failure times that were converted to interfailure times, and then to program failure rates.

11.2.2 Generating A Mixture Of Representative and Directed Testing Data

The process of generating data that could be obtained from a mixed approach to testing was slightly different than generating representative testing data. This section describes these differences.

Fault	Failure Rate	Fault	Failure Rate
12.2	0.3333333	17.1	0.0015432
24.1	0.0344828	7.1	0.0025974
22.2	0.0322581	17.2	0.0004836
22.1	0.0526316	26.7	0.0015152
22.3	0.0128205	23.1	0.0001931
14.2	0.1250000	13.1	0.0000548
20.2	0.0909091	25.1	0.0003589
14.1	0.0024390	9.1	0.0003834
6.2	0.0476190	26.2	0.0006345
8.2	0.0036232	3.4	0.0001058
26.6	0.0135135	16.1	0.0003194
20.1	0.0136986	25.3	0.0007133
3.2	0.0012953	23.2	0.0000544
12.1	0.0023041	18.1	0.0045872
6.1	0.0344828	16.2	0.0000851
3.3	0.0062893	9.2	0.0000910
6.3	0.1428571		
11.1	0.0039370		
3.1	0.0053763		
19.1	0.0285714		
21.1	0.0046296		
8.1	0.3333333		
26.1	0.0013245		

FIG. 49. Generated Representative Set One

Fault	Failure Rate	Fault	Failure Rate
20.2	0.0232558	26.1	0.0003253
6.1	0.0312500	8.2	0.0006262
14.1	0.0079365	16.1	0.0009606
22.2	0.0108696	7.1	0.0013587
22.1	0.0208333	3.1	0.0004953
26.6	0.0109890	26.5	0.0000910
14.2	0.0066225	12.2	0.0007692
17.1	0.0086207	6.3	0.0001166
24.1	0.0119048	3.4	0.0001420
9.1	0.0030581	17.2	0.0000757
12.1	0.0833333	16.2	0.0001149
21.1	0.0400000	25.1	0.0000329
22.3	0.0043668		
3.3	0.0038168		
25.2	0.0049261		
11.1	0.0036232		
6.2	0.0029499		
26.7	0.0027322		
20.1	0.0007849		
3.2	0.0003922		
19.1	0.0625000		
8.1	0.0004666		
23.1	0.0081967		

FIG. 50. Generated Representative Set Two

Fault	Failure Rate	Fault	Failure Rate
24.1	0.0500000	14.2	0.0112360
22.1	0.0172414	21.1	0.0005845
22.2	0.0027933	19.1	0.0019685
14.1	0.0588235	17.2	0.0009615
3.2	0.0434783	16.1	0.0019380
22.3	0.0714286	26.1	0.0005599
6.1	0.0035211	12.2	0.0002147
12.1	0.2500000	25.2	0.0003804
20.2	0.0082645	9.2	0.0002263
23.1	0.0031250	9.1	0.0001856
17.1	0.0034364	21.2	0.0001381
11.1	0.1250000	25.3	0.0001907
6.2	0.0022371	16.2	0.0004218
3.3	0.0022422	13.1	0.0001182
3.1	0.3333333	23.2	0.0000231
8.1	0.0106383		
26.7	0.0016103		
20.1	0.0025510		
26.6	0.0021277		
26.4	0.0004348		
8.2	0.0035587		
6.3	0.0013477		
7.1	0.0019231		

FIG. 51. Generated Representative Set Three

Fault	Failure Rate	Fault	Failure Rate
22.1	0.0476190	7.1	0.0009681
24.1	0.0555556	25.2	0.0027701
14.1	0.0121951	26.1	0.0007862
22.2	0.0103093	17.2	0.0016129
26.6	0.0072464	3.2	0.0001279
3.3	0.0555556	14.2	0.0005299
6.1	0.0054054	12.2	0.0058480
26.7	0.2000000	26.5	0.0004502
22.3	0.0147059	26.4	0.0000615
16.2	0.0227273	6.3	0.0000967
19.1	0.0588235	26.2	0.0000799
12.1	0.0011198	25.1	0.0000845
17.1	0.0034483	18.1	0.0002861
11.1	0.0013699		
8.1	0.0072464		
23.1	0.0033898		
6.2	0.0277778		
20.2	0.0078125		
8.2	0.0020080		
3.1	0.0029762		
21.1	0.0029940		
20.1	0.0026954		
9.1	0.0007225		

FIG. 52. Generated Representative Set Four

Determining The Testing Cross-Over Point

Before generating data for a mixed method testing process, the point at which simulated testing would switch from representative methods to directed methods had to be identified. The actual determination of this cross-over point will ultimately be made by testers when they feel that representative testing is starting to"take too long".

For this experiment, the cross-over from representative methods to directed methods was made when the first interfailure time exceeding 1000 was observed. Therefore, the crossover occured when 1000 representative test cases had been generated and executed without exposing a fault.

For each directed data set, the representative failure rate data for the set is taken from the corresponding generated representative data set. Thus, the data in each generated mixed testing set matches the data in the corresponding generated representative teting set up until the crossover point. For example, in the first pair of generated sets, the first 26 values of the representative data set are identical to the first 26 values of the mixed data set.

Determining The Number Of Directed Test Cases To Generate

After the cross-over point had been reached during the generation of a mixed data set, instead of simulating the failure behavior of the software over the remainder of the 100,000 test cases as was done when representative testing was simulated, the behavior of the software when executing five passes through the refined functional test cases defined by Wild [31] was simulated.

Five passes through the refined functional test set represents the execution of about 500 directed test cases. In practice, testers will probably only make one pass through their directed test sets. Multiple passes were made during this experiment in order to maximize the amount of data available for analysis. As it turned out, however, the first pass through the directed test suite usually ended up finding over ninety percent of the faults that were found during the five iterations. Figure 53 shows the number of faults found for each data set for each directed testing

Pass	Set 1	Set 2	Set 3	Set 4
1	9	14	12	9
2	2	1	1	0
3	0	0	0	1
4	0	0	0	0
5	0	0	0	1

FIG. 53. Number Of Faults Found For Each Directed Testing Pass For Each Data Set

pass.

Determining Fault Exposure Under Directed Testing

During generation of representative testing data, the probability of finding a given fault was dictated by the fault's operational failure rate from Wild's table. During simulation of directed testing, however, the probability of finding a given fault was dictated by fault's failure rate under directed testing. Additionally, instead of recording the program failure rate for each of the faults found during the directed testing phase of the data generation, the fault failure rate was recorded.

11.3 Results

After the data sets had been generated, they were used as input to several reliability models and the results were compared.

As in the experiment presented in the previous chapter, the analysis of the models consisted of generating OP-Plots for each model for each data set, comparing the best fits for each model for each complete data set, and comparing the stability of the models using parameter progression plots.

Fault	Failure Rate	Fault	Failure Rate
12.2	0.3333333	17.1	0.0015432
24.1	0.0344828	7.1	0.0025974
22.2	0.0322581	17.2	0.0004836
22.1	0.0526316	9.2	0.0000060
22.3	0.0128205	16.1	0.0000280
14.2	0.1250000	23.1	0.0000720
20.2	0.0909091	25.1	0.0000140
14.1	0.0024390	25.2	0.0000800
6.2	0.0476190	25.3	0.0000030
8.2	0.0036232	26.2	0.0000090
26.6	0.0135135	26.4	0.0000060
20.1	0.0136986	26.5	0.0000040
3.2	0.0012953	9.1	0.0000470
12.1	0.0023041	26.7	0.0002430
6.1	0.0344828		
3.3	0.0062893		
6.3	0.1428571		
11.1	0.0039370		
3.1	0.0053763		
19.1	0.0285714		
21.1	0.0046296		
8.1	0.3333333		
26.1	0.0013245		

FIG. 54. Generated Mixed Testing Set One

Fault	Failure Rate	Fault	Failure Rate
20.2	0.0232558	8.2	0.0000980
6.1	0.0312500	16.1	0.0000280
14.1	0.0079365	19.1	0.0002640
22.2	0.0108696	23.1	0.0000720
22.1	0.0208333	25.1	0.0000140
26.6	0.0109890	25.3	0.0000030
14.2	0.0066225	26.1	0.0001400
17.1	0.0086207	26.2	0.0000090
24.1	0.0119048	26.4	0.0000060
9.1	0.0030581	26.5	0.0000040
12.1	0.0833333	9.2	0.0000060
21.1	0.0400000		
22.3	0.0043668		
3.3	0.0038168		
25.2	0.0049261		
11.1	0.0036232		
6.2	0.0029499		
26.7	0.0027322		
20.1	0.0007849		
3.1	0.0001350		
3.2	0.0001950		
6.3	0.0000320		
8.1	0.0002250		

FIG. 55. Generated Mixed Set Two

Fault	Failure Rate	Fault	Failure Rate
24.1	0.0500000	16.1	0.0000280
22.1	0.0172414	19.1	0.0002640
22.2	0.0027933	21.1	0.0000850
14.1	0.0588235	25.1	0.0000140
3.2	0.0434783	25.2	0.0000800
22.3	0.0714286	25.3	0.0000030
6.1	0.0035211	26.1	0.0001400
12.1	0.2500000	26.2	0.0000090
20.2	0.0082645	26.5	0.0000040
23.1	0.0031250	9.1	0.0000470
17.1	0.0034364		
11.1	0.1250000		
6.2	0.0022371		
3.3	0.0022422		
3.1	0.3333333		
8.1	0.0106383		
26.7	0.0016103		
20.1	0.0025510		
26.6	0.0021277		
26.4	0.0004348		
6.3	0.0000320		
8.2	0.0000980		
9.2	0.0000060		

FIG. 56. Generated Mixed Set Three

Fault	Failure Rate	Fault	Failure Rate
22.1	0.0476190	3.2	0.0001950
24.1	0.0555556	6.3	0.0000320
14.1	0.0121951	16.1	0.0000280
22.2	0.0103093	25.1	0.0000140
26.6	0.0072464	25.2	0.0000800
3.3	0.0555556	26.1	0.0001400
6.1	0.0054054	26.2	0.0000090
26.7	0.2000000	26.4	0.0000060
22.3	0.0147059	26.5	0.0000040
16.2	0.0227273	25.3	0.0000030
19.1	0.0588235	9.2	0.0000060
12.1	0.0011198		
17.1	0.0034483		
11.1	0.0013699		
8.1	0.0072464		
23.1	0.0033898		
6.2	0.0277778		
20.2	0.0078125		
8.2	0.0020080		
3.1	0.0029762		
21.1	0.0029940		
20.1	0.0026954		
9.1	0.0007225		

FIG. 57. Mixed Set Four

11.3.1 The Results Of Analyzing The Predictive Accuracy Of The Models

Figures 58 to 73 show the OP Plots that were generated for each model for each data set. The sum of the squared error for each iteration of each OP Plot was averaged and is shown in Figure 74. It is difficult to tell much difference between the models simply by looking at the plots, but the data in Figure 74 indicates that the Order Statistics model and the Musa Log model perform similarly. Both of these models performed better than the Jelinski-Moranda model and the Musa Basic model.

When looking at the OP Plots, it is evident that the OP Plots for the Order Statistics model have fewer points plotted than do the other models. The cause for the difference in the number of plotted points is the Order Statistics model's use of data from a combination of representative and directed testing.

Under purely representative testing, every fault that is found is represented on the OP Plot by its program failure rate. The Order Directed Testing Property, however, dictates that predicted program failure rates will only have meaning under directed testing as testing nears coverage.

For this reason, instead of plotting 10-15 points corresponding to the faults found under directed testing, only 5 points are plotted on the OP Plot - one for each directed testing coverage pass simulated in this experiment.

11.3.2 Comparing The Best Fits For Each Model

Figures 75 to 90 show the best fits for each model for each of the data sets. One of the most noticeable differences between these plots and the best fit plots in the last chapter is the discontinuity present in the plots for the Order Statistics model. This discontinuity is caused by the switch from representative testing to directed testing in the mixed method approach to testing. When representative testing is used, the program failure rates are plotted. Once testing switches to directed methods, fault failure rates are plotted. Therefore, the discontinuity in the best fit plot reflects the fact that program failure rates values are generally an

order of magnitude (or more) larger than fault failure rate values.

Combination of these two different quantities into a single fitting process without biasing the results was accomplished by normalizing the error measures during fitting. Instead of calculating the fitting error as:

$$e = \sum_{i=1}^{n} (predicted_i - observed_i)^2, \tag{21}$$

the fitting error was calculated as:

$$e = \sum_{i=1}^{n} \left(\frac{predicted_i - observed_i}{observed_i} \right)^2. \tag{22}$$

Upon examination of the best fit plots and the data in Figure 91, it is apparent that performance of the Order Statistics model when fitting to the entire data set compares favorably to the other models. For all of the test sets, the error measure for the Order Statistics model is similar to or better than the error measure for the Musa Log model. The error measures for the Order Statistics model and the Musa Log model are much better than the error measures for the Jelinski-Moranda model and the Musa Basic Model.

One of the most important observations to be made about the best fit plots is that the Order Statistics model does a very good job of fitting to the fault failure rates present in the mixed method data sets. The fact that the fault failure rates are sorted into descending order by the Order Statistics model, along with the fact that the fault failure rates seem more stable than the program failure rates seems to greatly improve the goodness of fit. These observations lend yet more support to the argument that software reliability practicioners should consider switching their emphasis from quantities observed during testing to quantities estimated during debugging in order to improve model performance.

11.3.3 Comparing The Parameter Progressions For Each Model

The parameter progressions for each parameter for each model during the generation of the OP Plots is shown in Figures 92 to 100.

As in the previous chapter, after examining the parameter progression plots, it appears that in most cases the Order Statistics model's estimates for N as faults are discovered appear to be more stable than the estimates provided by the Jelinski-Moranda Model and the Musa Basic model. The plots generated from the parameter estimate sequences of the latter models show that the estimate for N almost always increases as new data is added. The plots generated from the parameter estimate sequences of the Order Statistics model (with the exception of Set 3) consist of several subsequences within which N is fairly constant. From these plots, it appears that the Order Statistics model provided more stable predictions for N as faults were discovered.

Examination of the parameter progression plots for the other parameters for all of the models seems to lead to the same conclusion. The subsequences of parameter progressions in the plots tend to be more stable for the Order Statistics model than for the Jelinski-Moranda model and the Musa Basic model. The stability of the Musa Log model seems to similar to the stability of the Order Statistics model.

11.4 Verification Of The Ordered Directed Testing Property

In previous work, Mitchell and Zeil [17] conducted a statistical test on the failure rate data used in this chapter to determine the relationship between the directed testing failure rates and the operational failure rates for each fault in the data set. Mitchell and Zeil calculated the Spearman rank correlation coefficient for the two failure rate sets to be 0.42. A coefficient of 0.42 or higher has only a 0.0027 probability of arising in independent variables. These results lend preliminary

support to the validity of the Ordered Directed Testing Property.

11.5 Conclusions

It appears that the results from the Order Statistics model under mixed testing and the Musa Log model under representative testing are very similar. Both of these models seem to provide better fits and predictive performance than the Jelinski-Moranda model and the Musa Basic model.

An important difference between the Order Statistics model and the Musa Log model, however, is that the the Order Statistics Model required far fewer test cases to generate its failure set, because it used a mixed method approach to testing that utilized directed testing. Of course, since it is more expensive to generate a directed test case than a representative test case, this advantage may be somewhat offset or negated. Nonetheless, the use of multiple testing methods for reliability assessment should yield a more robust and comprehensive testing process than would be obtained if a single method were used.

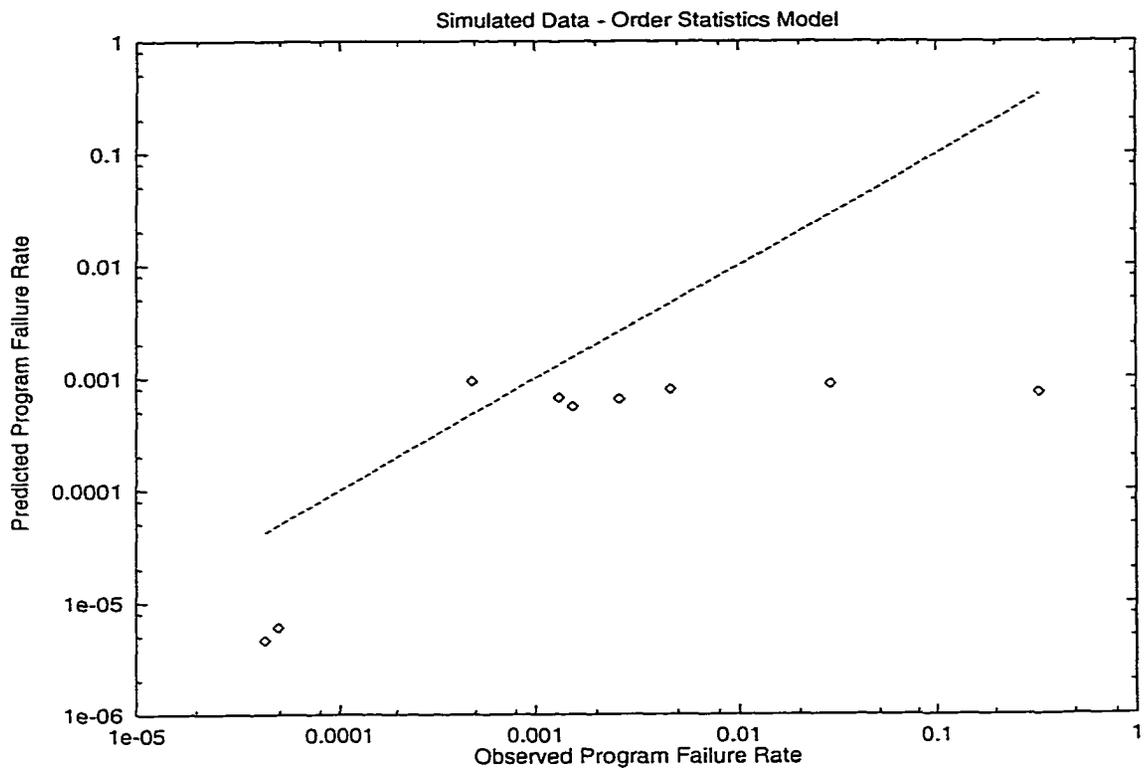

FIG. 58. OS Model OP Plot For Generated Data Set One

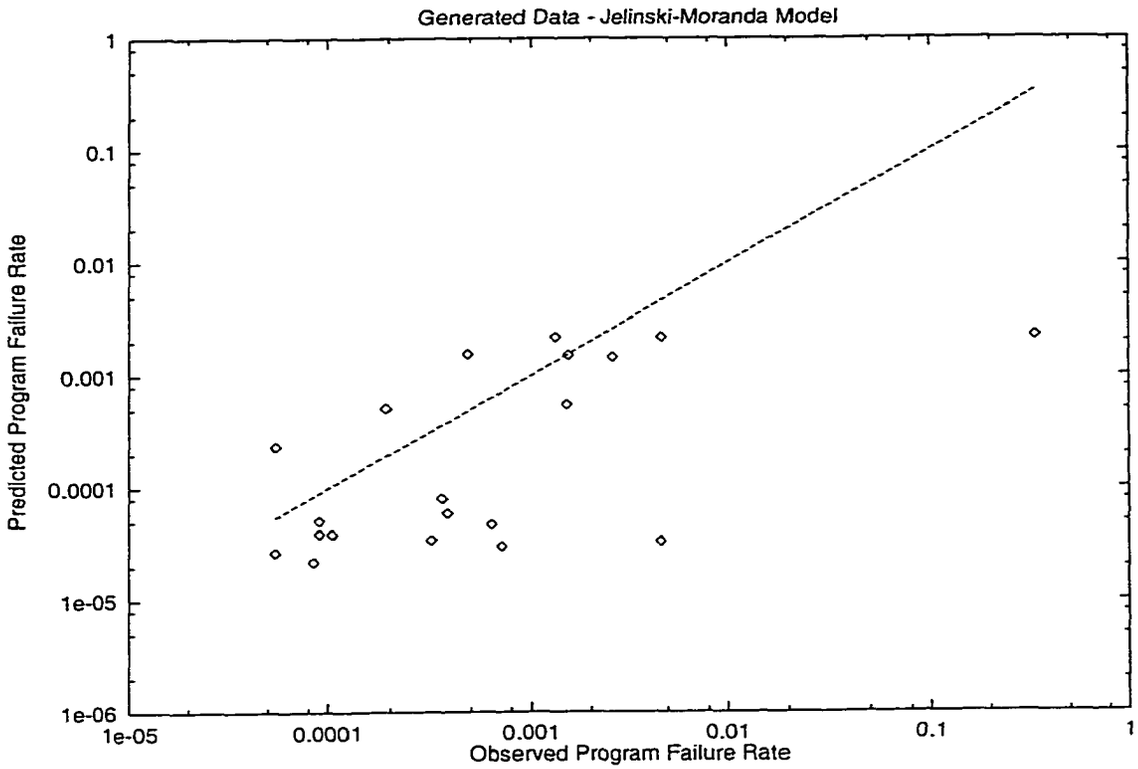

FIG. 59. JM Model OP Plot For Generated Data Set One

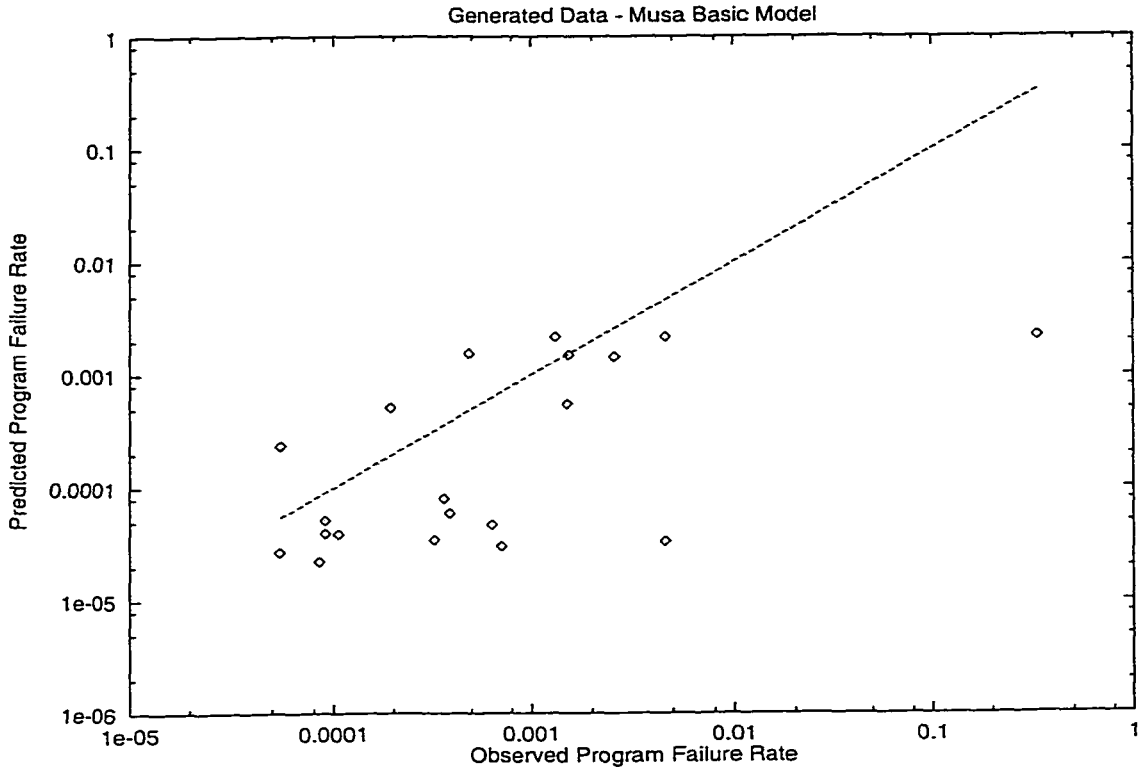

FIG. 60. MB Model OP Plot For Generated Data Set One

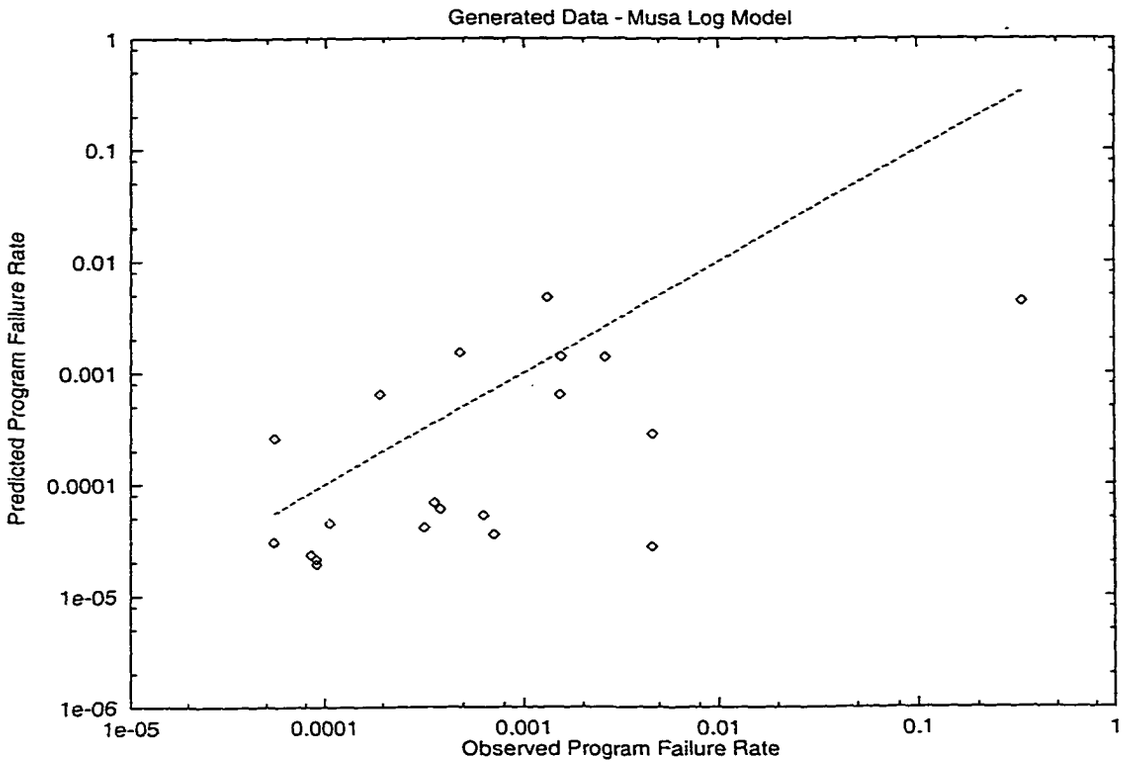

FIG. 61. ML Model OP Plot For Generated Data Set One

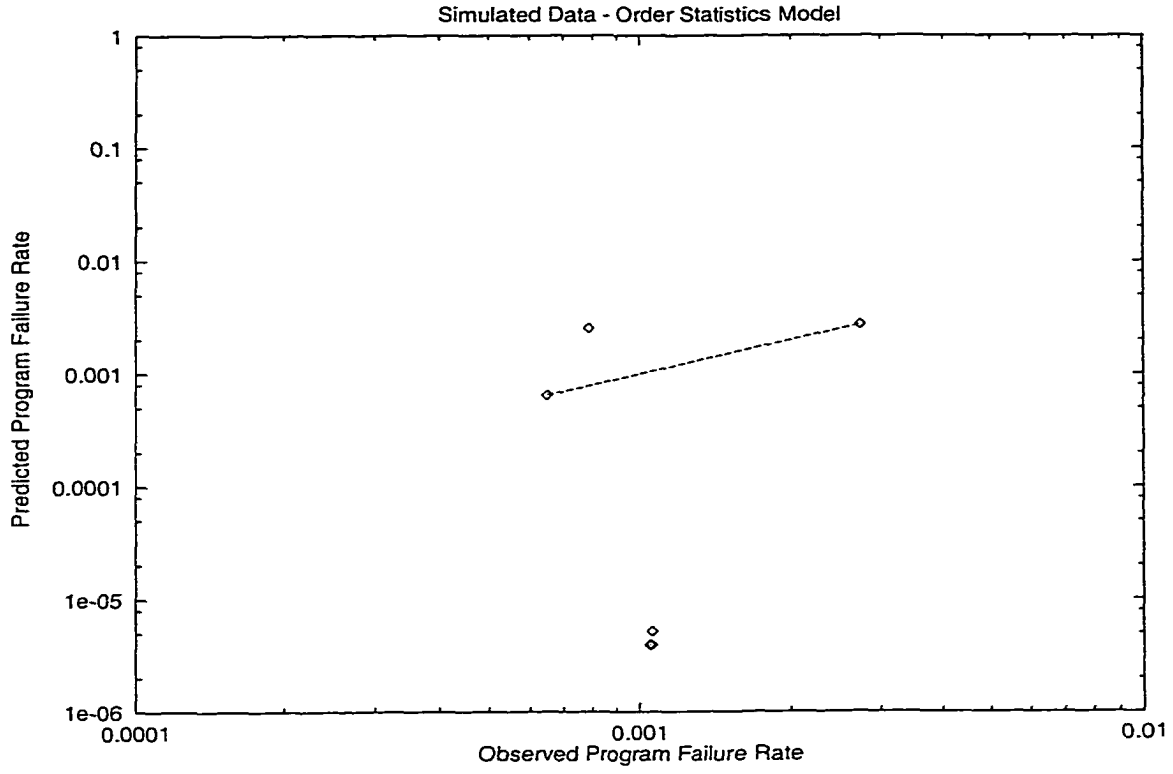

FIG. 62. OS Model OP Plot For Generated Data Set Two

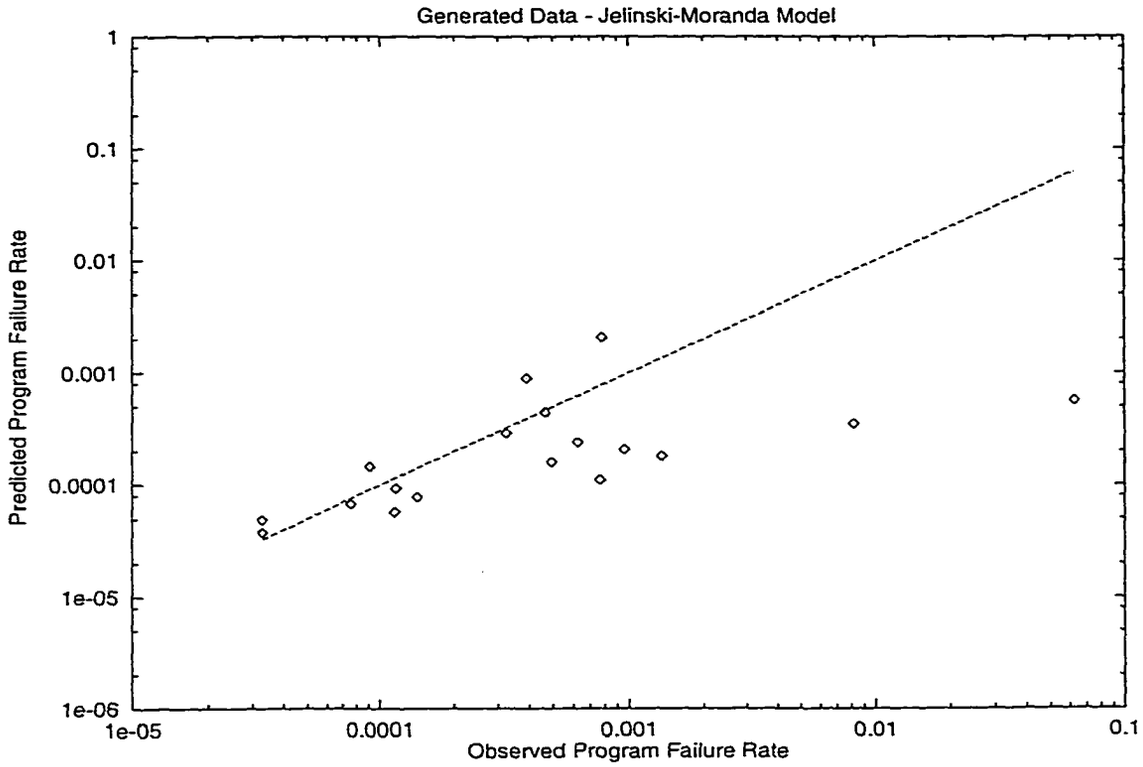

FIG. 63. JM Model OP Plot For Generated Data Set Two

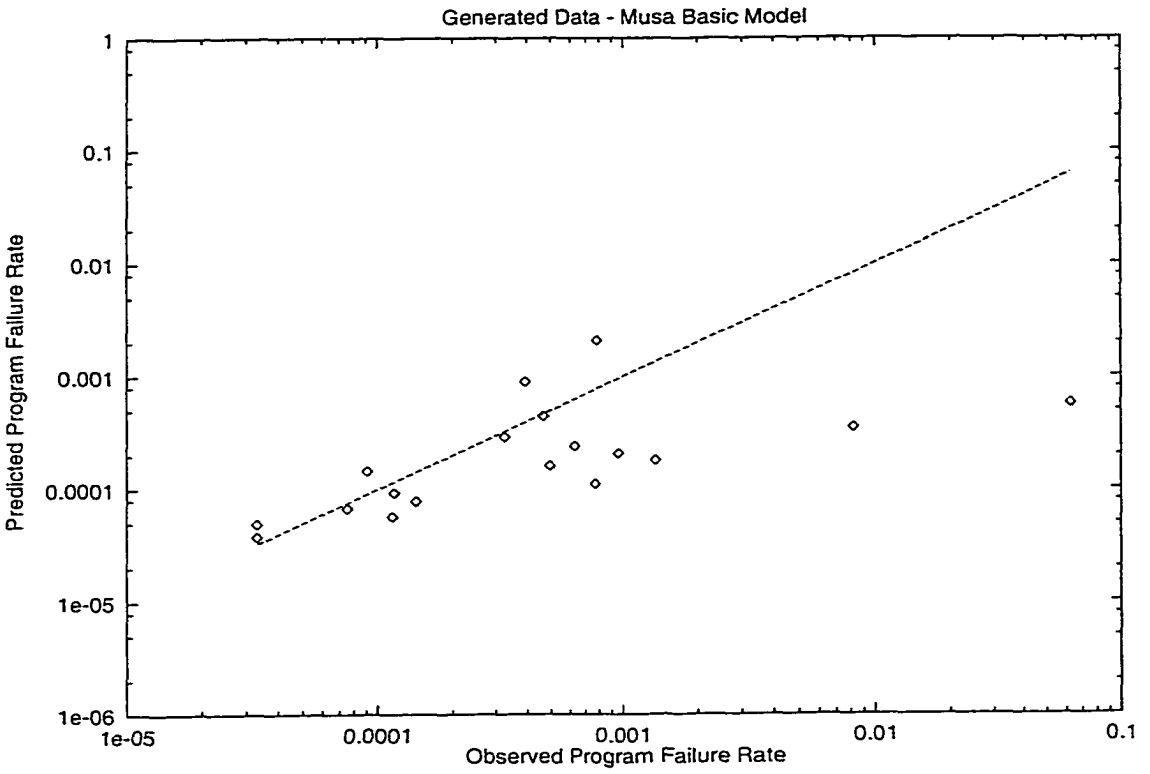

FIG. 64. MB Model OP Plot For Generated Data Set Two

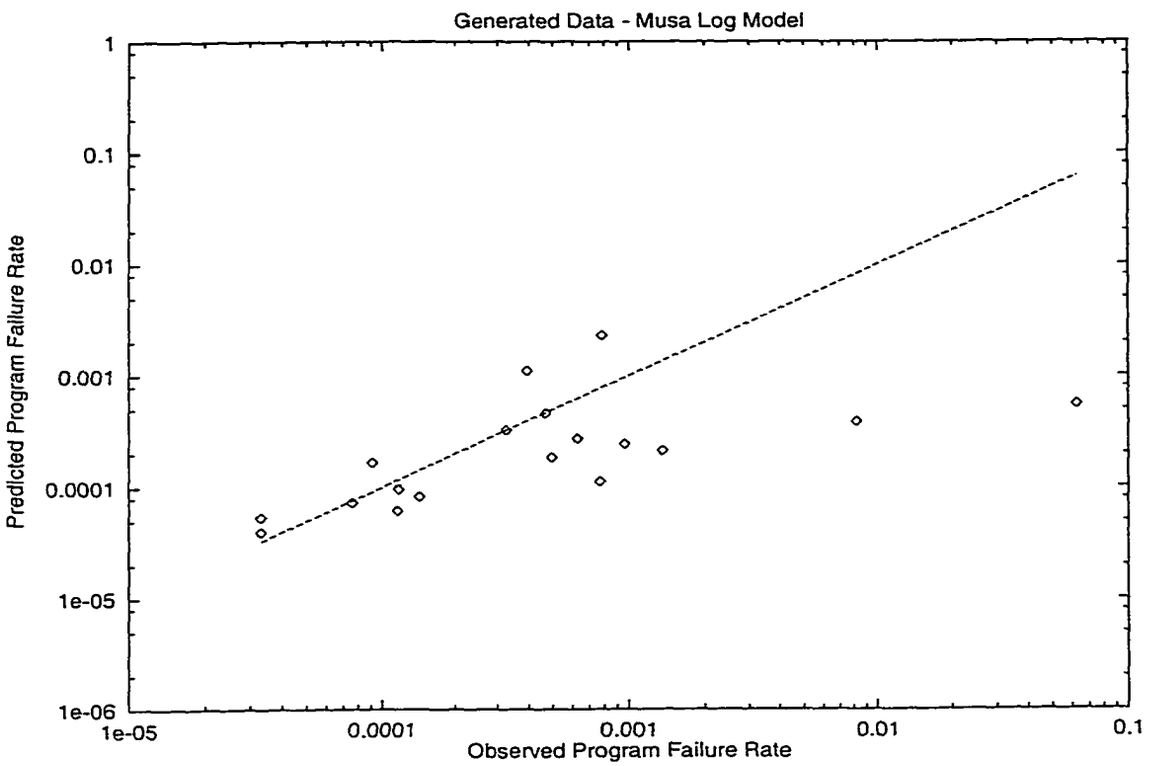

FIG. 65. ML Model OP Plot For Generated Data Set Two

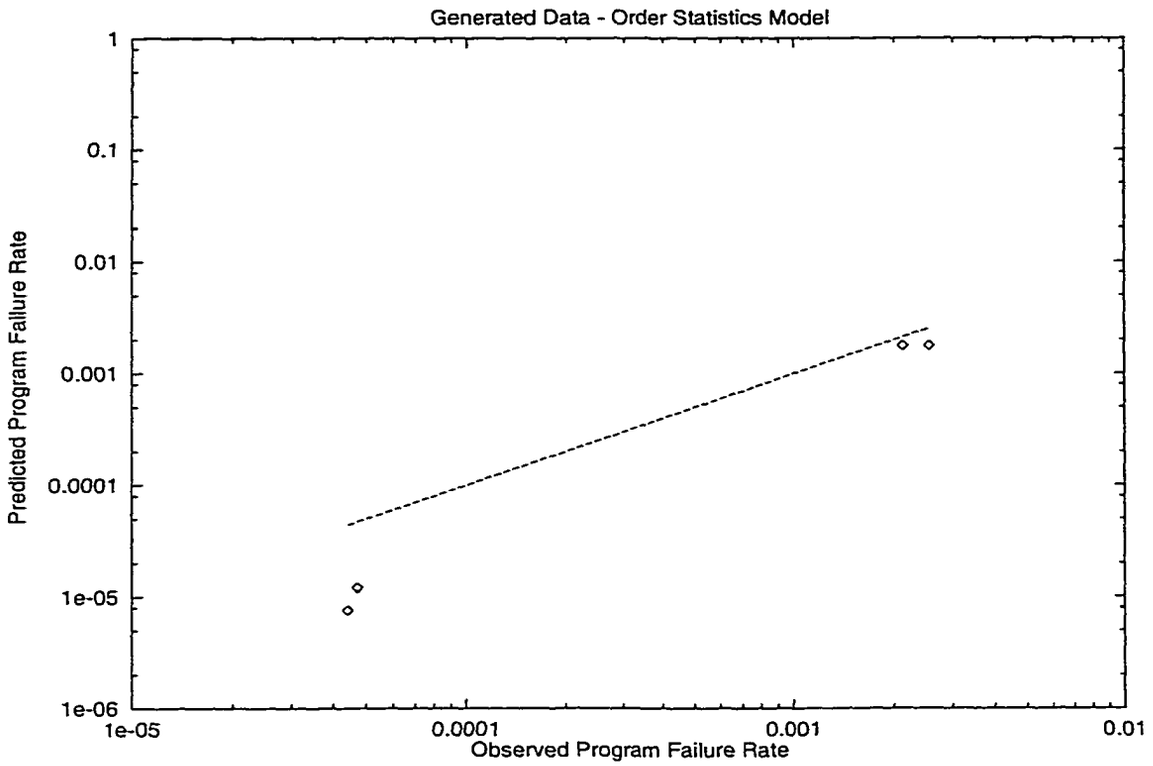

FIG. 66. OS Model OP Plot For Generated Data Set Three

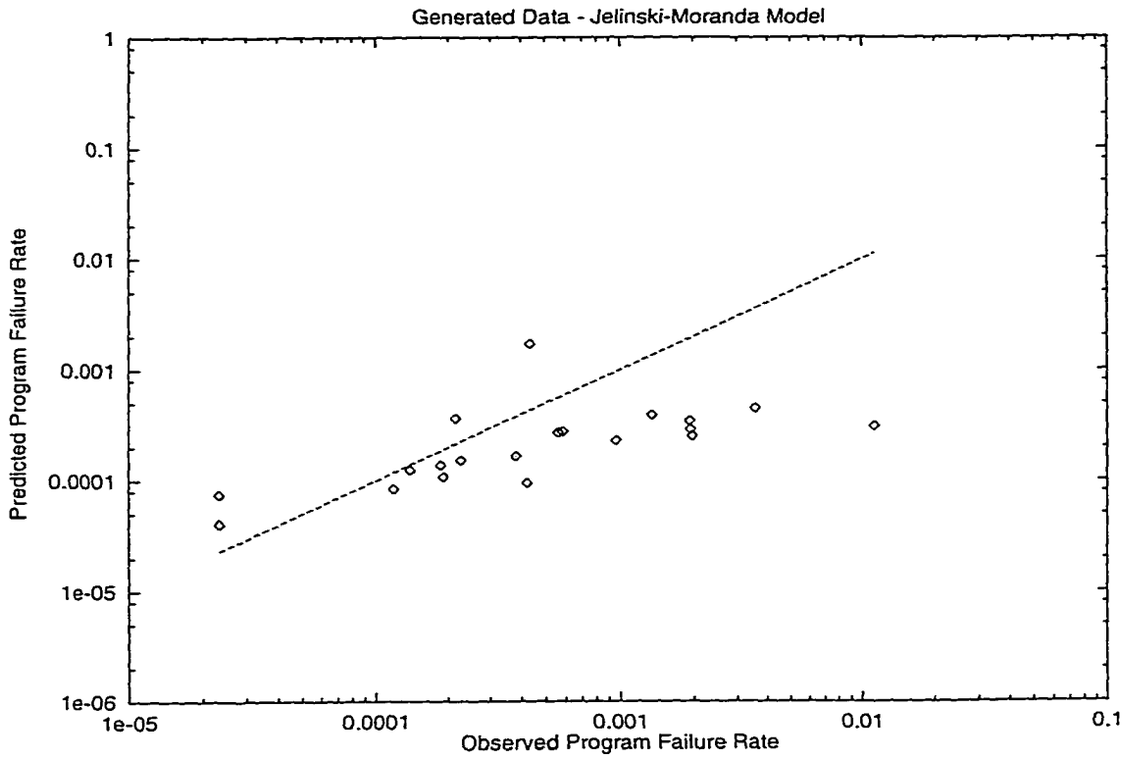

FIG. 67. JM Model OP Plot For Generated Data Set Three

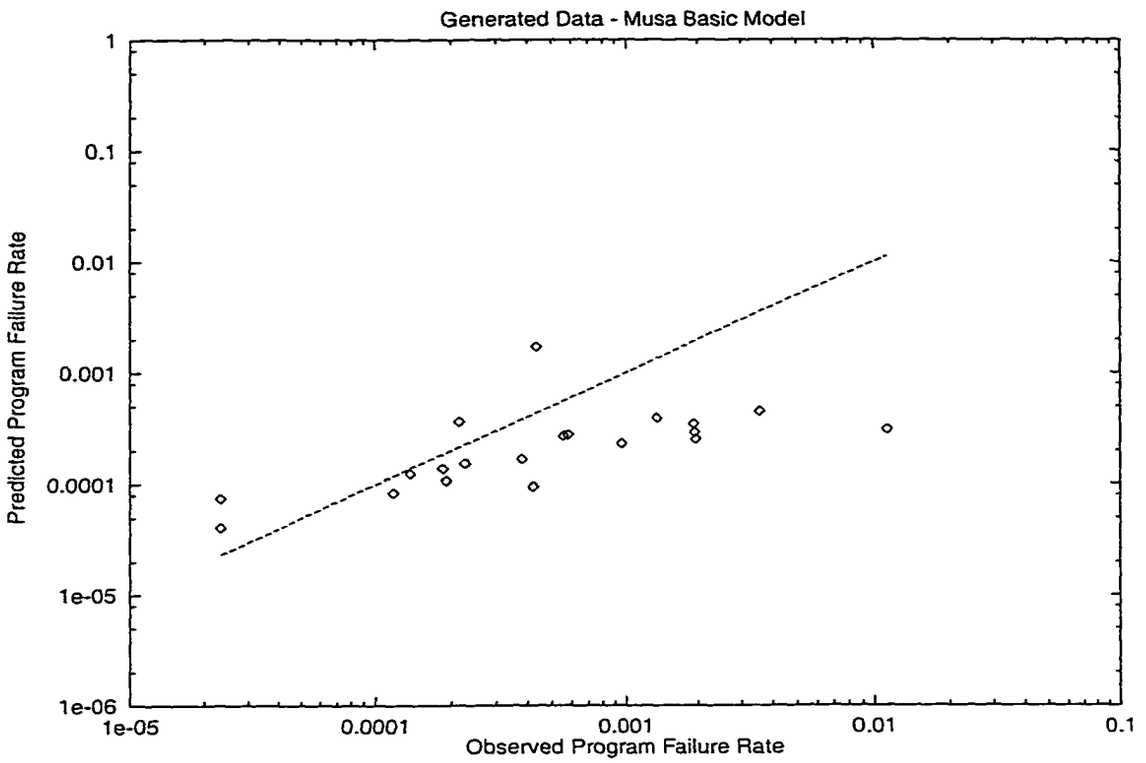

FIG. 68. MB Model OP Plot For Generated Data Set Three

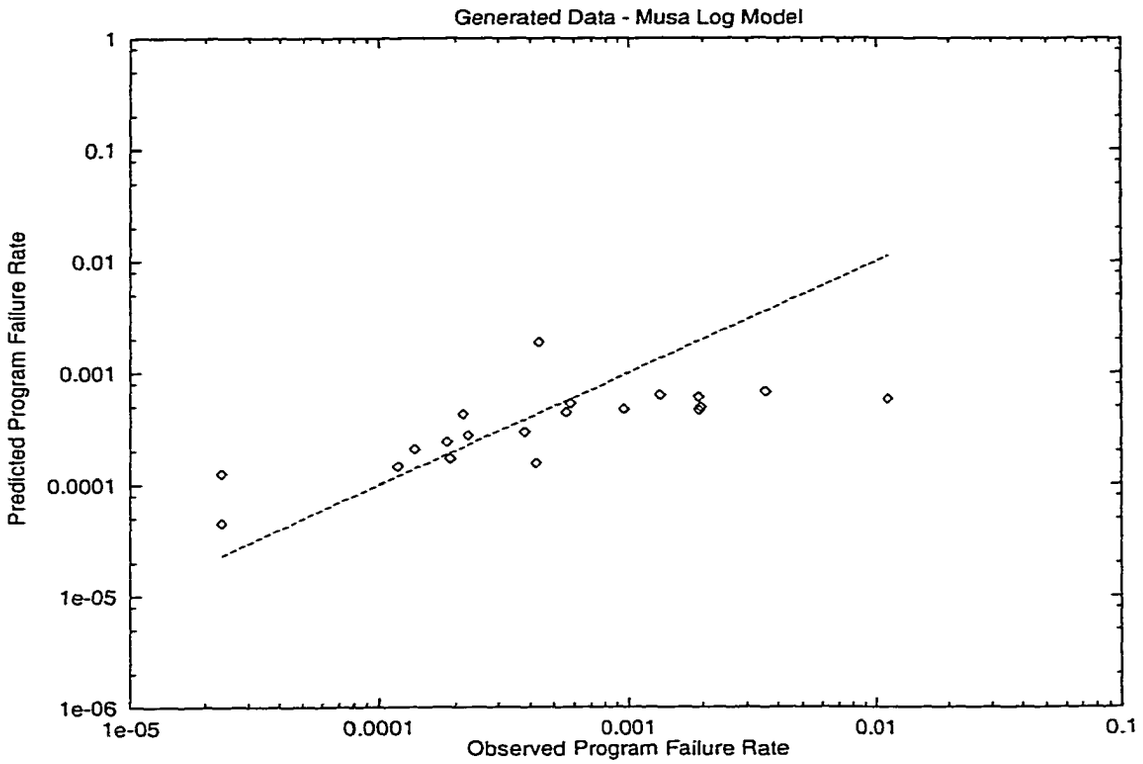

FIG. 69. ML Model OP Plot For Generated Data Set Three

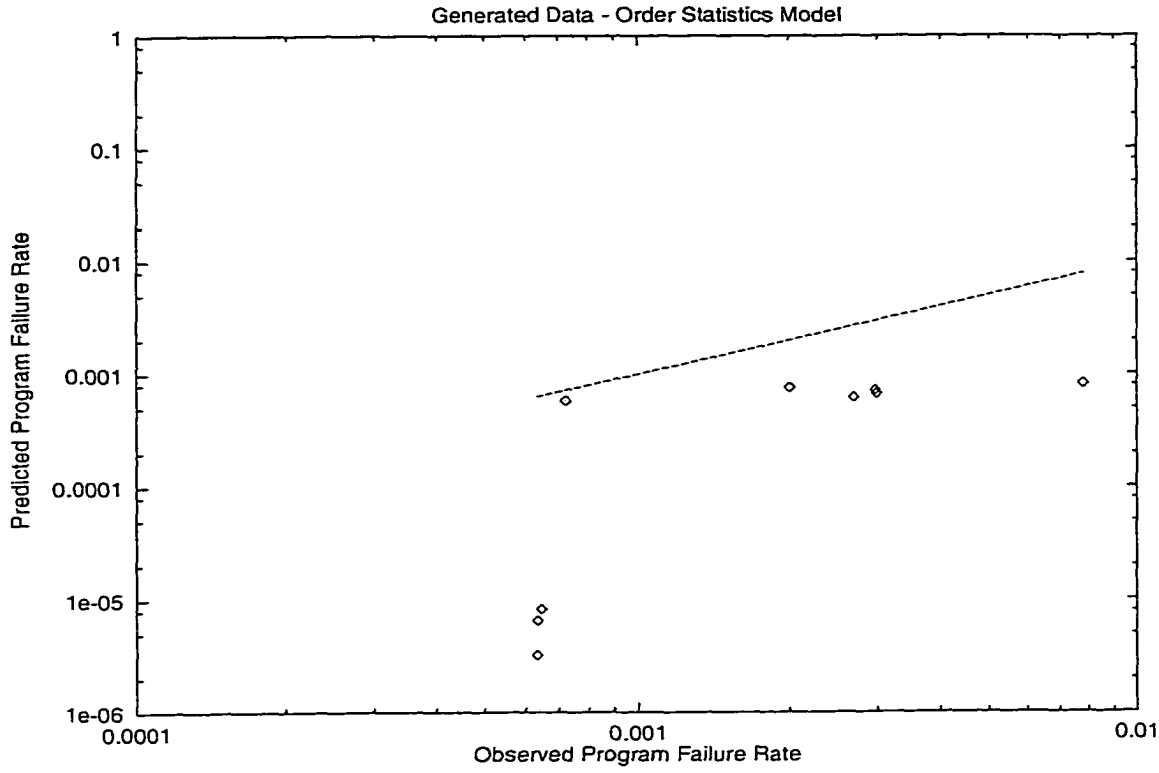

FIG. 70. OS Model OP Plot For Generated Data Set Four

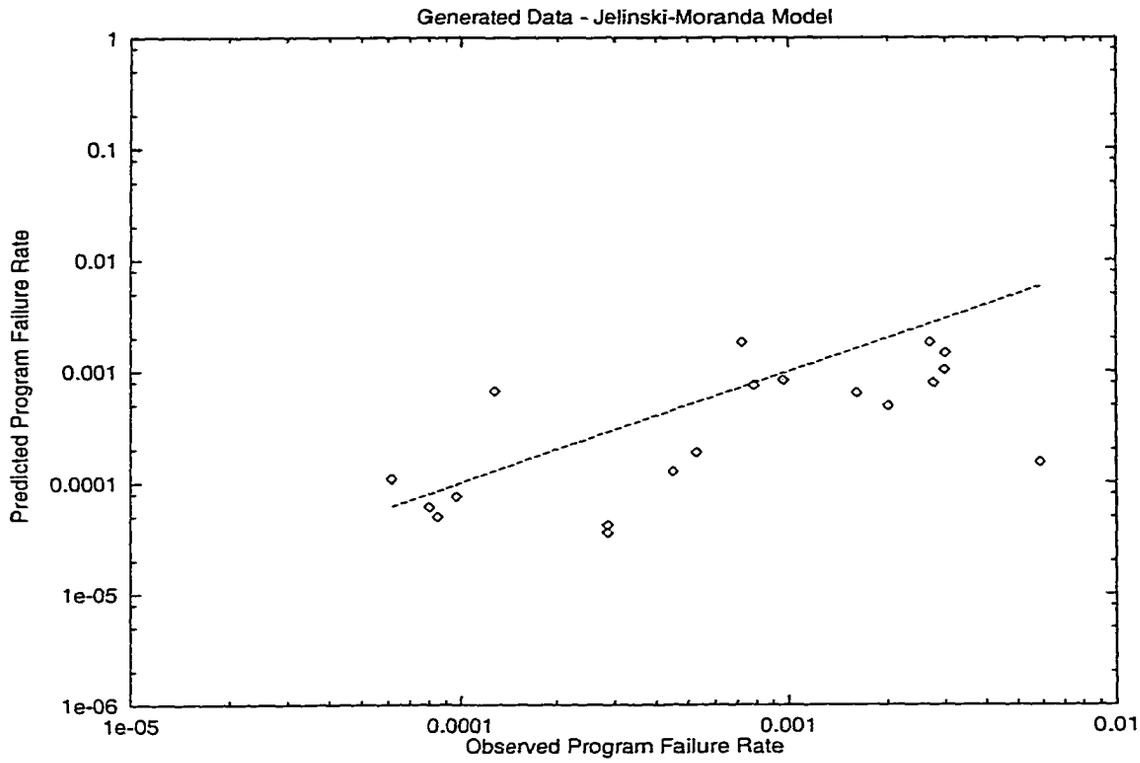

FIG. 71. JM Model OP Plot For Generated Data Set Four

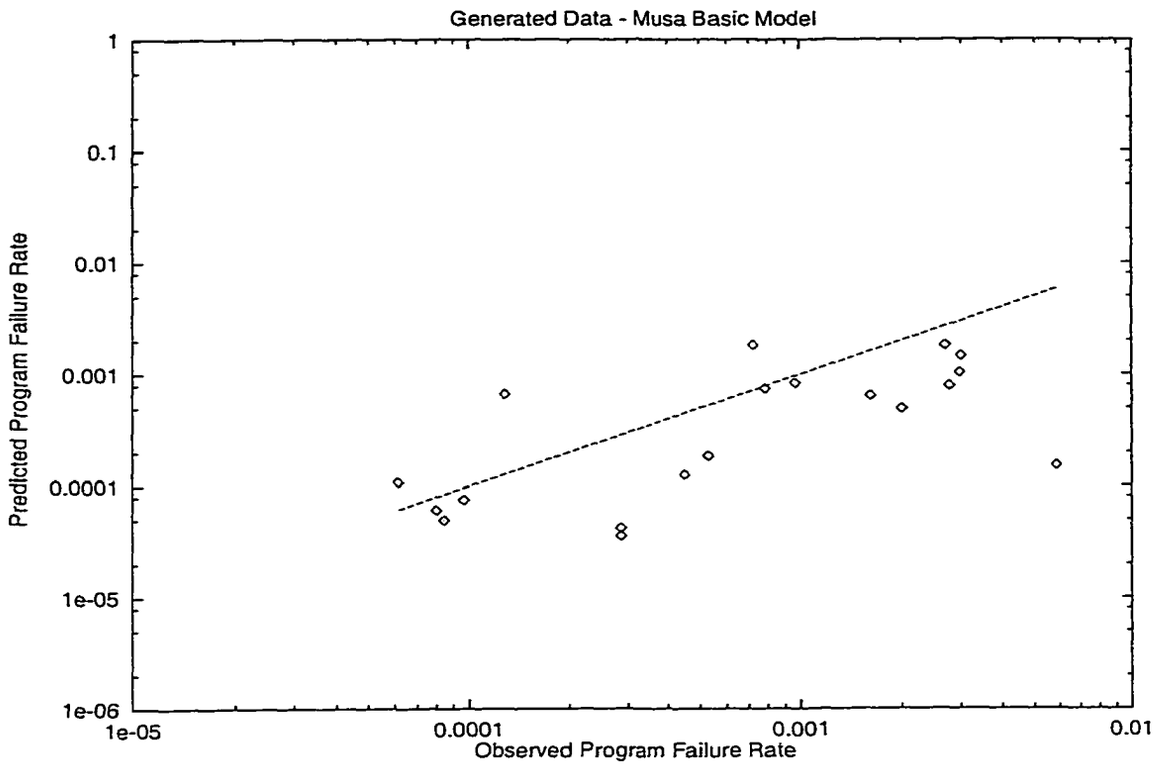

FIG. 72. MB Model OP Plot For Generated Data Set Four

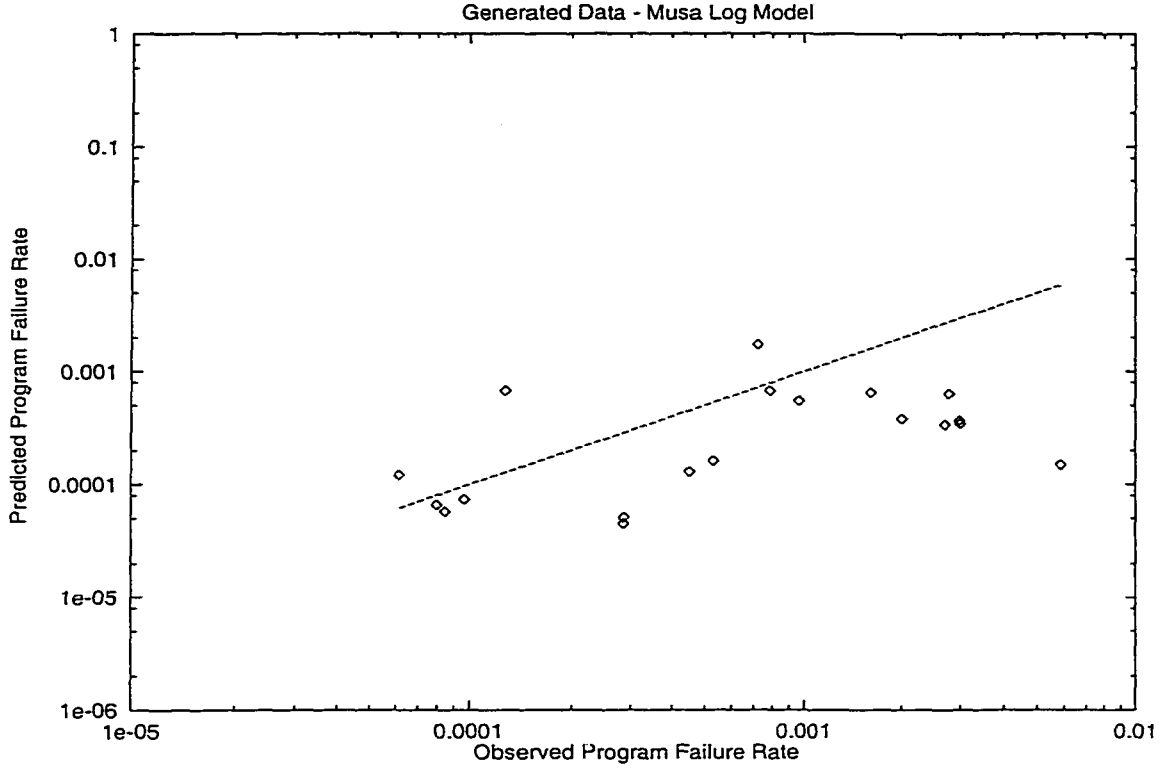

FIG. 73. ML Model OP Plot For Generated Data Set Four

Model	JM	MB	ML	OS
Set 1	21.9155	21.9155	18.43406	15.24542
Set 2	15.91654	15.91654	8.871714	7.095273
Set 3	15.46484	15.46484	13.69872	11.33527
Set 4	17.21274	17.21274	13.63313	10.99466

FIG. 74. Error For The OP Plots

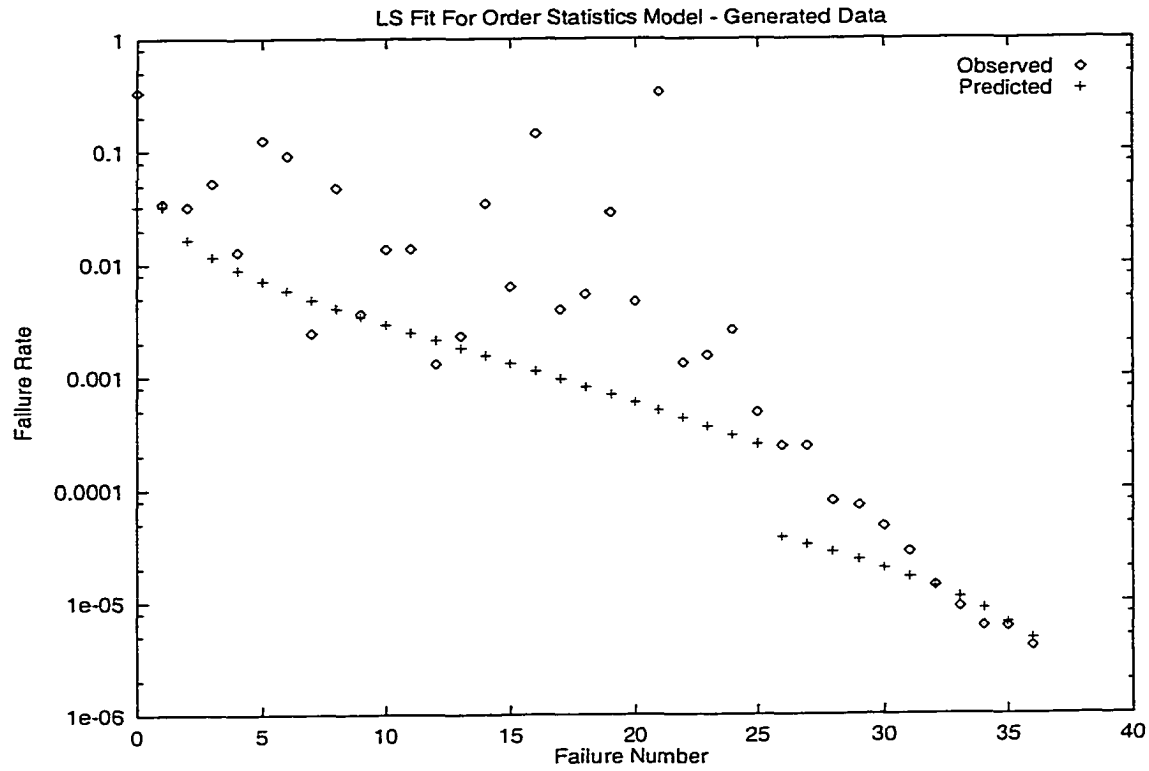

FIG. 75. Best Fit For OS Model Simulated Data Set One

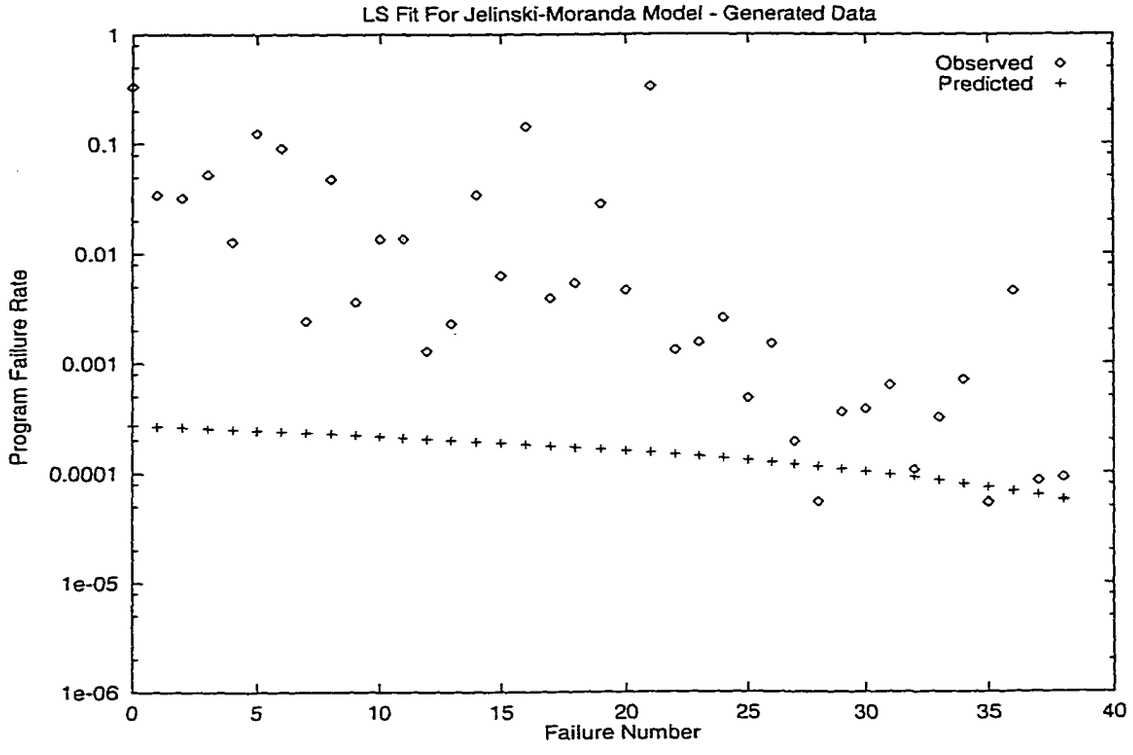

FIG. 76. Best Fit For JM Model Simulated Data Set One

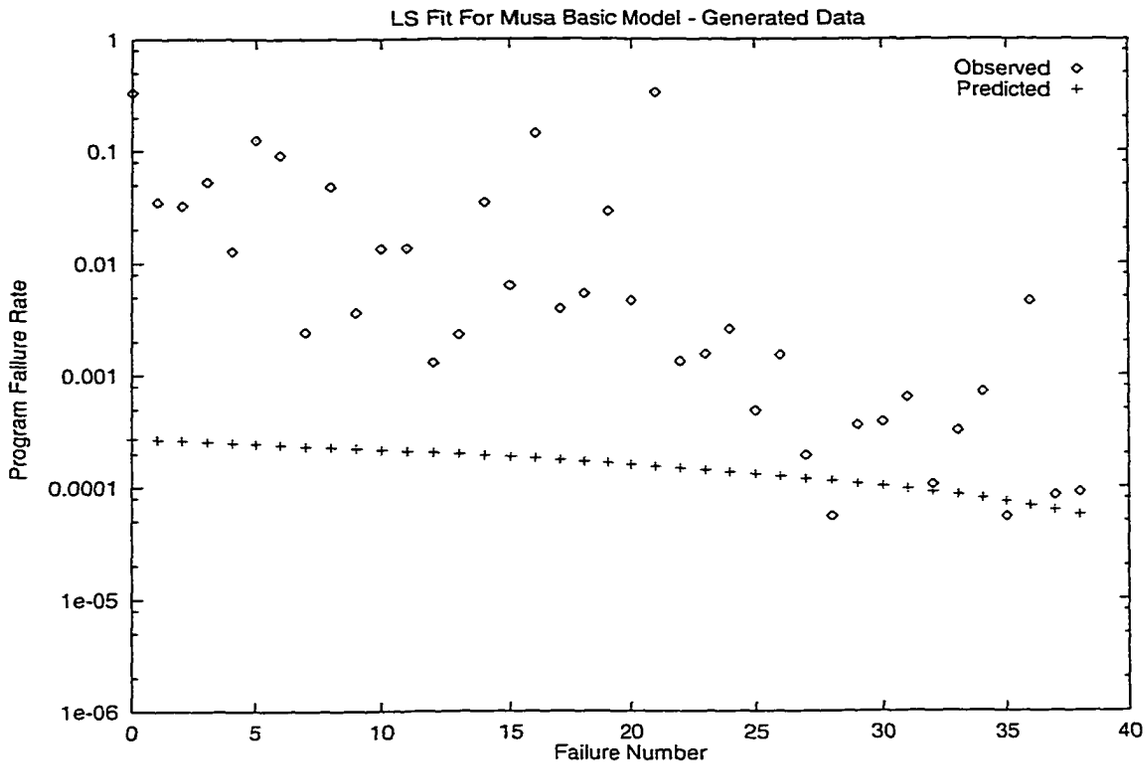

FIG. 77. Best Fit For MB Model Simulated Data Set One

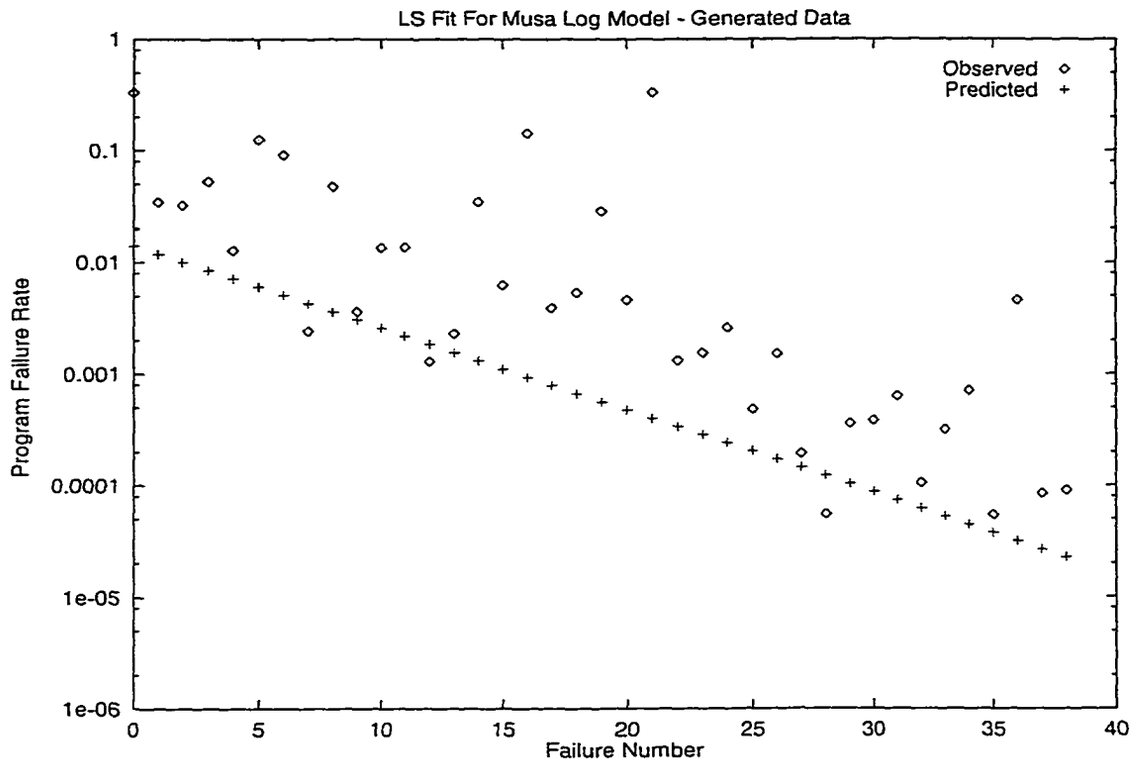

FIG. 78. Best Fit For ML Model Simulated Data Set One

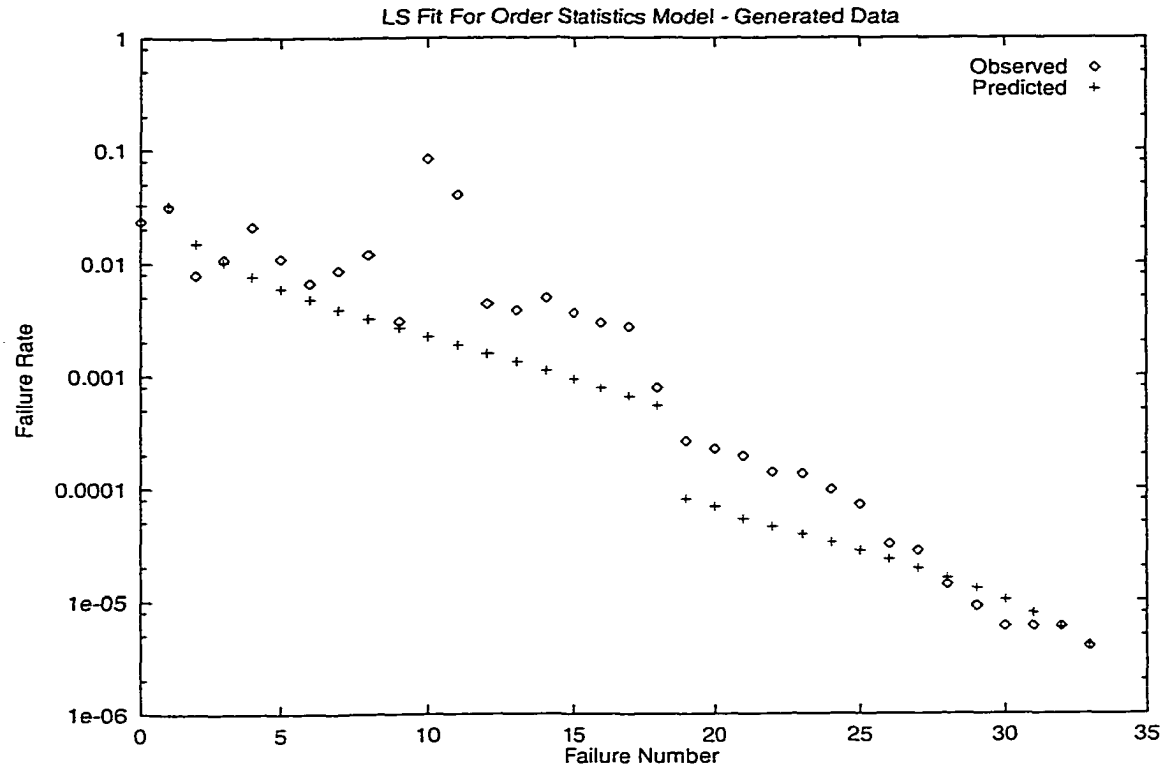

FIG. 79. Best Fit For OS Model Simulated Data Set Two

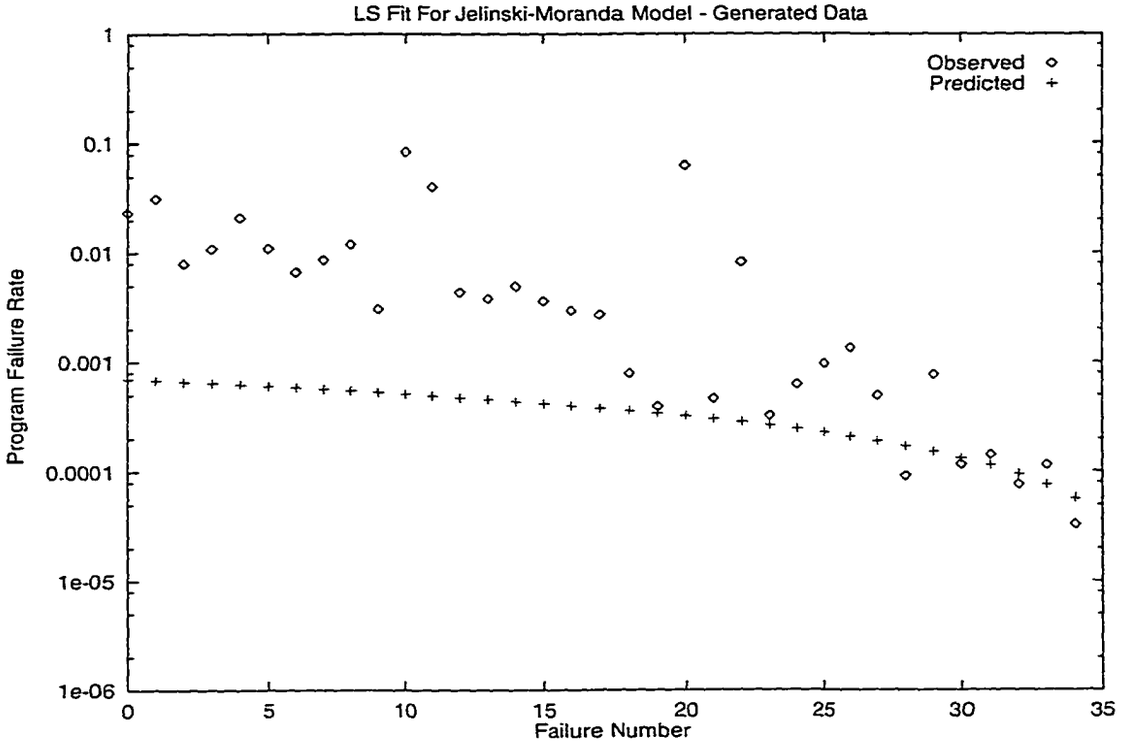

FIG. 80. Best Fit For JM Model Simulated Data Set Two

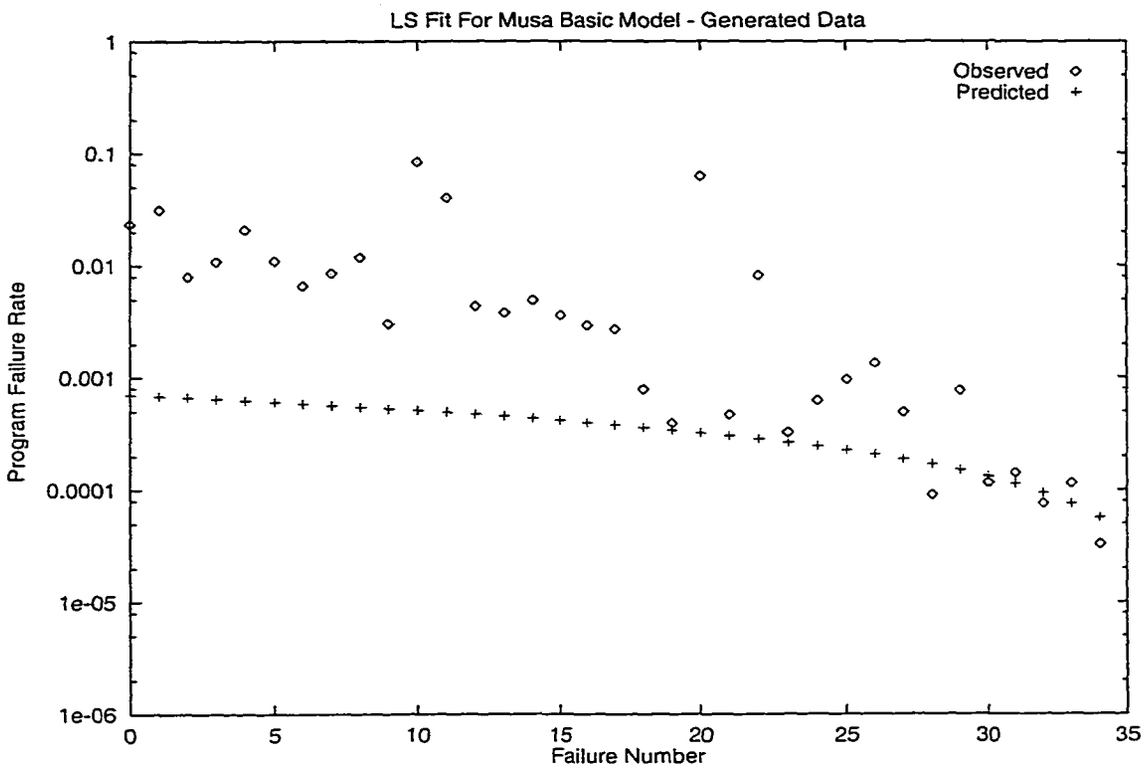

FIG. 81. Best Fit For MB Model Simulated Data Set Two

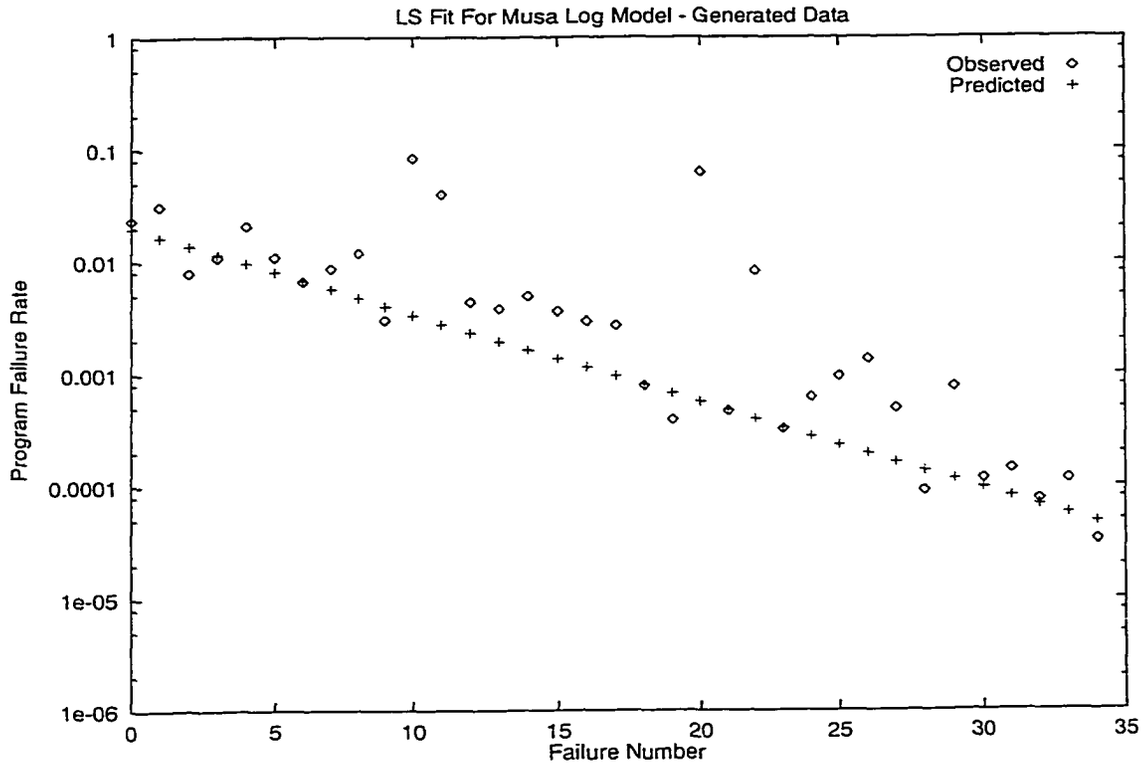

FIG. 82. Best Fit For ML Model Simulated Data Set Two

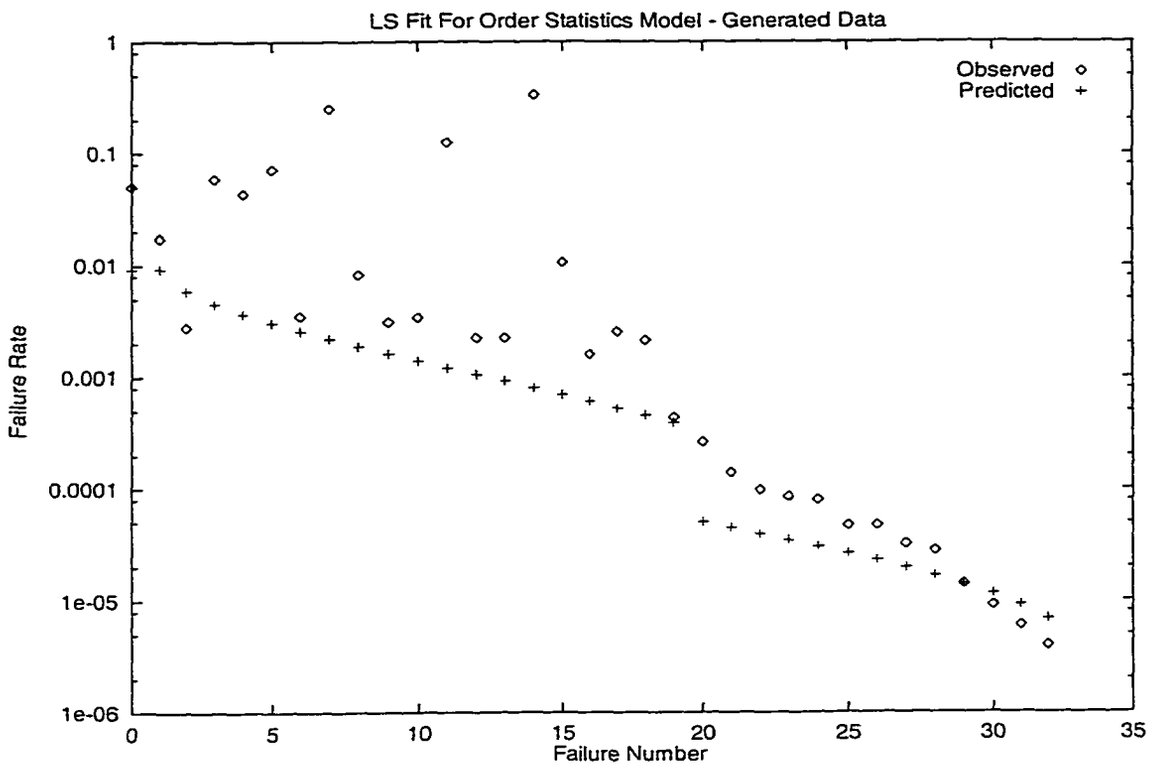

FIG. 83. Best Fit For OS Model Simulated Data Set Three

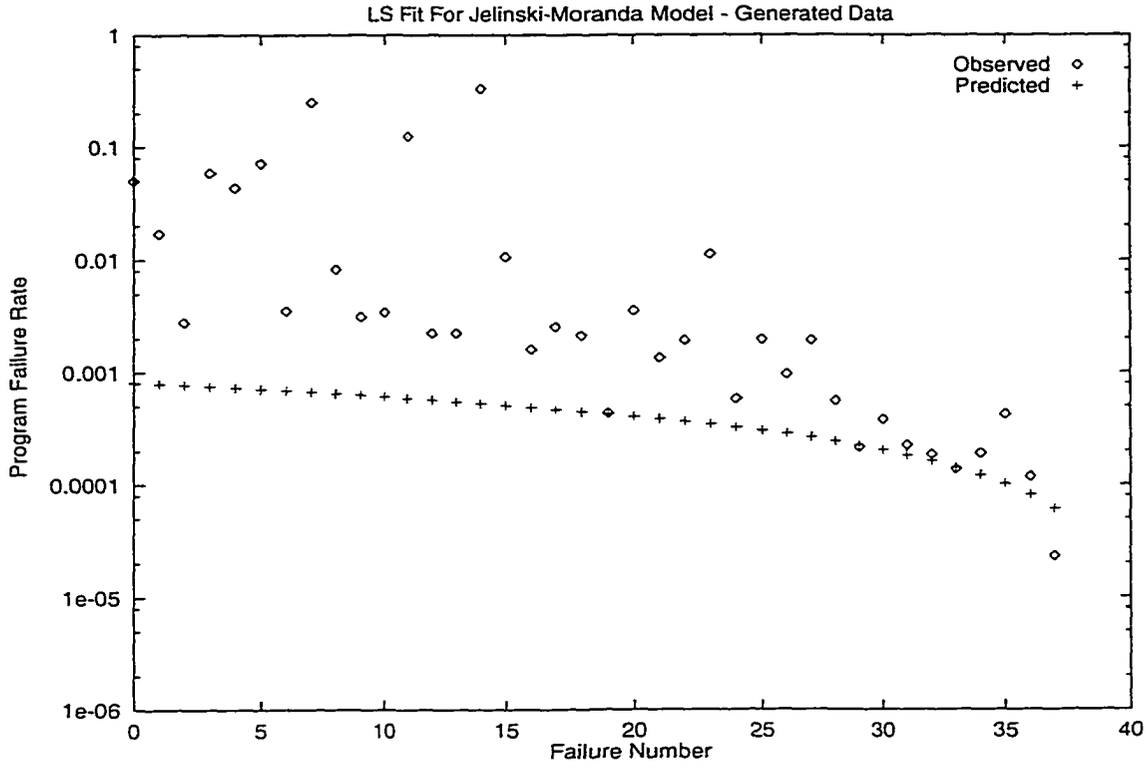

FIG. 84. Best Fit For JM Model Simulated Data Set Three

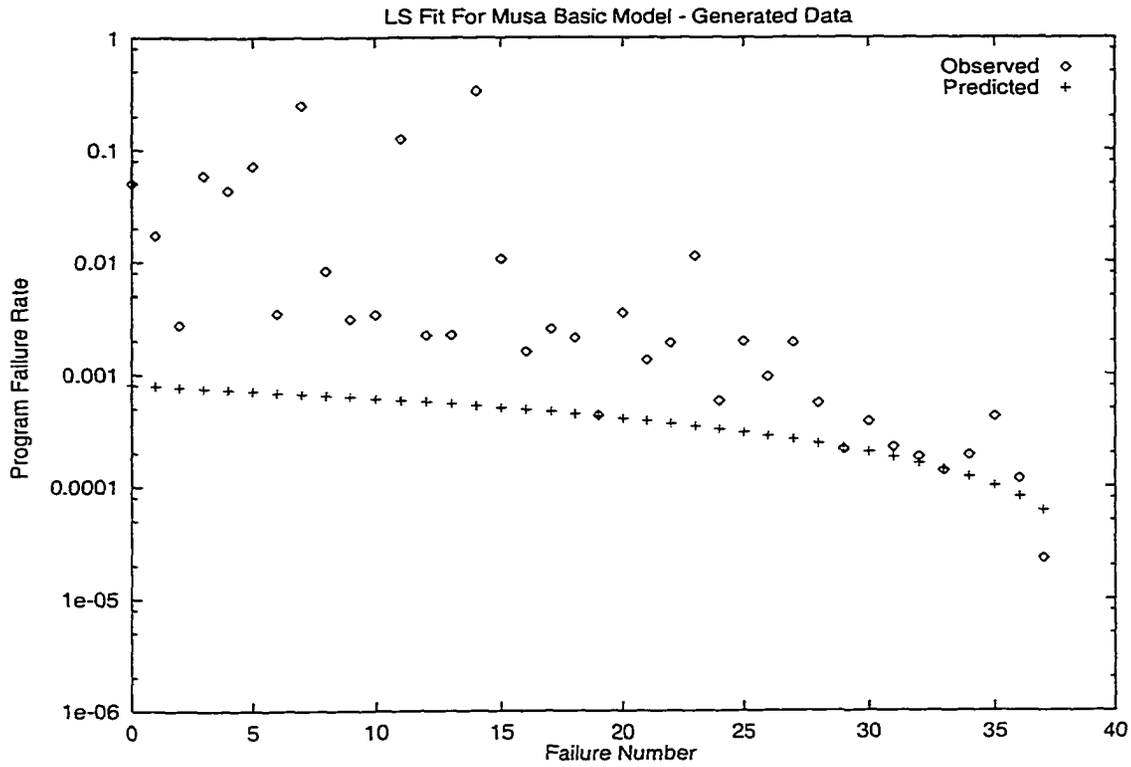

FIG. 85. Best Fit For MB Model Simulated Data Set Three

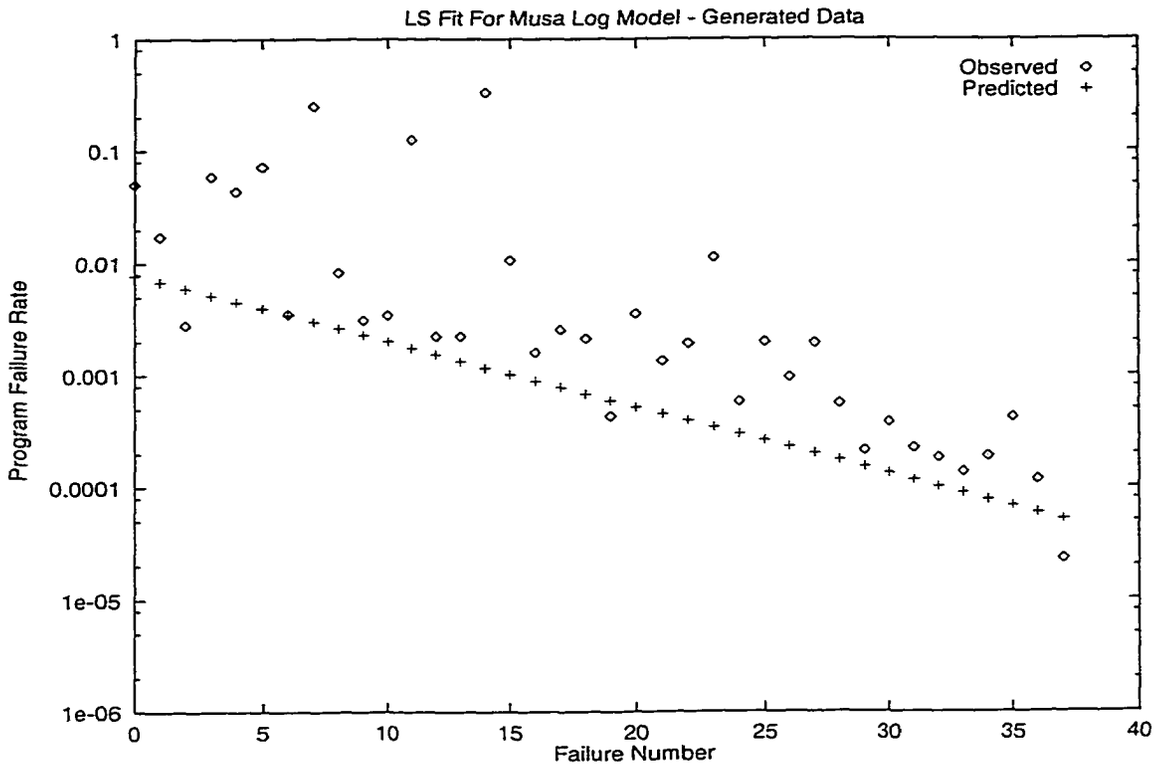

FIG. 86. Best Fit For ML Model Simulated Data Set Three

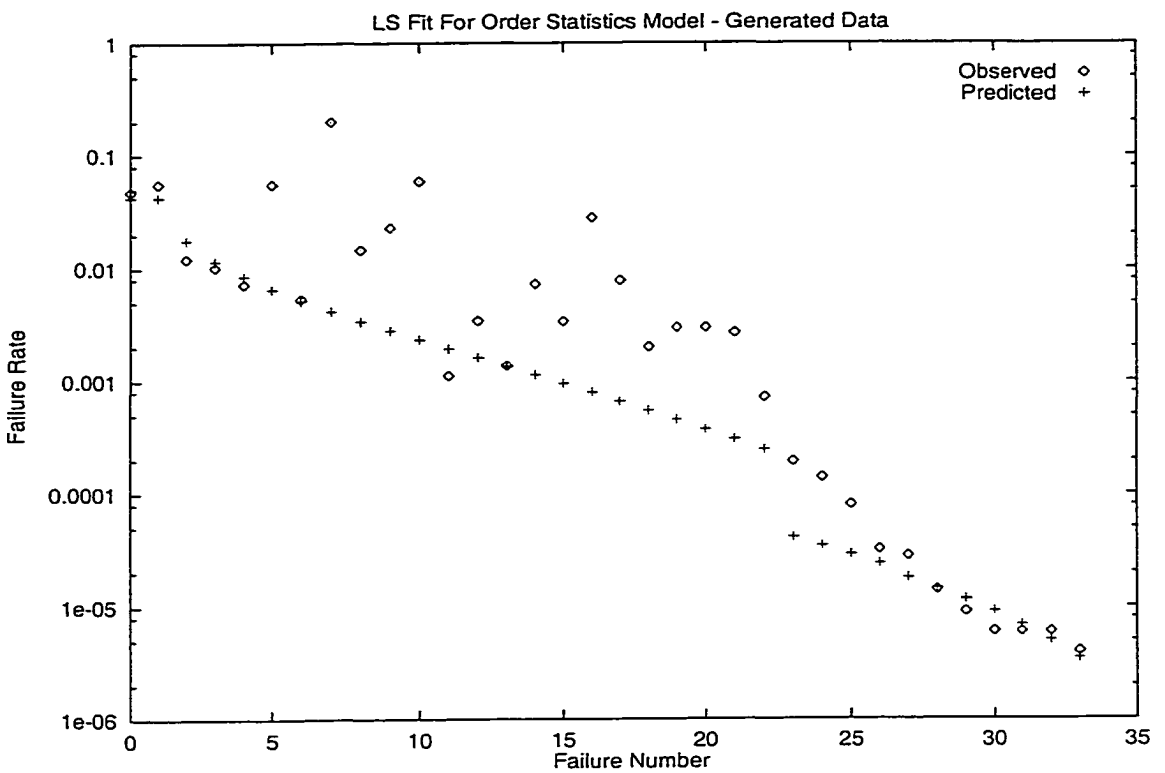

FIG. 87. Best Fit For OS Model Simulated Data Set Four

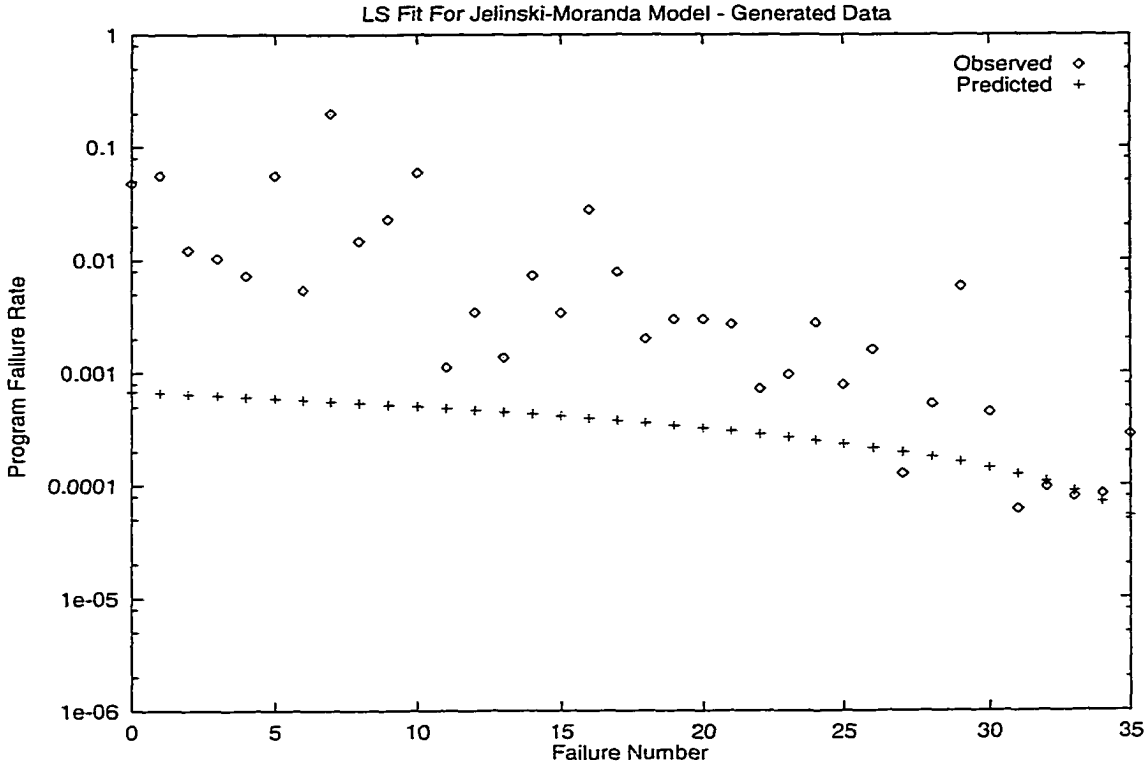

FIG. 88. Best Fit For JM Model Simulated Data Set Four

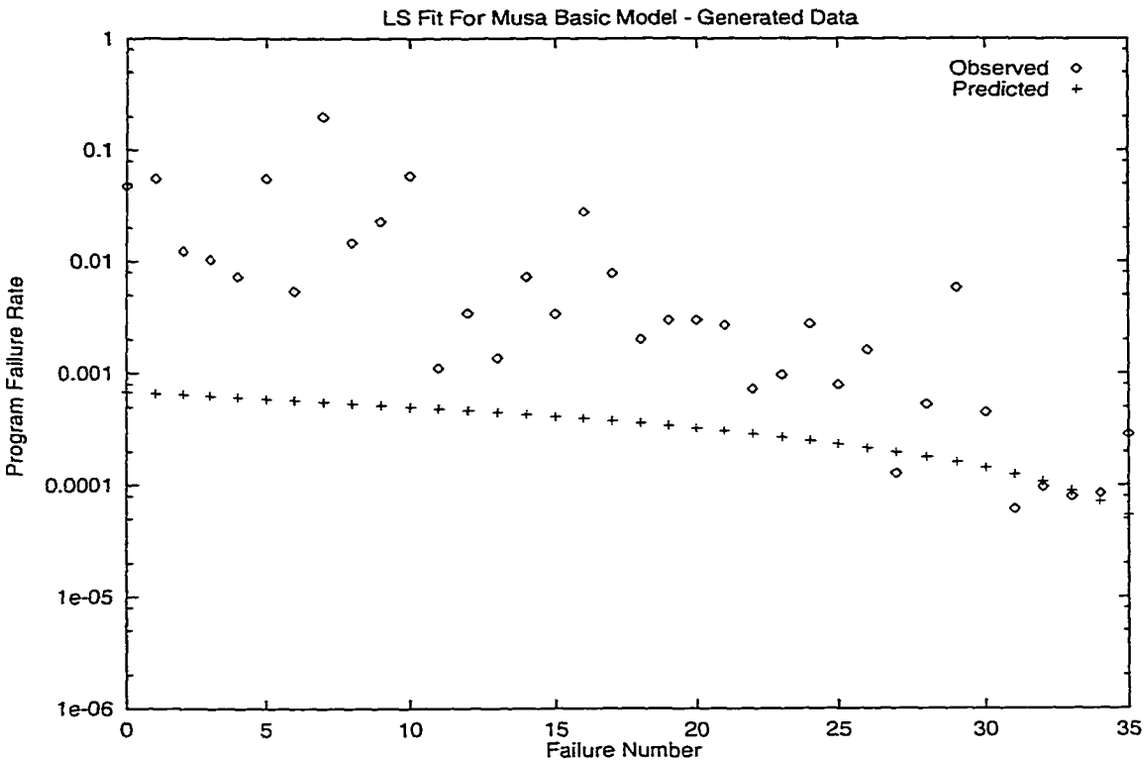

FIG. 89. Best Fit For MB Model Simulated Data Set Four

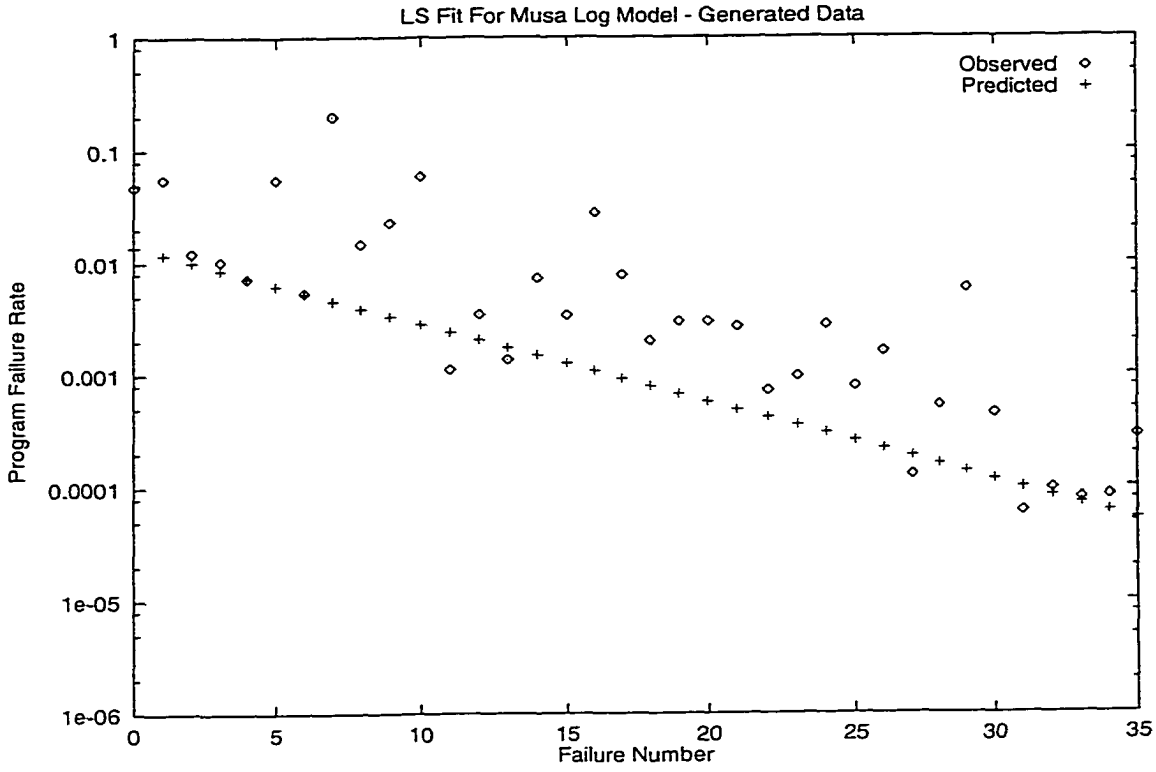

FIG. 90. Best Fit For ML Model Simulated Data Set Four

Model	JM	MB	ML	OS
Set 1	30.3649	30.3649	24.8865	18.738
Set 2	22.0017	22.0017	11.7156	11.7305
Set 3	24.0289	24.0289	20.3609	15.9356
Set 4	25.1962	25.1962	17.5946	13.5606

FIG. 91. Error For The Fits To The Full Data Set

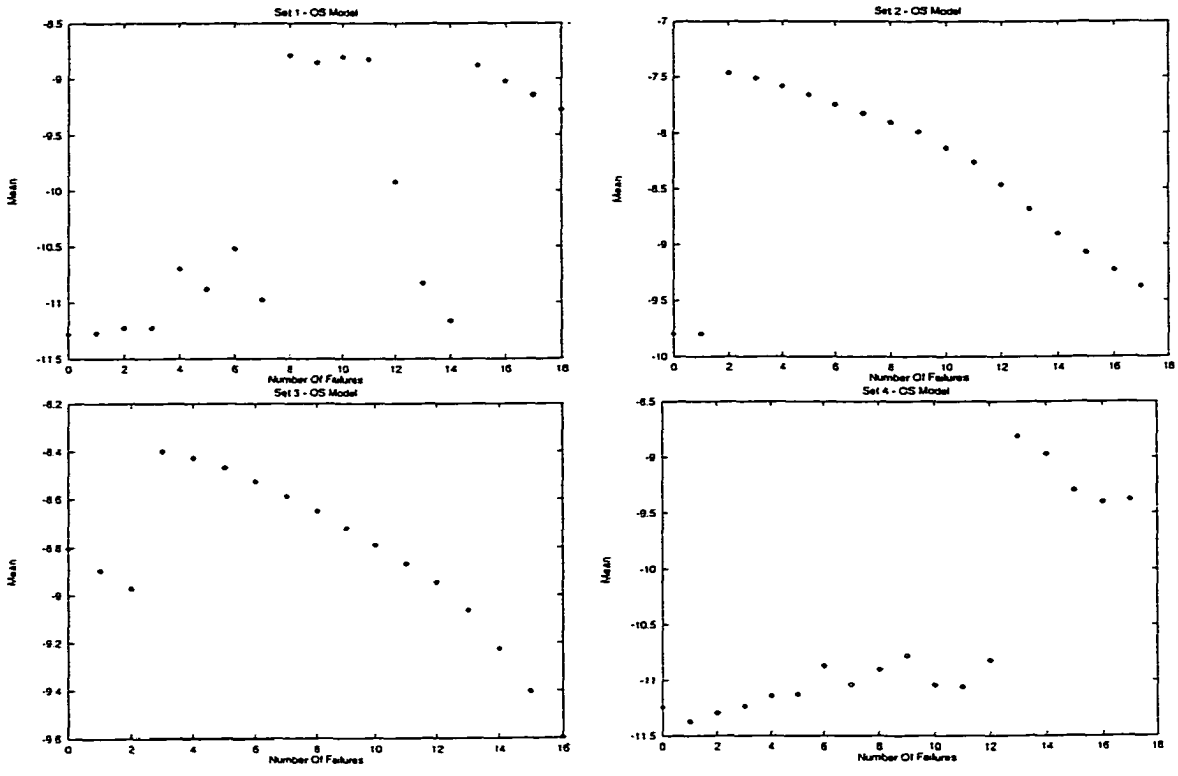

FIG. 92. OS Model Mean Progression For All Data Sets

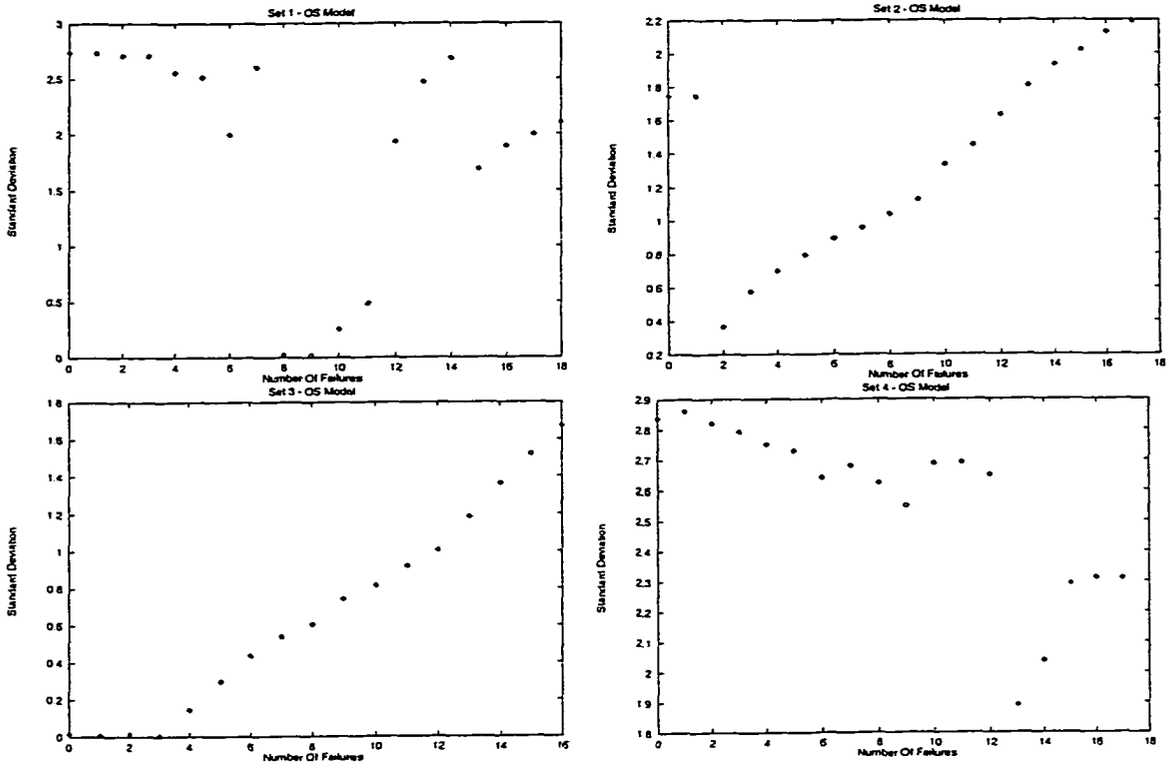

FIG. 93. OS Model Standard Deviation Progression For All Data Sets

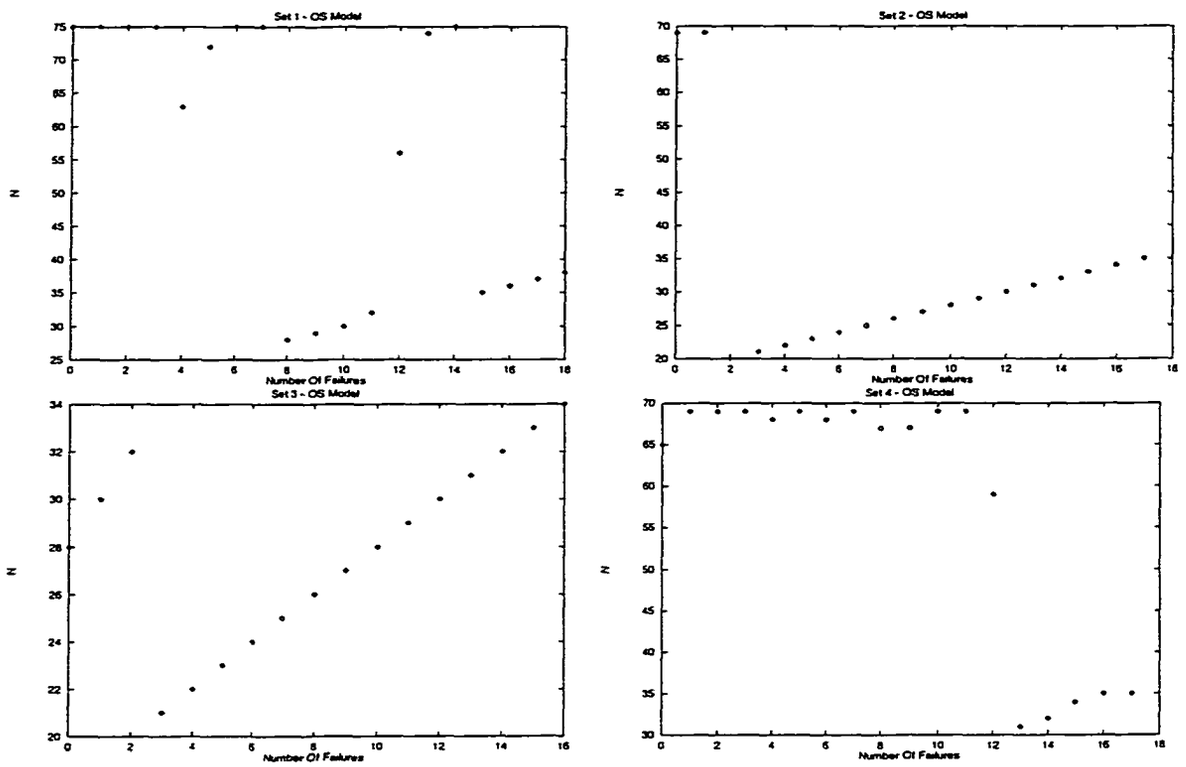

FIG. 94. OS Model N Progression For All Data Sets

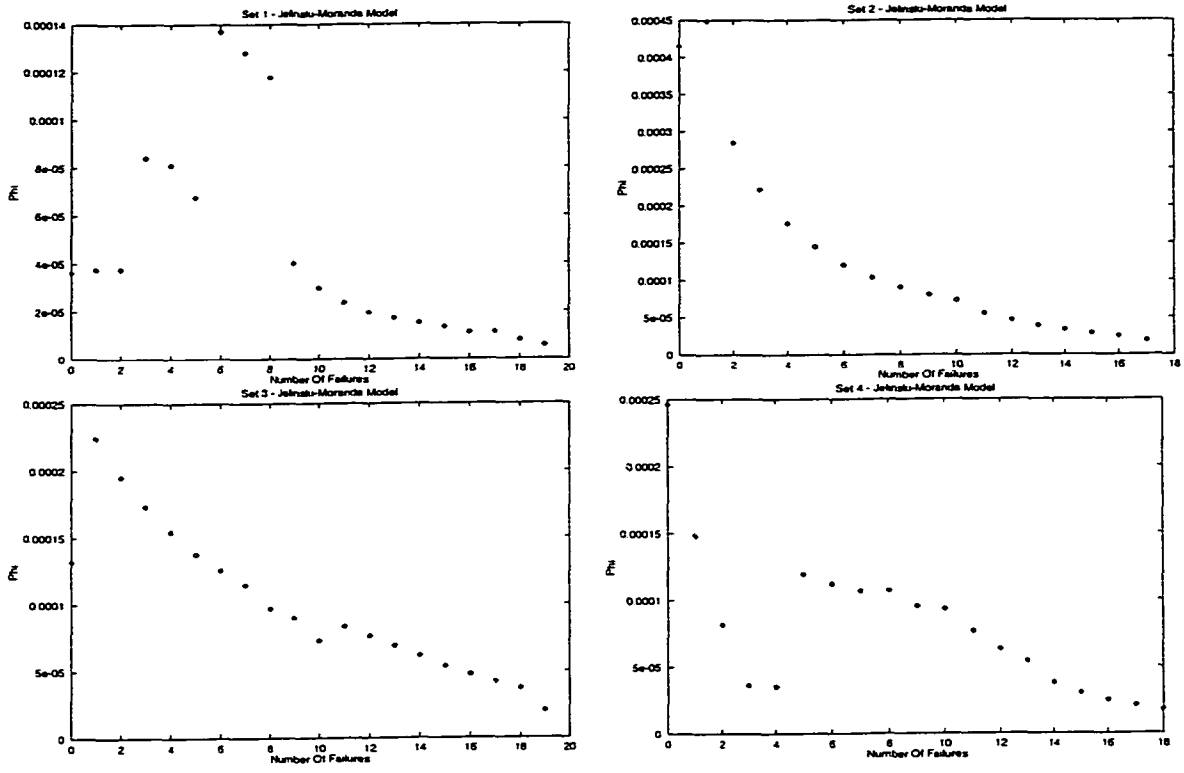

FIG. 95. Jelinski-Moranda Phi Progression For All Data Sets

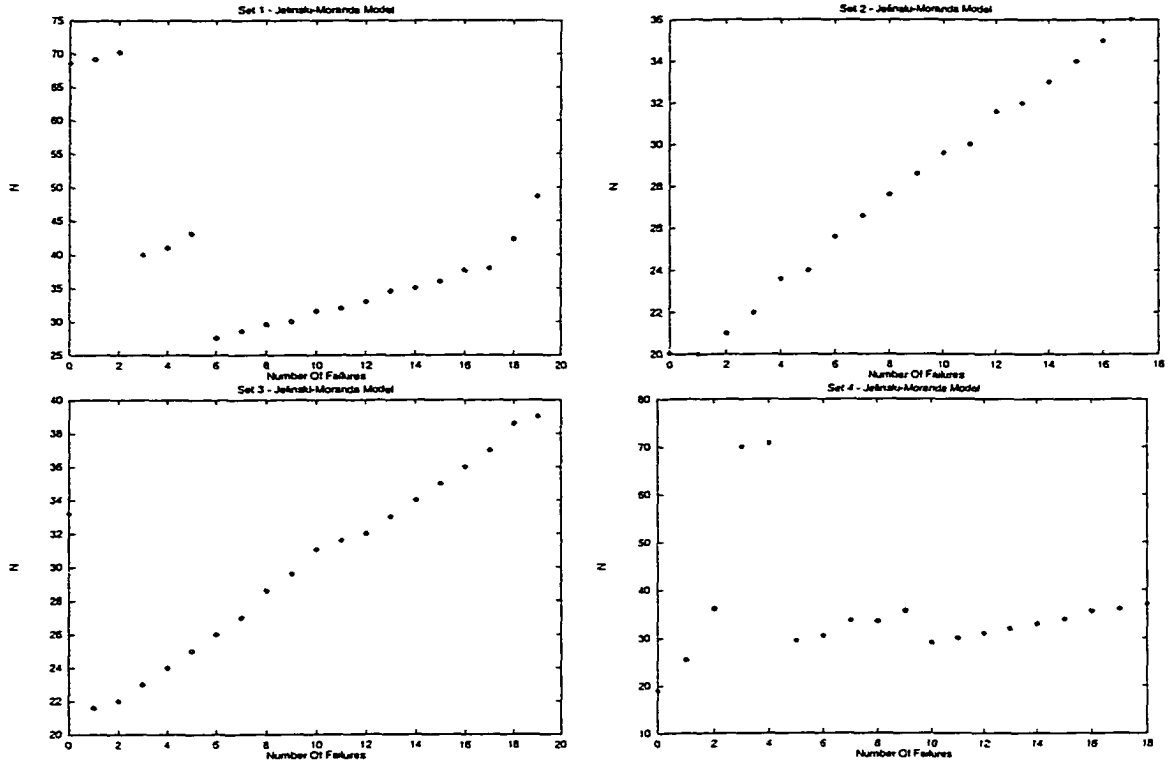

FIG. 96. Jelinski-Moranda N Progression For All Data Sets

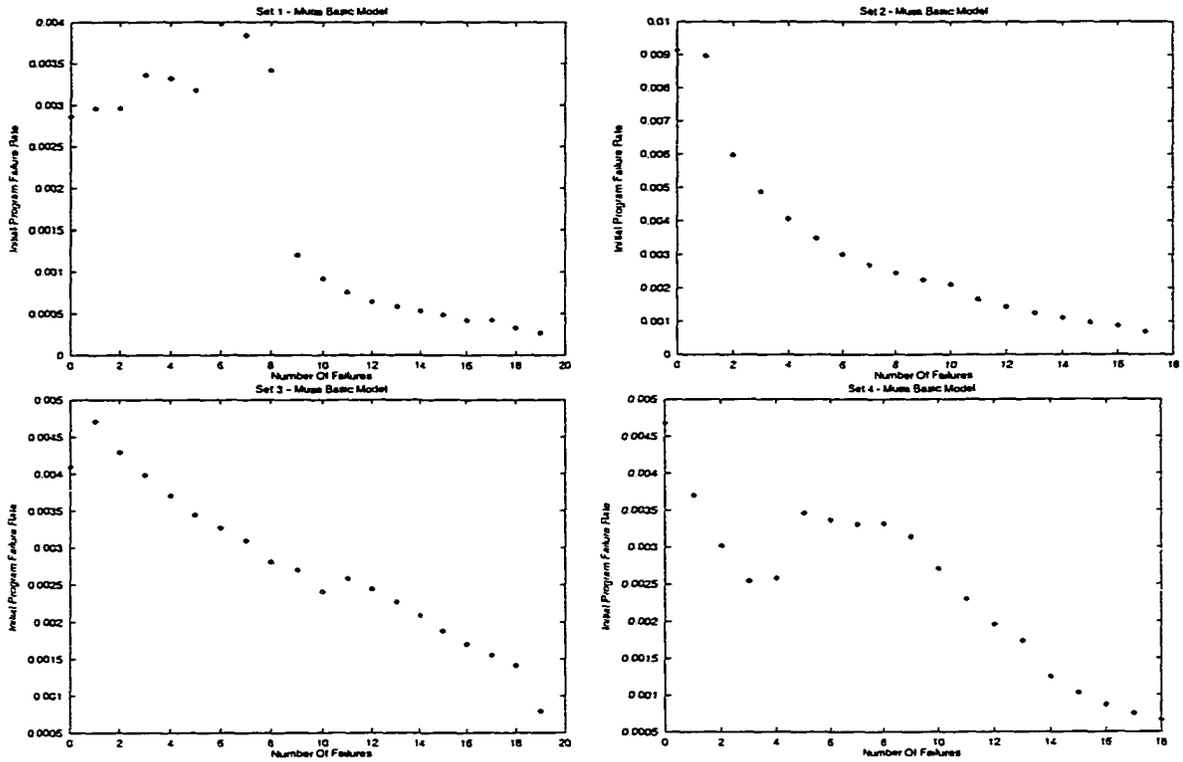

FIG. 97. Musa Basic Initial Program Failure Rate Progression For All Data Sets

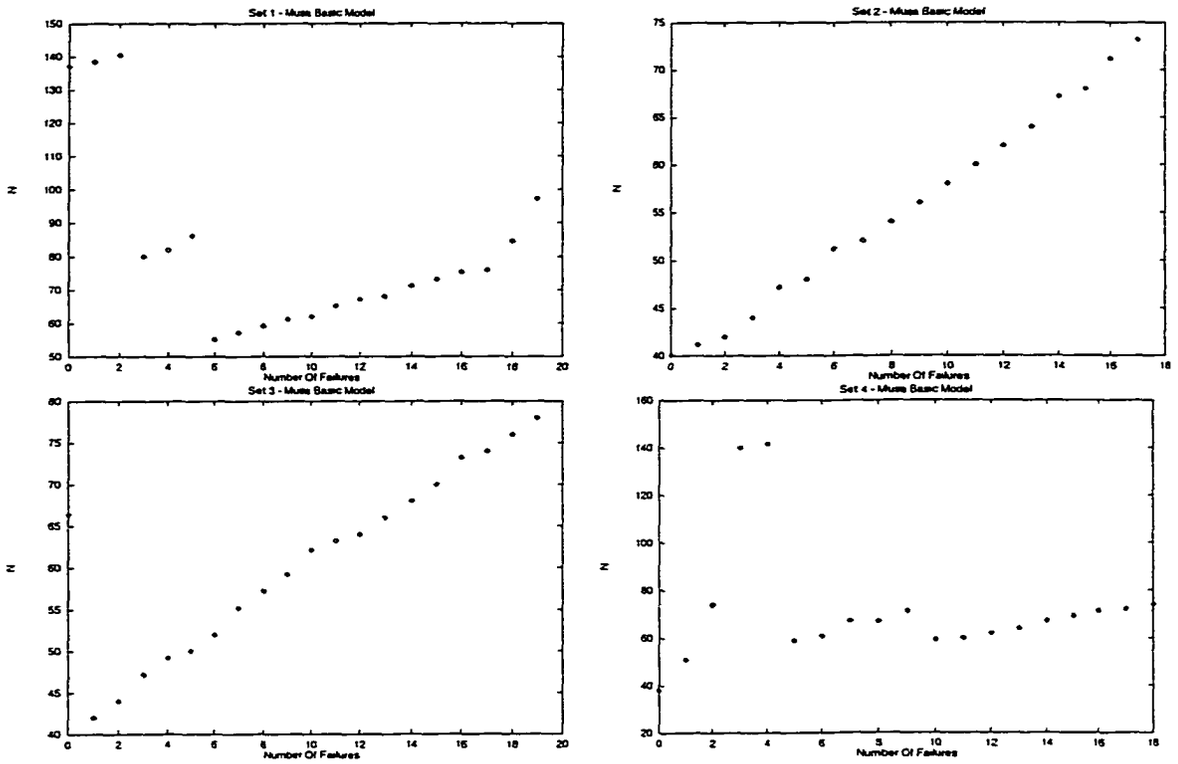

FIG. 98. Musa Basic N Progression For All Data Sets

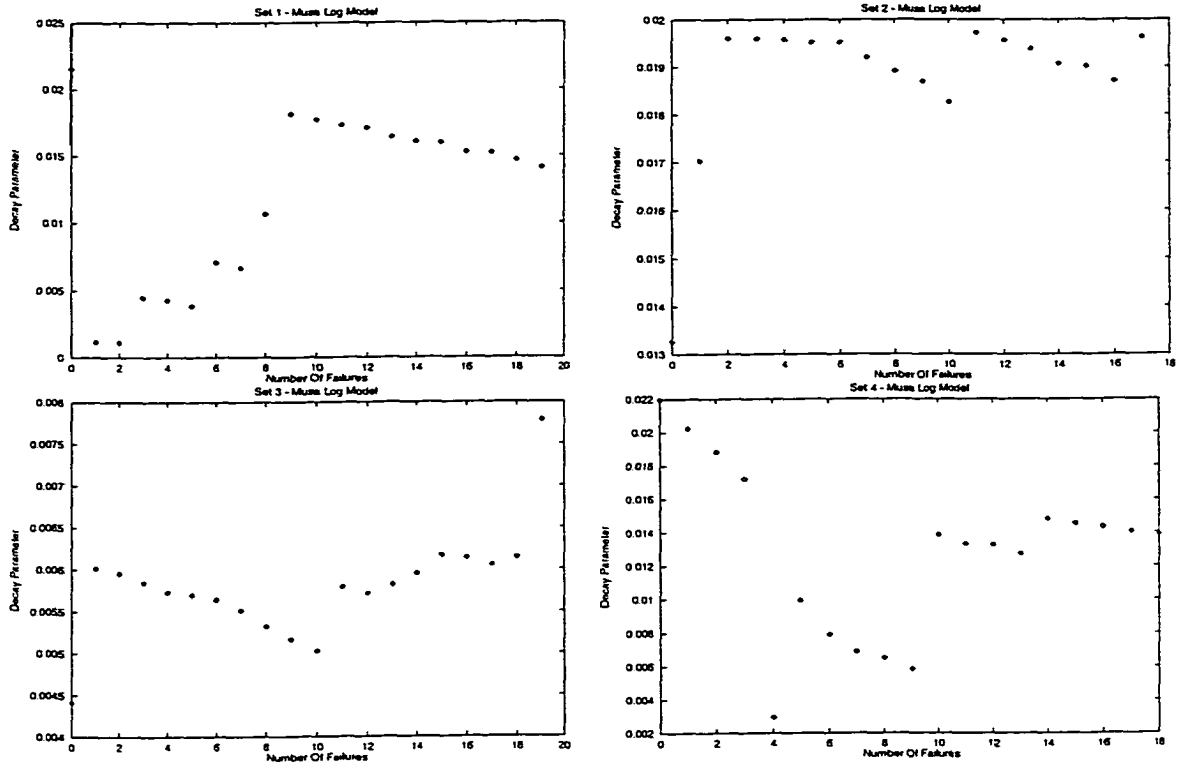

FIG. 99. Musa Log Decay Parameter Progression For All Data Sets

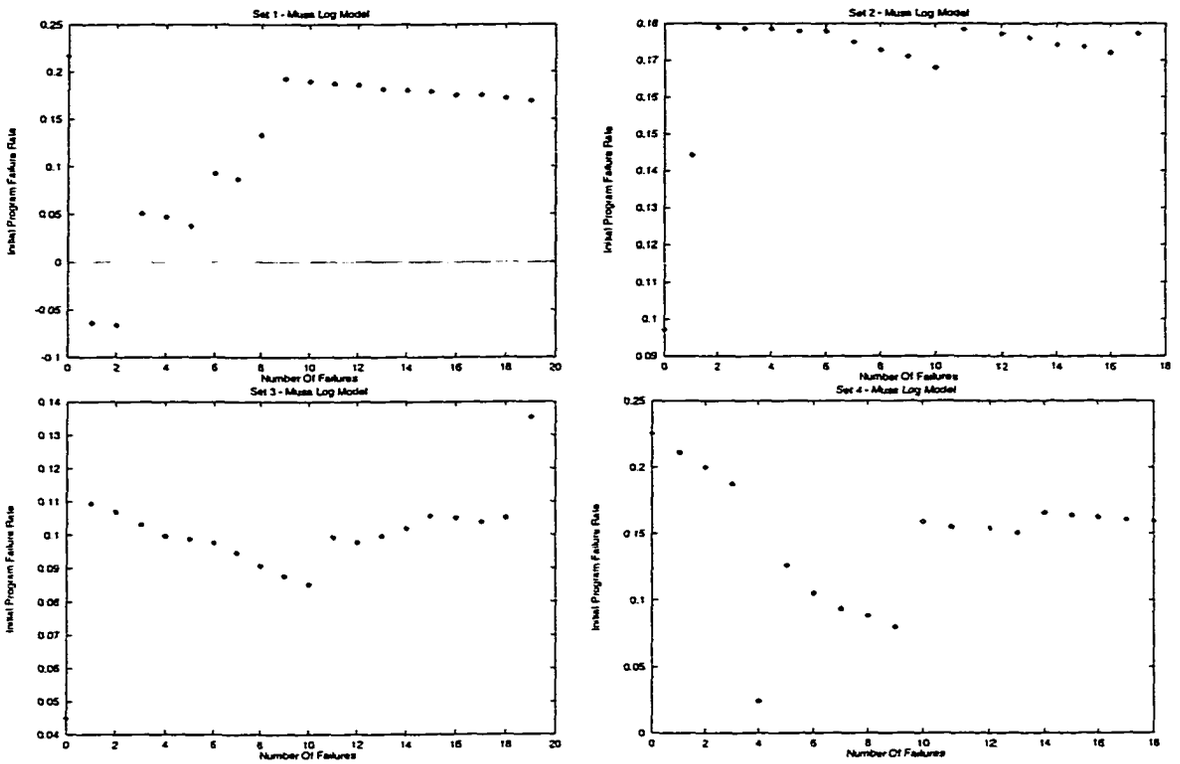

FIG. 100. Musa Log Initial Program Failure Rate Progression For All Data Sets

Chapter 12

Testing An Existing Software System

The experiments described in the two previous chapters used pre-existing failure data as input into the models. In order to gain more insight into the reliability testing process using the mixed method approach to testing, the next step in this work was to conduct an experiment that involved testing and debugging an existing software system. This chapter describes the experiment and the results that were obtained.

12.1 System Description

The software system chosen for this experiment is the server portion of a real-time alarm tracking software system used by Eastalco Aluminum Company. The plant being monitored has two production lines with a total of 480 identical pieces of equipment (pots) used in the process. For each of these pots, the values of about 125 process variables (temperatures, effects, etc.) are tracked and fed to a plant-wide data historian that stores the data in a relational database.

Once a minute, the alarm server software examines the data for each of the 240 pots in the production line and uses the values for 13 of the 125 available process variables to determine if any of eight different alarm conditions are met.

This alarm information is instantly available to plant management and plant floor personnel, who are running the client portion of the alarm software. Plant personnel are able to use the client software to quickly determine which pots are in alarm, and are able to select a given pot to receive a graphical presentation (time vs. value plot) of all of the significant process variables for that pot.

The server portion of this software system consists of two modules. The first module is a 600 line Delphi program responsible for providing the server's graphical interface. This module also provides support for interprocess communication with the client software. The second module is a 3500 line Dynamic Link Library (DLL) written using C++. This DLL is responsible for retrieving the required process data from the plant data historian and determining the alarm status for each pot in the production line.

The alarm server software was chosen for this experiment for several reasons. First, the list of bugs discovered in this system had been maintained since the software had been released for general use in the plant. Whenever a bug was found during system use and the fix was applied, the original (incorrect) code was kept in the source program as a comment. Thus, for this experiment, it was fairly straightforward to go back and construct a version of the software containing all of the known faults. A total of 13 known faults were documented in the software. A table detailing each of these faults will be presented later in this chapter.

Second, since the software had been running for several years in an industrial environment, a large amount of data was available to use for representative testing. The lack of representative data and the subsequent need to construct an operational profile to drive testing is one of the basic problems confronting software reliabilty researchers. Selection of this software system allowed us to avoid this problem.

Third, the nature of the outputs (byte values corresponding to a given alarm state) made the creation of an automated oracle possible and fairly straightforward. Some of the other software systems we considered for use in this experiment had output that would be much more difficult to compare between the gold and test versions. For example, one software system considered for this experiment

was a text formatting utility. This system had a list of bugs that would cause the appearance of the output document to differ between the test and gold version, but the problem of how to compare the output of the two versions automatically seemed difficult at best.

Fourth, the selected software consists of over 4000 lines of code and contains 13 faults. The smaller programs we considered contained only a few faults, which would not be useful for making reliability estimates.

Finally, the selected software is an application that is used in a real-world environment, which made it much more attractive than using other alternatives, such as student programs.

12.2 Experiment Setup

The experiment described in this chapter was intended to provide the software reliability models under investigation with failure data from the software system under test. Setting up this experiment required several steps. This section describes each of these steps.

12.2.1 Obtaining Representative Data

The first step in setting up this experiment was to obtain representative data to use in the testing process. A program was developed to extract one year of representative data for each of the required process values from the plant data historian at the plant where the software had been running for several years.The extracted data was placed on a CD-R disk and was taken to the testing site where it was transferred to a hard drive for use by the input driver of the experiment.

12.2.2 Setting Up The Input Driver

Once the representative data had been obtained from the plant, the next step was to devise a way to feed this data to the alarm server in the test environment. At the plant, the data historian software receives input from the pots and stores the

data in a relational database. The alarm server software then retrieves the process data directly from this database and uses it in its calculations.

For this experiment, the driver that feeds the plant data to the data historian was modified to retrieve its data from a file-based source instead of directly from the pots. The input files used by this modified driver are text files containing representative data extracted from the database files obtained from the plant. This data is fed to the plant data historian, which stores it in its database system (just as it does at the plant). The alarm server then is able to retrieve this data for its own use.

12.2.3 Automating The Failure Detection Process

In order to automate the failure detection process, a gold version of the software and an oracle were required. The gold version was readily obtained, as it is merely the alarm server software compiled with all of the known faults removed. Conditional compilation flags were utilized to allow a single code base to be used to build both the gold and the test versions of the software.

An oracle was created to automate the process of checking the outputs of the test version against the ouputs of the gold version. This oracle consists of a graphical interface showing the alarm status of each of the pots in the gold and test systems. Whenever a discrepancy is detected between the gold and test versions, the oracle stops processing and outputs a log file detailing the time of the failure, the number of test cases that have been run before the failure occurred, and the nature of the failure.

12.3 Estimating Fault Failure Rates

In order to use the data from the experiment with the mixed method approach to testing, it was neccesary to determine the operational fault failure rates for each of the thirteen faults present in the test software. The fault failure rate for each fault was estimated by analyzing one month of representative data (44460 values

Fault	Description	FFR
1	Incorrect Tag Name For Breaker Faults	0.020923
2	Wrong zone name for Swing	0.000067204
3	Incorrect comparison for AE ACK	0.000224014
4	Incorrect operator for comparison to AE's limit	0.2564516
5	Array off by one in swing queue wraparound	0.000002240
6	Incorrect comparison for swing active vals	0.00067204
7	Incorrect comparison for Soda Ash on alarm.	0.0291667
8	Incorrect comparison for Soda Ash off alarm.	0.000067204
9	Incorrect operator for comparison to BF on limit.	0.020923
10	Incorrect operator for comparison to BF off limit.	0.0146461
11	Incorrect comparison for swing active vals	0.000000093
12	Incorrect type for minute vals	0.000067204
13	Possibility of negative number being passed to sleep function, resulting in an infinite loop.	0.0000113811728

FIG. 101. The Fault Set

for each input tag) to determine the number of times each fault would manifest as a failure. The set of faults (along with the each fault failure rate) is shown in Figure 101

As an example of how the fault failure rates were calculated, let us consider fault number seven, (Incorrect Operator For Soda Ash Alarm On). During the estimation process, it was determined that this fault will manifest as a failure whenever the soda ash tag has a value of one. Therefore, the failure rate estimation program was modified to calculate the percentage of time that the soda ash tag had a value of one during the month of data that was analyzed. This percentage (.029166667) was used as the estimated fault failure rate for this fault.

As a second (slightly different) example, let us consider fault number eight, (Incorrect Operator For Soda Ash Alarm Off). During the estimation process, it was determined that this fault will manifest as a failure whenever the soda ash alarm acknowledgement tag has a value of one AND the Soda Ash Alarm is active. The fault failure rate (.000067204) was calculated by scanning one month of representative data to calcuate the percentage of time that this combination of input values occurred.

Fault	Interfailure Time
1	180
10	2130
9	3390
2	2370
6	2370
12	180
11	180
13	29370
7	40830
8	14490
4	180

FIG. 102. Faults Found During Representative Testing

12.4 Representative Testing

The first part of this experiment involved testing the software using representative methods. During this part of the experiment, the input driver was used to send representative data to the data historian for use by the test software. The test version of the software was initially compiled to contain all of the known faults. As testing was conducted, the oracle compared output of the gold version to the output of the test version. Whenever a discrepancy was found, the gold and test versions were stopped and the fault responsible for causing the failure was found and removed from the test version. The gold and test versions of the software were then restarted. This process was repeated for the duration of testing.

During the course of this experiment, a total of about 200,000 test cases were generated, executed, and tested for correctness. Testing was stopped after the test software processed 100,000 consecutive test cases without a failure.

The list of faults that were found during representative testing is shown in Figure 102. The first column of the table shows the fault number, as given in Figure 101. The second column of the table shows the interfailure time (in terms of number of test cases) for the fault. The faults in this table are listed in the same order in which they were found during this experiment.

12.4.1 Representative Testing Results

As in the experiments presented in previous chapters, the analysis of the models consisted of generating OP-Plots for each model for the data, comparing the best fits for each model, and comparing the stability of the models using parameter progression plots.

Comparing The Predictive Accuracy Of Each Model

The OP Plots for each model are shown in Figures 103 to 106. To obtain the OP Plots, the failure data was input into the models in incremental steps. For example, for the first pass of OP Plot generation, only five failure rates were used as input, and the models predicted the sixth failure rate. This prediction of the sixth failure rate was paired with the actual observed sixth failure rate and the result was plotted as a point on the OP Plot. This process was repeated, with an additional input point being added during each step, until all points were considered.

When looking at the OP Plots for the models, it appears that all of the models performed similarly under representative testing. These observations are supported by the data in Figure 107, which shows the relative error for the OP Plots for each model.

Comparing The Best Fits For Each Model

The best fits for each model under representative testing are shown in Figures 108 to 111. Once again, we see that all of the models performed similarly. The relative error for the best fits for is model is shown in Figure 112. From the graphs, it appears that all of the models were overly optimistic about the predicted program failure rate.

Comparing The Parameter Progressions For Each Model

The parameter progressions for each model are shown in Figures 113 to 121. These graphs show how the estimated values for each model parameter changed from

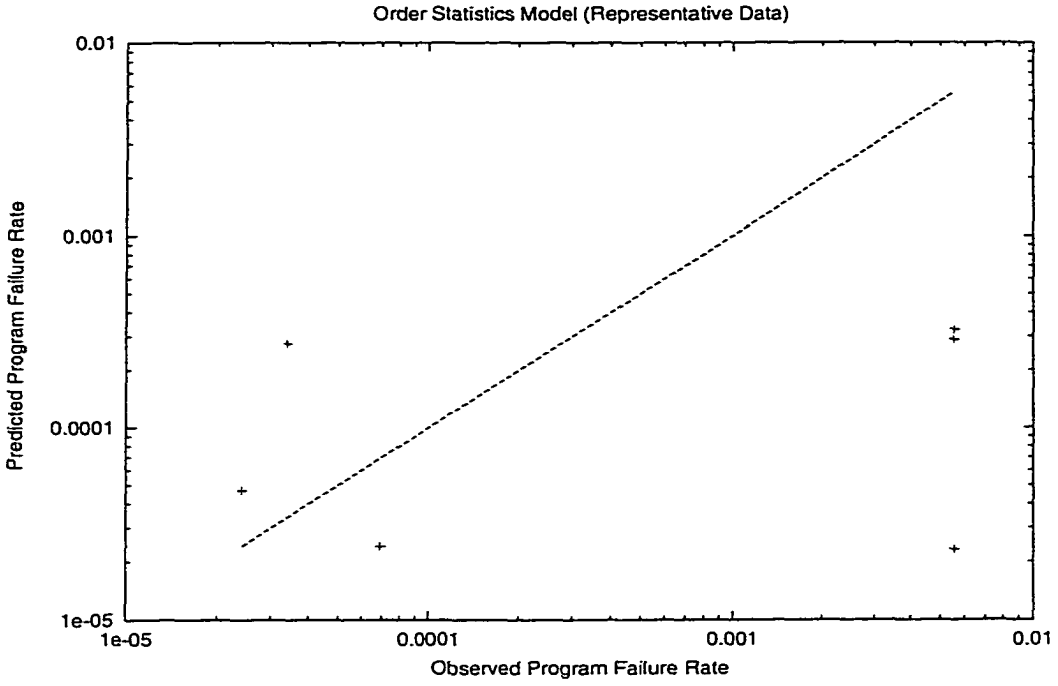

FIG. 103. OP Plot For Order Statistics Model (Representative Data)

one step of the OP Plot generation to the next. For all of the models under representative testing, the model parameters are fairly stable as long as the data is fairly stable. We see a change in the pattern for each parameter sequence when the number of failures is 6 on the X-Axis. This point in the data corresponds to the point where the observed interfailure time goes from 180 to 29370, which could be expected to cause a model to change its parameter estimates.

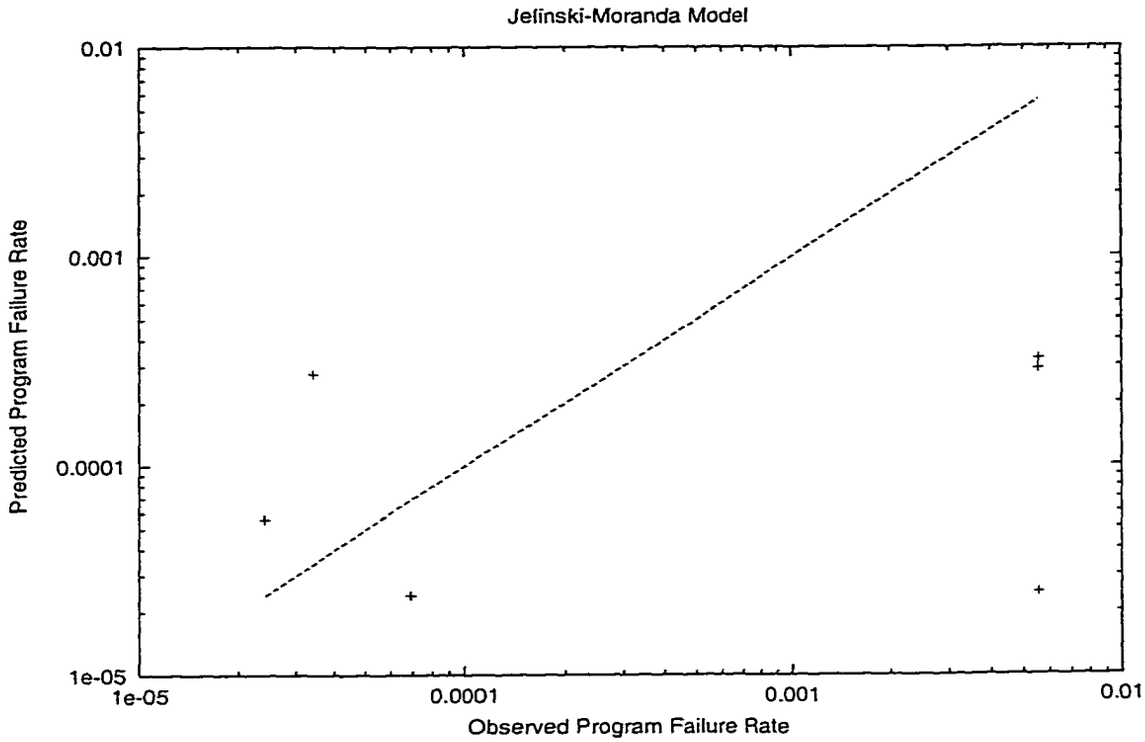

FIG. 104. OP Plot For Jelenski-Moranda Model(Representative Data)

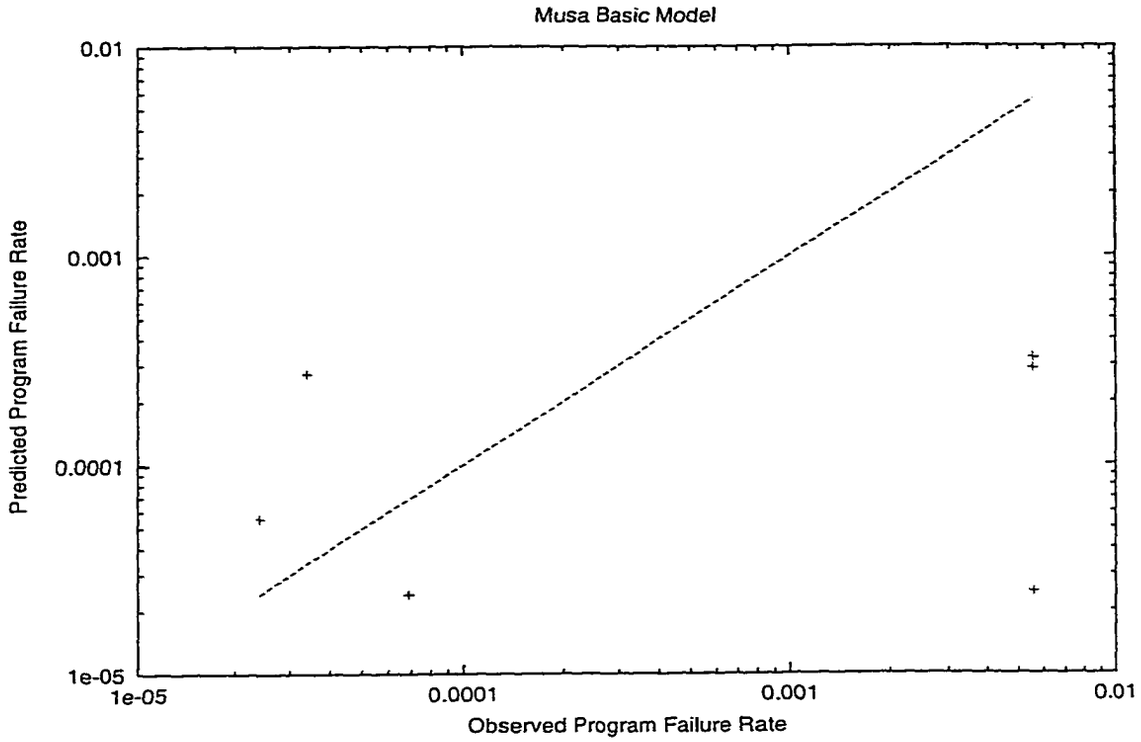

FIG. 105. OP Plot For Musa Basic Model (Representative Data)

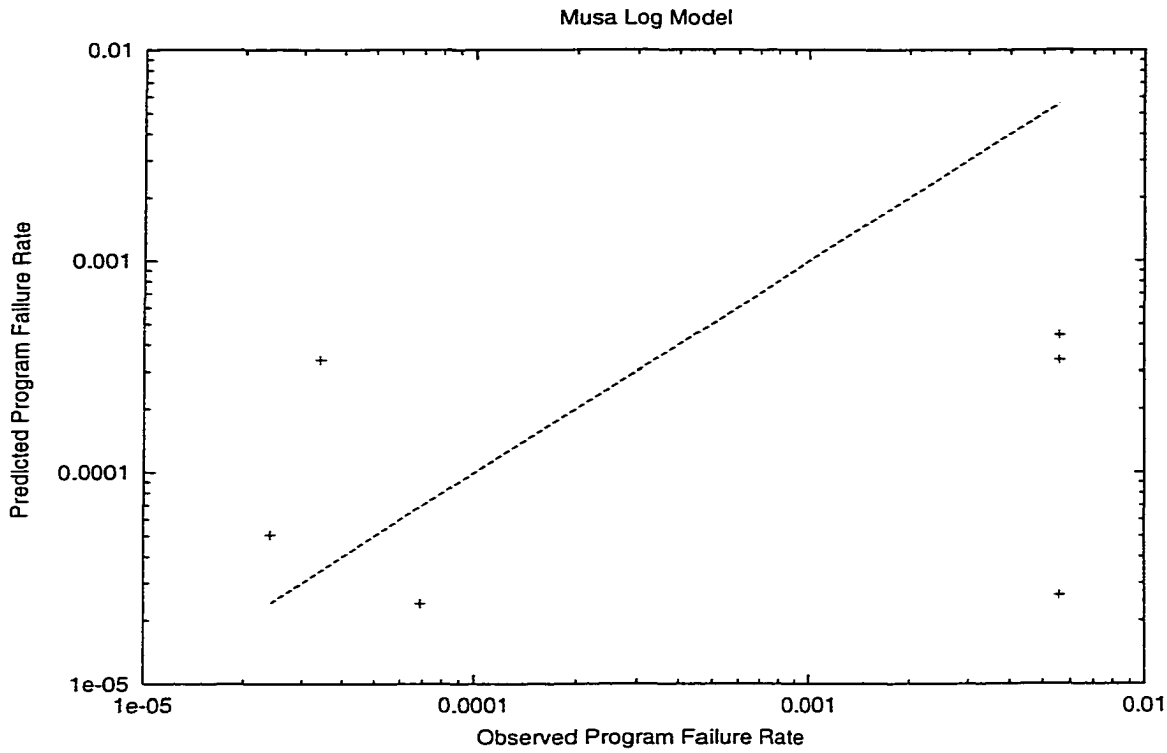

FIG. 106. OP Plot For Musa Log Model (Representative Data)

Model	JM	MB	ML	OS
Error	55.5226	54.2328	84.3532	54.0217

FIG. 107. Relative Error For The OP Plots Under Representative Testing

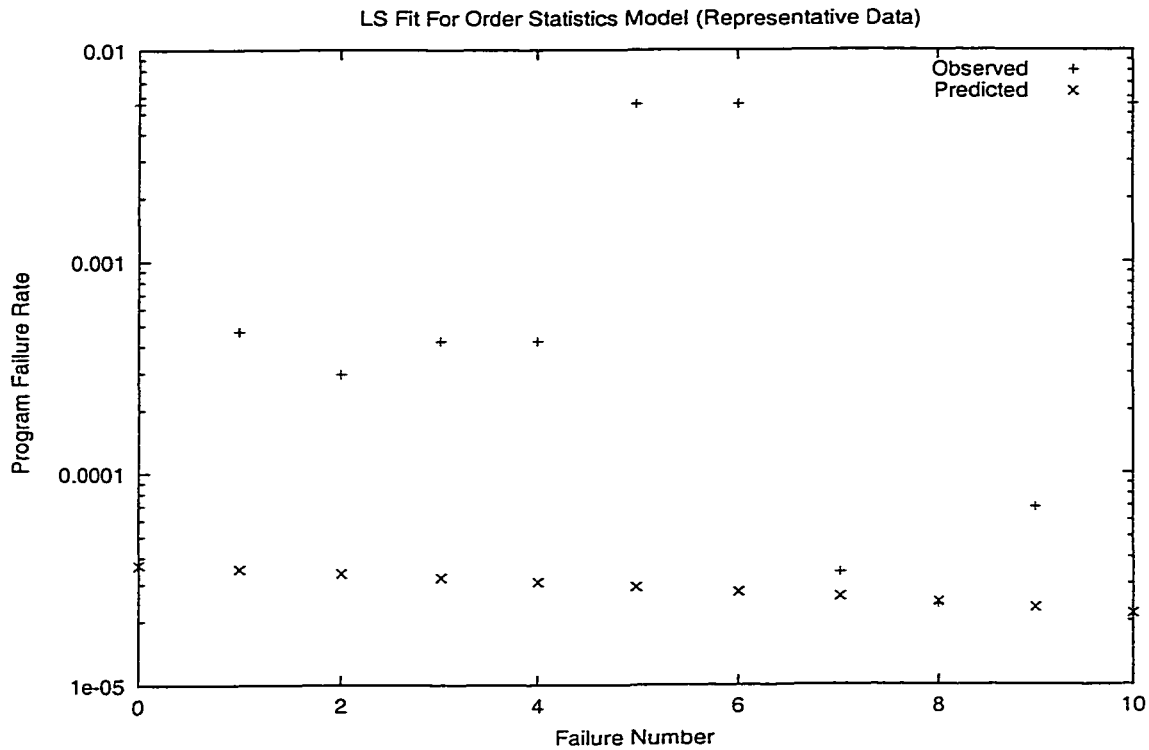

FIG. 108. Best Fit For Order Statistics Model (Representative Data)

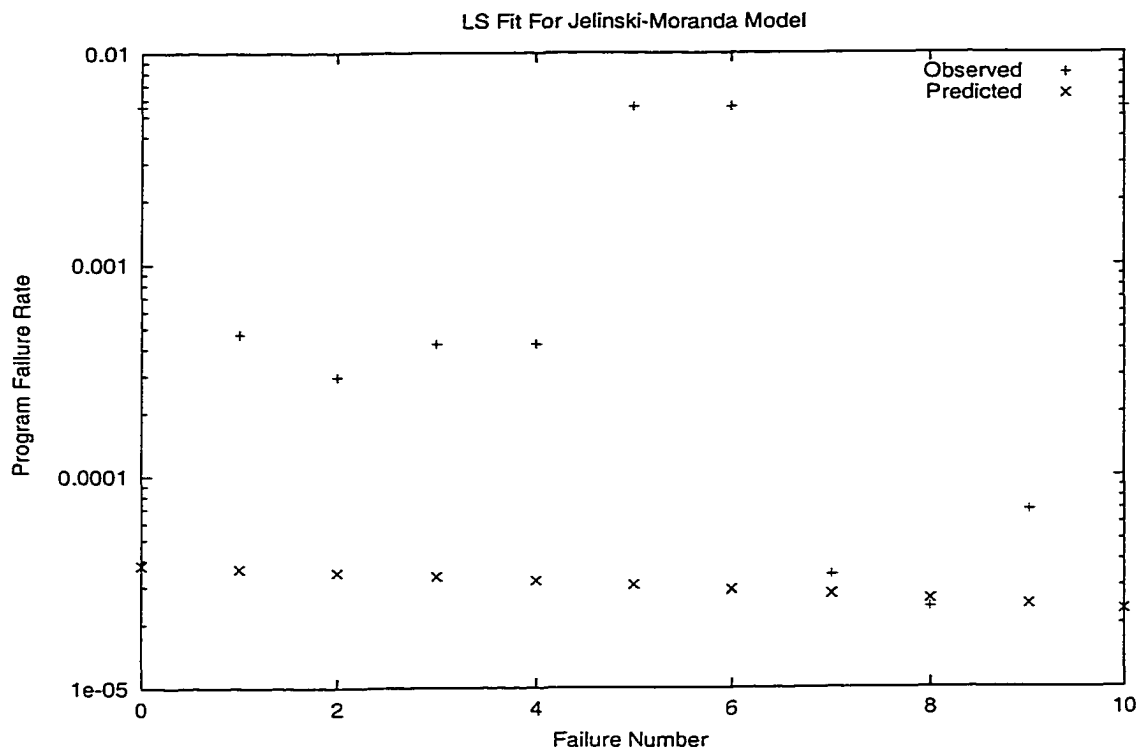

FIG. 109. Best Fit For Jelinski Moranda Model (Representative Data)

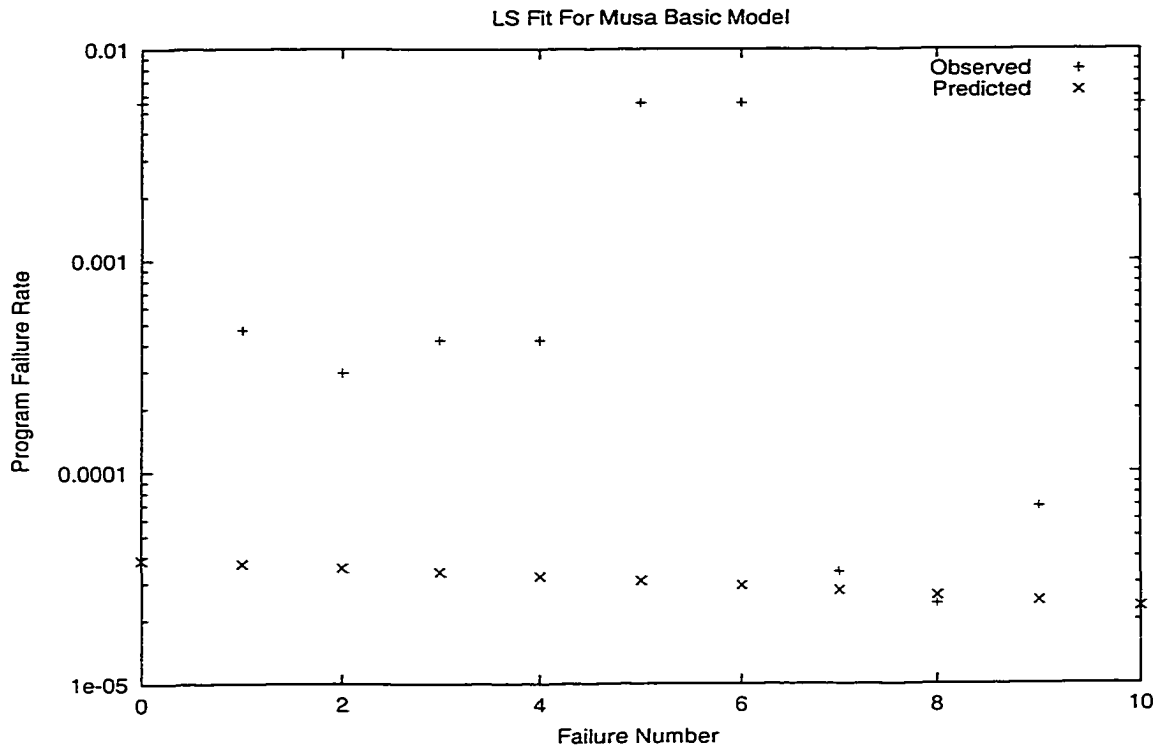

FIG. 110. Best Fit For Musa Basic Model (Representative Data)

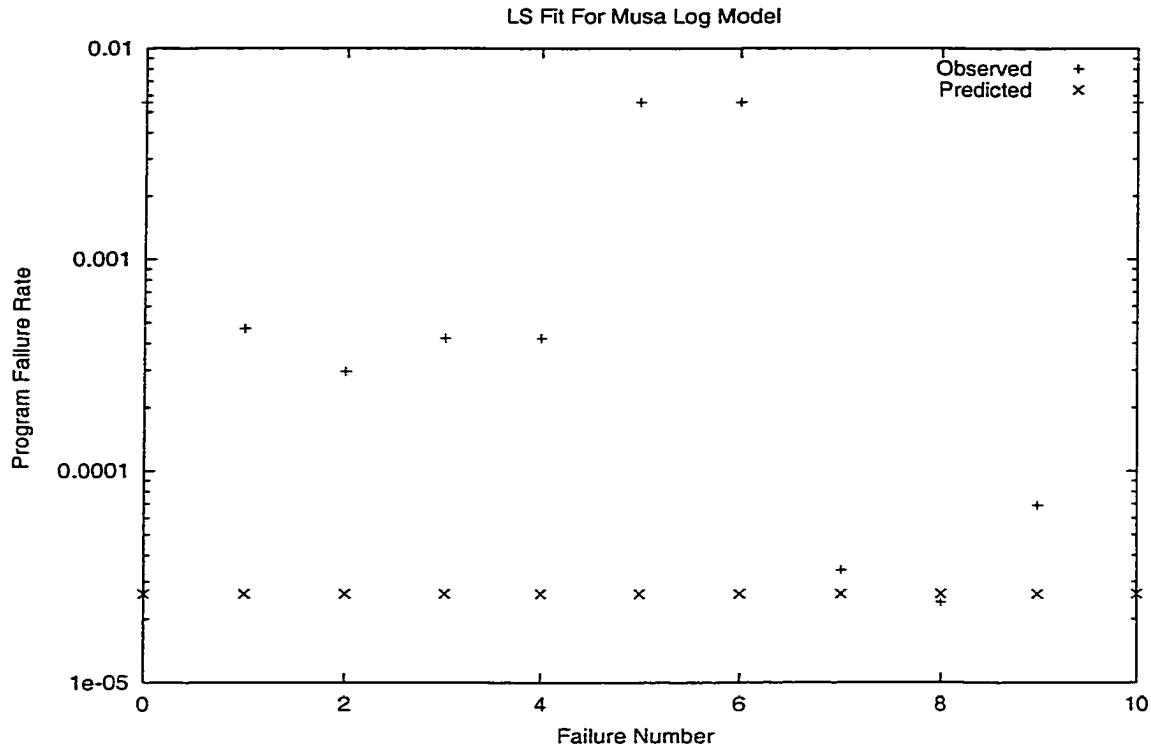

FIG. 111. Best Fit For Musa Log Model (Representative Data)

Model	JM	MB	ML	OS
Error	7.74072	7.73239	7.88717	7.80969

FIG. 112. Relative Error For The Best Fits Under Representative Testing

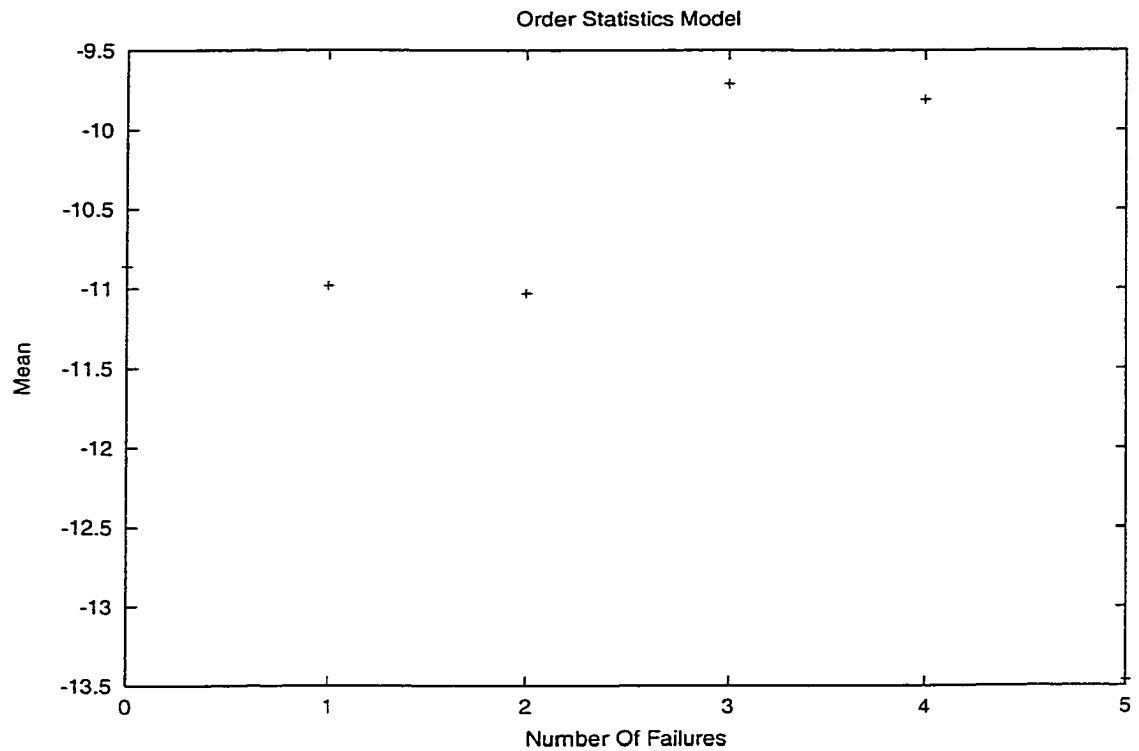

FIG. 113. OS Model Mean Progression With Representative Data

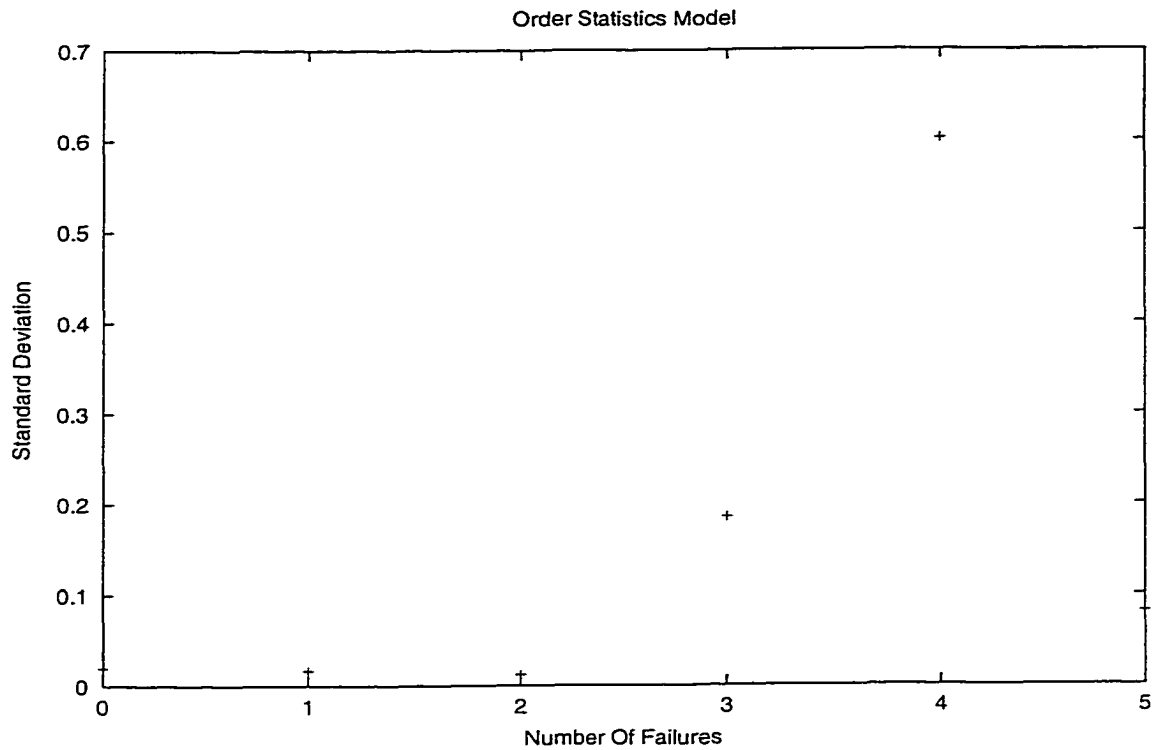

FIG. 114. OS Model Standard Deviation Progression With Representative Data

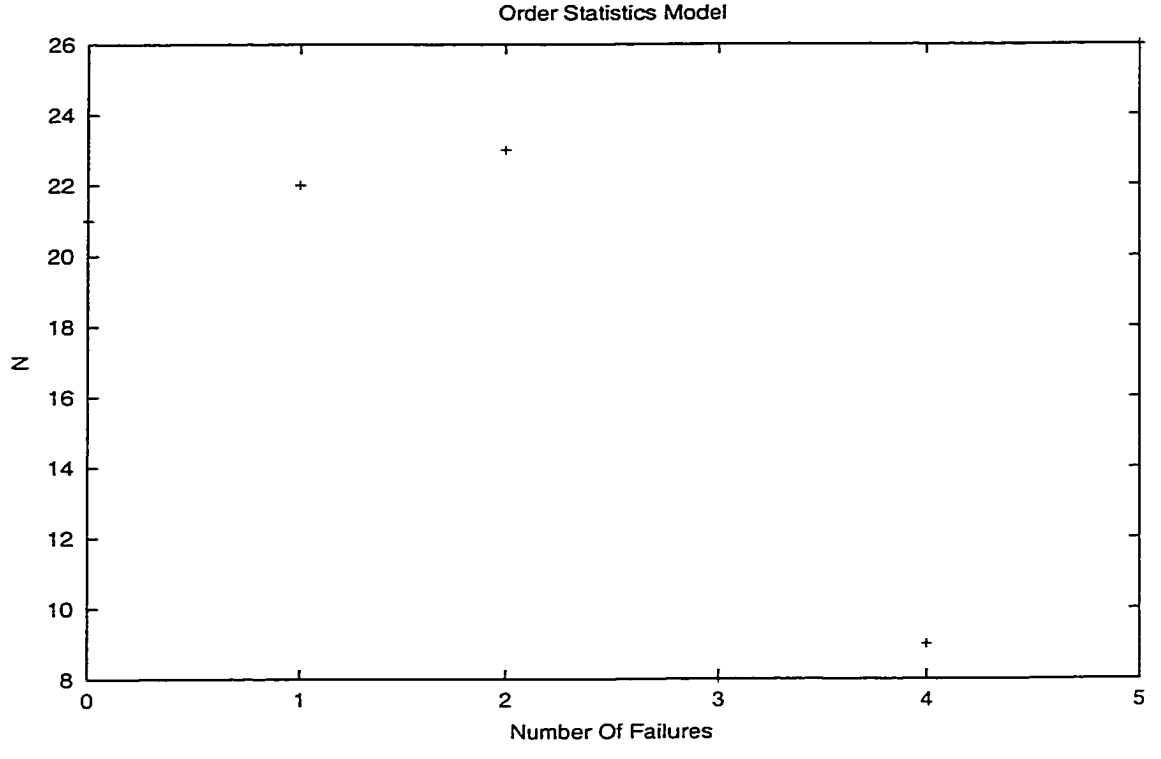

FIG. 115. OS Model N Progression With Representative Data

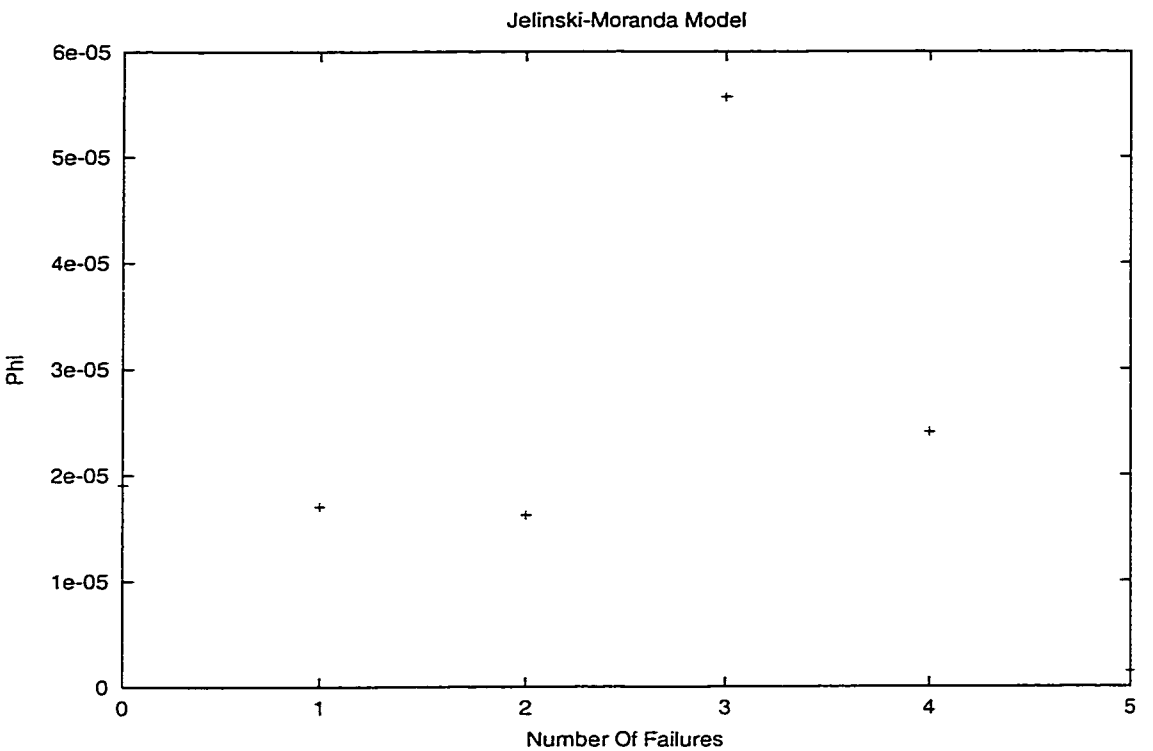

FIG. 116. Jelinski-Moranda Phi Progression With Representative Data

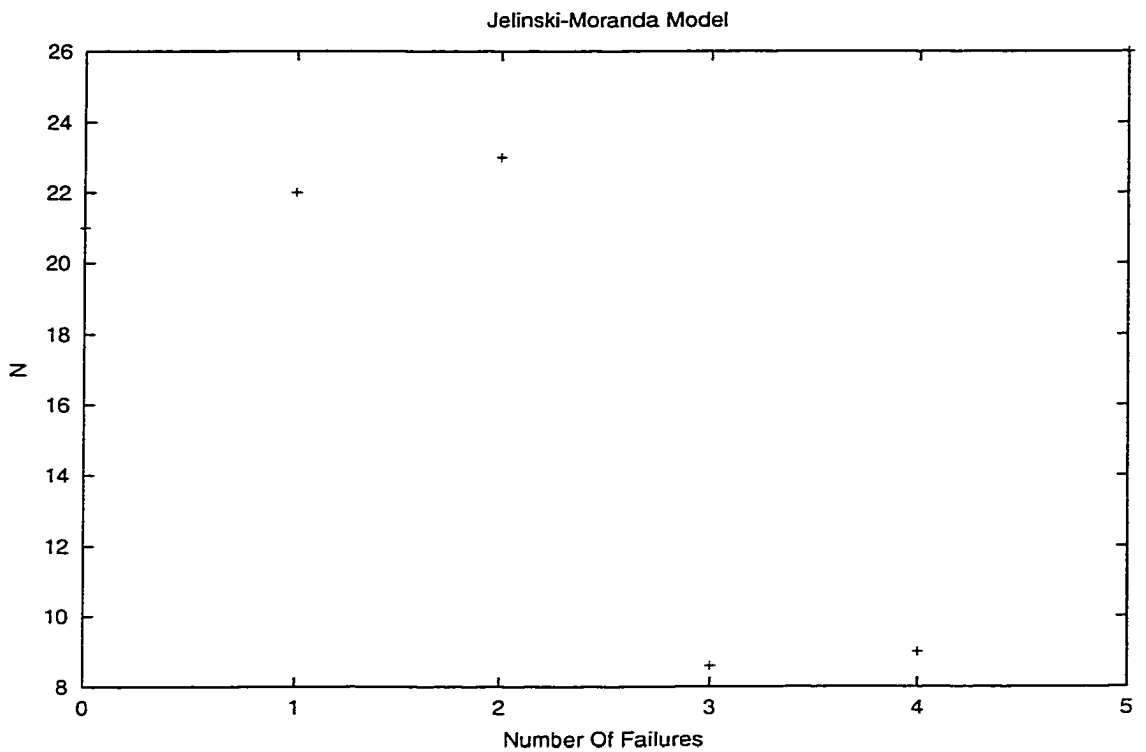

FIG. 117. Jelinski-Moranda N Progression With Representative Data

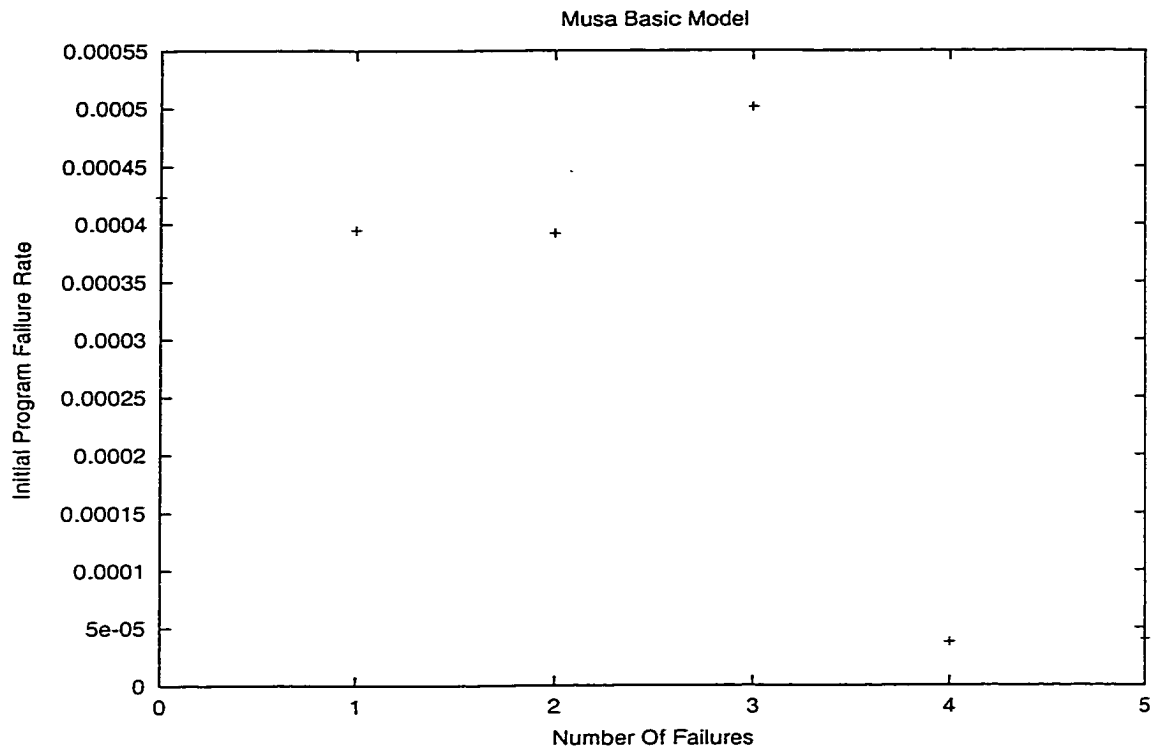

FIG. 118. Musa Basic Initial Failure Rate Progression With Representative Data

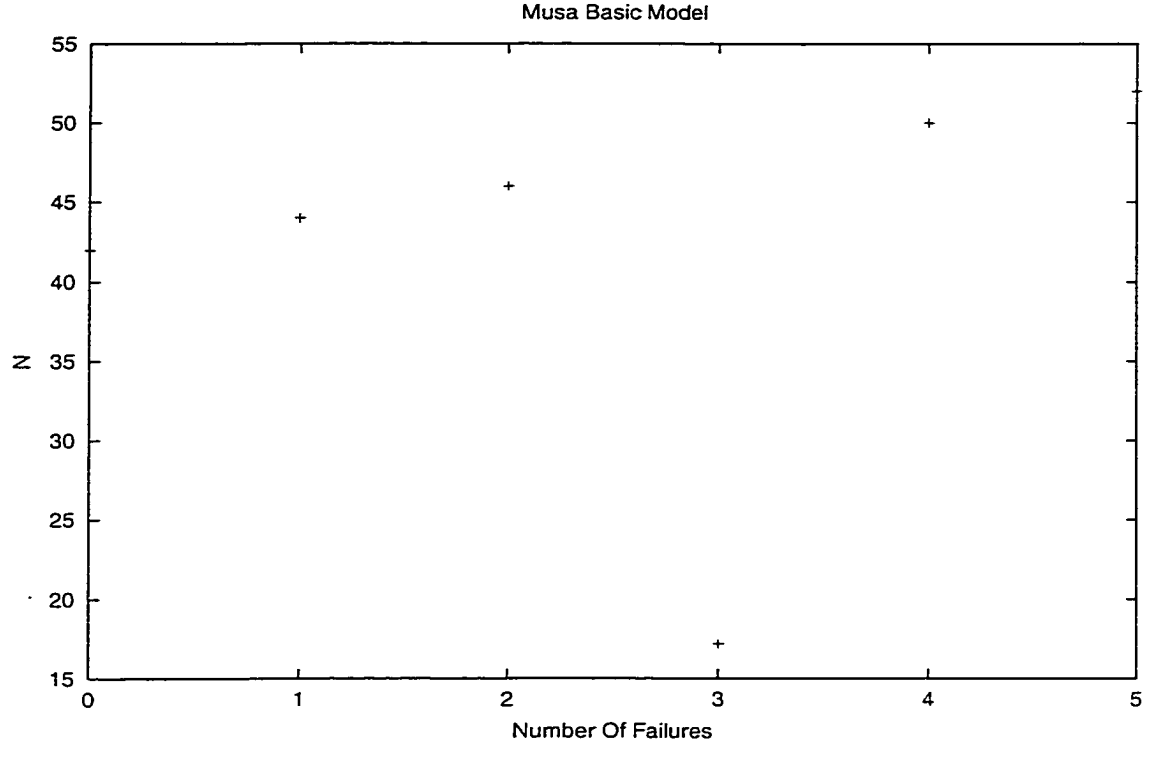

FIG. 119. Musa Basic N Progression With Representative Data

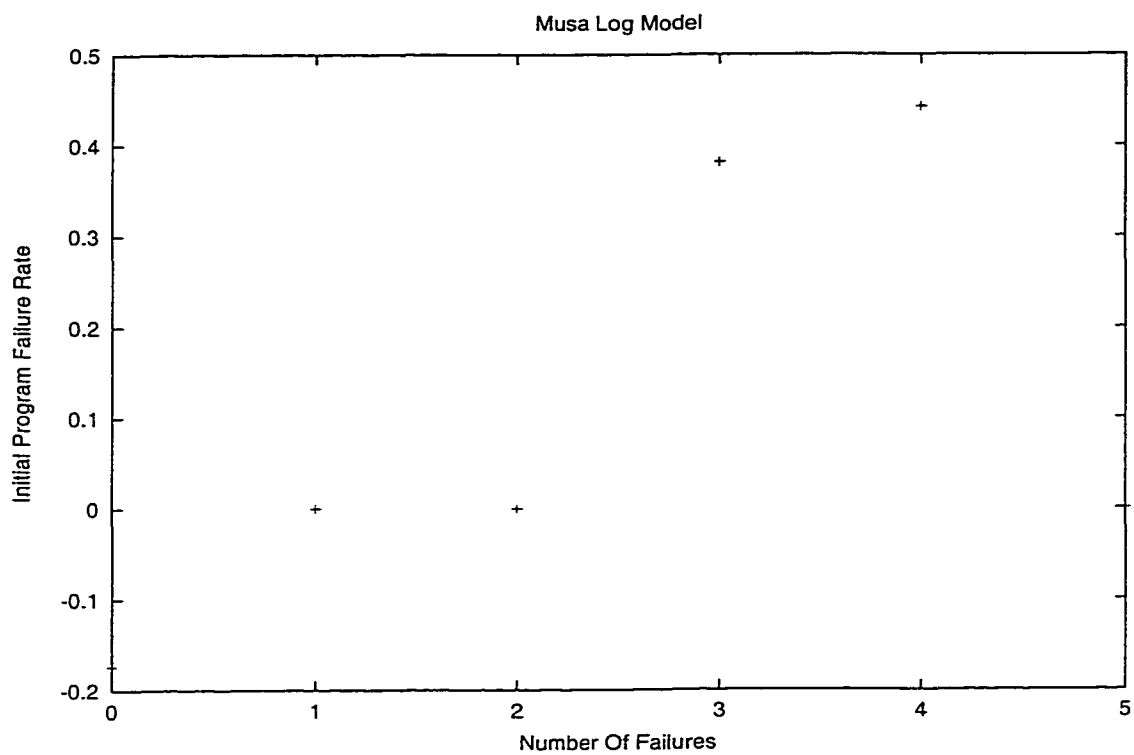

FIG. 120. Musa Log Initial Failure Rate Progression With Representative Data

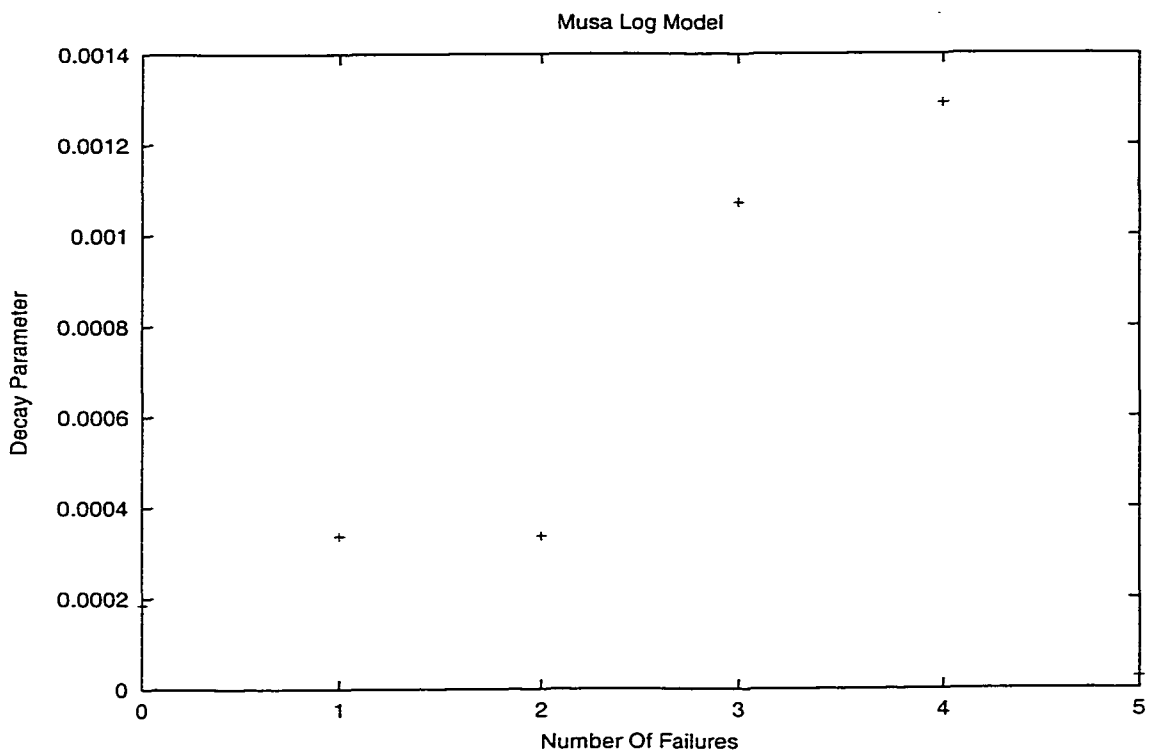

FIG. 121. Musa Log Decay Progression With Representative Data

12.5 Mixed Method Testing

The second part of this experiment involved testing the software using the mixed method approach to testing described in earlier chapters. During the representative testing phase of the mixed method testing process, the input driver fed representative data to the data historian for use by the test software. The test version of the software was initially compiled to contain all of the known faults. As testing was conducted, the oracle compared output of the gold version to the output of the test version. Whenever a discrepancy was found, the gold and test versions were stopped and the fault responsible for causing the failure was found and removed from the test version. The gold and test versions of the software were then restarted. This process was repeated until one full day of testing passed without finding any new faults. At this time, the testing process switched from representative testing to directed testing. To reach this point in the testing process required a total of 85,800 test cases, with the last 43,200 of these test cases being run to satisfy the crossover criteria.

After switching to directed testing, the input driver was fed uniform random data for each tag value. This data then sent to the data historian and finally to the alarm server software. The directed testing criteria that was used for this experiment was statement coverage. For this experiment, instrumentation was added to the software being tested to keep track of which lines of code had been executed during the testing process. This information was logged to output files for analysis to aid in the creation of the test cases required to meet the coverage requirements.

A total of 5400 test cases were run (simulating three hours of system execution) using uniform random data. During this interval, about 74 percent coverage was obtained. The remainder of the directed testing process involved directly targeting specific unexecuted lines of code and creating test cases to execute as many of those lines of code as possible. A total coverage of about 95 percent was eventually obtained. None of the unexecuted code contained any of the faults being tested for. The uncovered code consisted of error handling code that would

Fault	Interfailure Time	Fault Failure Rate
1	9270	
9	10260	
4	3810	
2	180	
12	2100	
6	1170	
10	5700	
13	3750	
3		.000224014
7		.02916
8		.000067204

FIG. 122. Faults Found During Mixed Testing

be executed in case of corrupted databases, indexes, or network failures.

During the course of this experiment, a total of about 91,000 cases were generated, executed, and tested for correctness, with about 94 percent of these test cases occuring during the representative phase of testing.

The list of faults that were found during mixed testing are shown in Figure 122. The first column of the table shows the fault number, as given in Figure 101. The second column of the table shows the interfailure time (in terms of number of test cases) for the fault if it was found during representative testing. The third column of the table shows the fault failure rate for the fault if it was found during directed testing. The faults in this table are listed in the same order in which they were found during this experiment. The set of faults found under mixed testing and the set of faults found under representative testing are very similar. In fact, they only differ by one element. This observation lends support to the Ordered Directed Testing Property, which states that directed and representative testing will uncover largely the same set of faults as testing approaches coverage.

12.5.1 Mixed Testing Results

Since the failure data under mixed testing was gathered separately from the data gathered under representative testing, the observed failure times were very different. These differences would make it difficult to make meaningful comparisions between the Order Statistics Model under mixed testing and the other models under representative testing. For this reason, hybrid versions of the Jelinski-Moranda Model, the Musa Basic Model, and the Musa Log Model were developed during this research.

Like the Order Statistics Model, these hybrid models will accept failure data in the form of either program failure rates or fault failure rates. The hybrid models were developed by modifying the existing models to make fault failure rate predictions based on the existing model parameters. The calculation made by each model to predict fault failure rates is derived from that model's program failure rate calculation.

When estimating fault failure rates, the Jelinski Moranda Hybrid Model uses the following calculation:

$$ffr_i = \phi \tag{23}$$

where ϕ has the same meaning as in the original model. This equation for the fault failure rate follows immediately from the Jelinksi-Moranda equation for the program failure rate which is:

$$\lambda_i = (N - i)\phi. \tag{24}$$

When estimating fault failure rates, the Musa Basic Hybrid Model uses the following calculation:

$$ffr_i = \frac{\lambda_0}{v_0} \tag{25}$$

where λ_0 and v_0 have the same meaning as in the original model. This equation for the fault failure rate was originally derived by Musa. [12]

When estimating fault failure rates, the Musa Log Hybrid Model uses the following calculation:

$$ffr_i = \lambda_0 \theta \exp(-\theta i) \tag{26}$$

where λ_0 and θ have the same meaning as in the original model. This equation for the fault failure rate was also originally derived by Musa. [12]

Once these hybrid models were developed, they were applied to the failure data from mixed testing and the results were analyzed. Once again, the analysis was carried out by generating OP-Plots for each model for the data, comparing the best fits for each model, and comparing the stability of the models using parameter progression plots.

Comparing The Predictive Accuracy Of Each Model

The OP Plots for each model are shown in Figures 123 to 126. To obtain the OP Plots, the failure data was input into the models in incremental steps. For example, for the first pass of OP Plot generation, only five failure rates were used as input, and the models predicted the sixth failure rate. This prediction of the sixth failure rate was paired with the actual observed sixth failure rate and the result was plotted as a point on the OP Plot. This process was repeated, with an additional input point being added during each step, until all points were considered.

When looking at the OP Plots for the models, it appears that all of the models performed similarly under mixed testing. These observations are supported by the data in Figure 127, which shows the relative error for the OP Plots for each model.

When comparing the results under mixed testing to the results under representative testing, one item of note is that the relative errors under mixed testing are less than the relative errors under representative testing. However, a large part of the discrepancy between theses results can be explained by the nature of the data used as input to the models. The data in Figure 102 appears to be much more erratic and noisy than the data in Figure 122. For example, consider the starting sequence of program failure rates in Figure 102. After seeing program failure rates of 180, 2130, 3390, 2370, 2370, it is not likely that any model will predict a subsequent failure rate of 180, which is the next oberved value. Likewise, after seeing 180, 2130, 3390, 2370, 2370, 180, 180, it is not likely that any model will predict that the next value will be anywhere close to 29370, which is the next

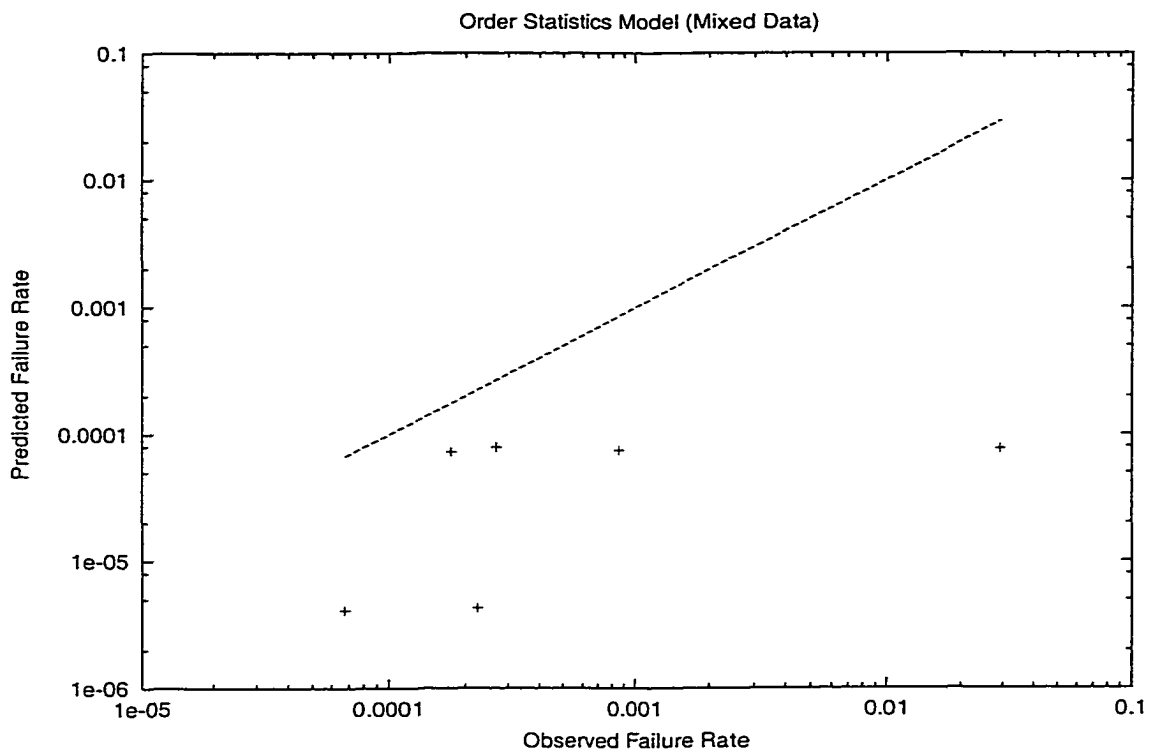

FIG. 123. OP Plot For Order Statistics Model (Mixed Data)

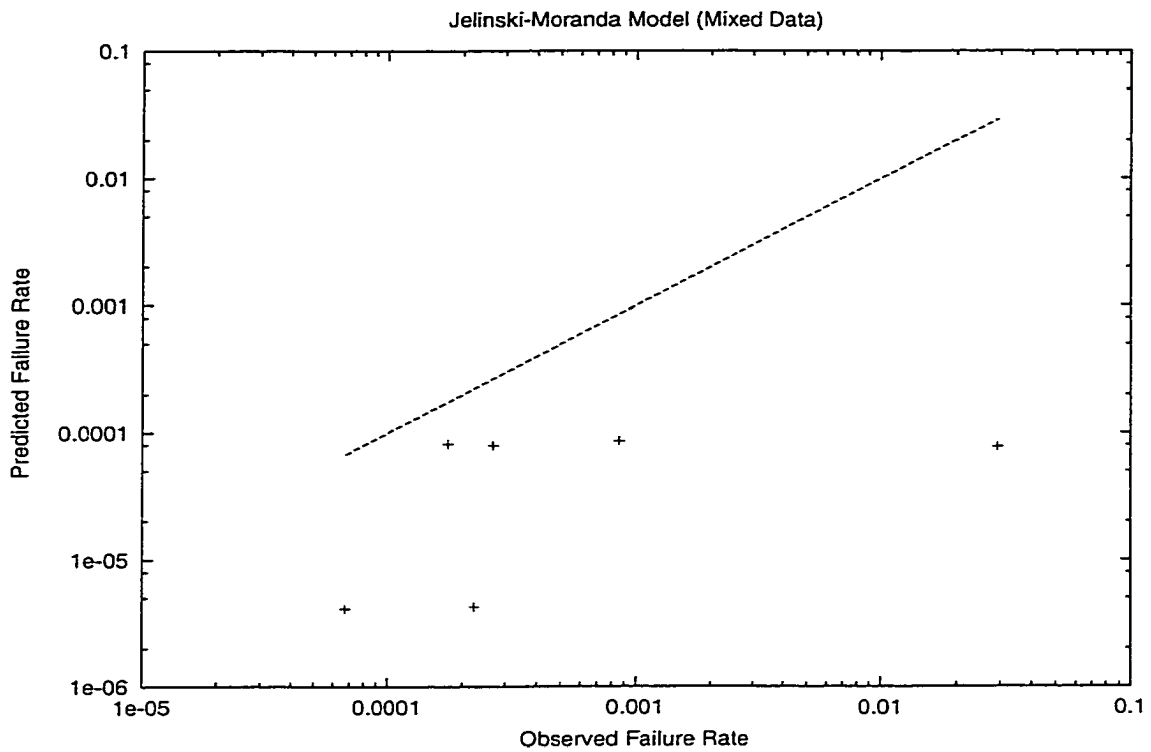

FIG. 124. OP Plot For Jelenski-Moranda Model(Mixed Data)

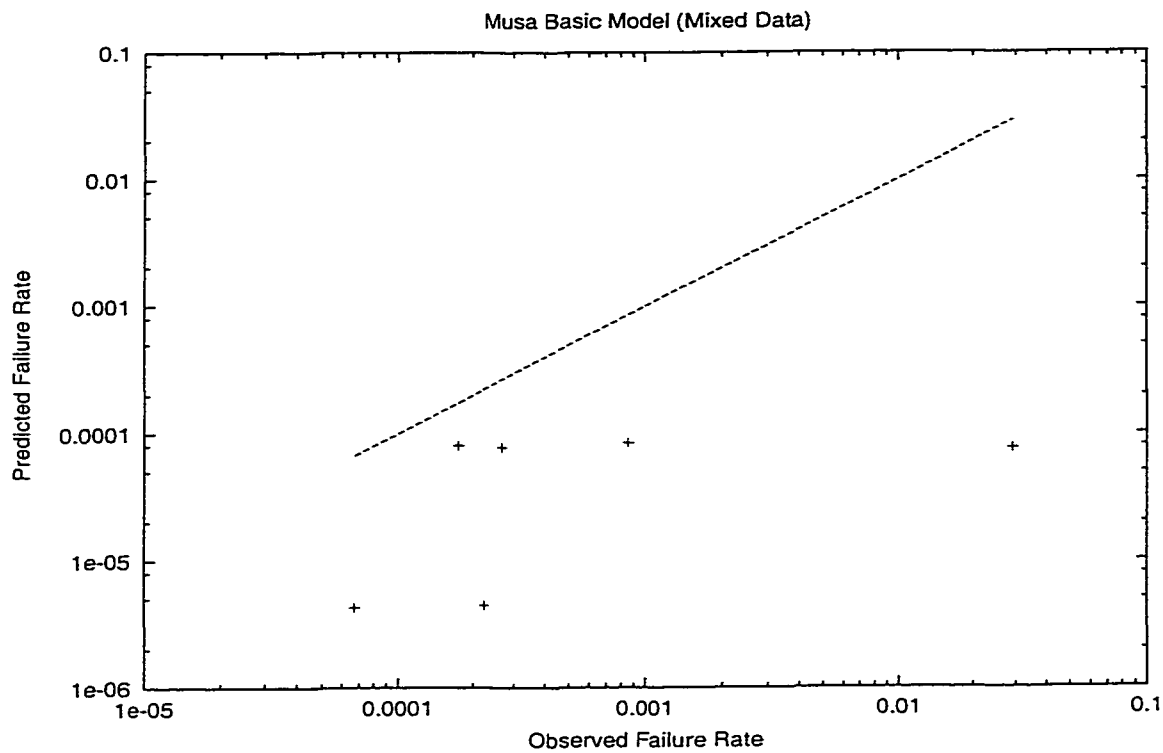

FIG. 125. OP Plot For Musa Basic Model (Mixed Data)

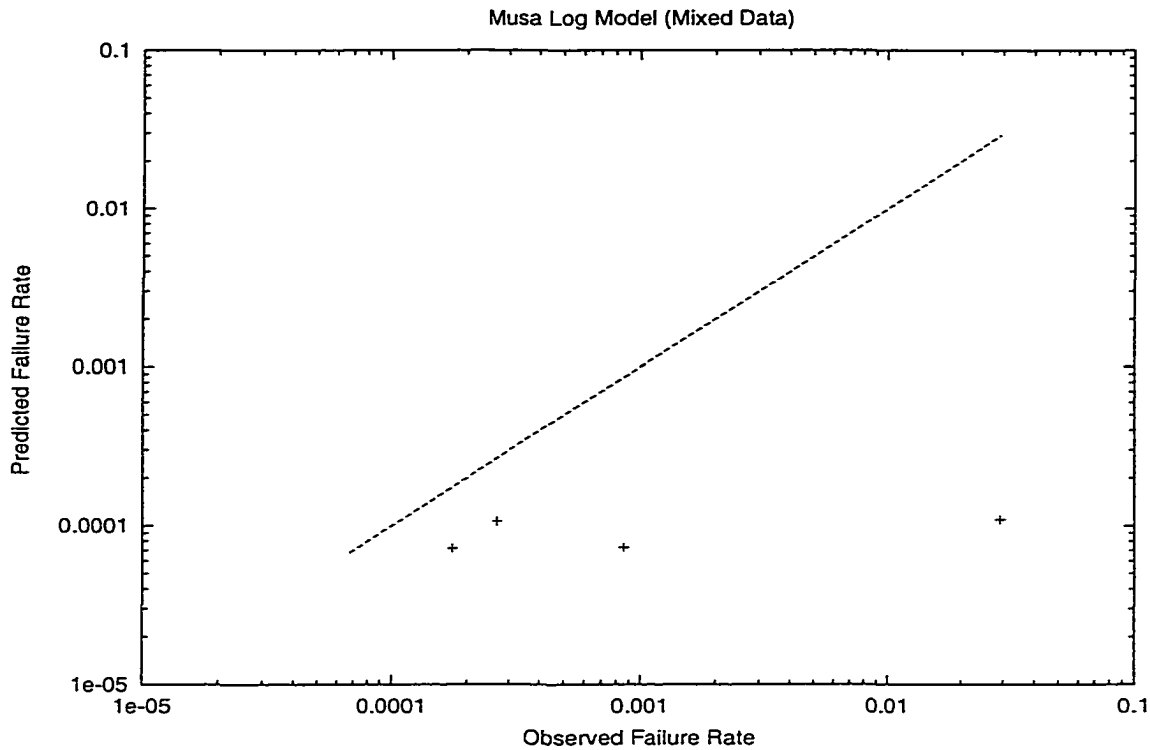

FIG. 126. OP Plot For Musa Log Model (Mixed Data)

Model	JM	MB	ML	OS
Error	4.42641	4.43517	4.54007	4.50907

FIG. 127. Relative Error For The OP Plots Under Mixed Testing

observed value. The data in Figure 102 does not exhibit these large swings in order of magnitude for the observed values, resulting in better predictions by the model.

Therefore, since it seems like model performance is dependent largely on the nature of the input data, it follows that the failure data acquisition process and the type of failure data used has as much or more impact on the quality of the predictions made as any other factor.

Comparing The Best Fits For Each Model

The best fits for each model under mixed testing are shown in Figures 128 to 131. Once again, we see that all of the models performed similarly. The relative error for the best fits for is model is shown in Figure 132. From the graphs, it appears that all of the models were overly optimistic about the predicted program failure rate. This optimism can possibly be explained by the fact that the first two observed interfailure times are the largest of all observed values, which would lead the models to believe that the software is more reliable than it really is.

Comparing The Parameter Progressions For Each Model

The parameter progressions for each model are shown in Figures 133 to 141. These graphs show how the estimated values for each model parameter changed from one step of the OP Plot generation to the next. For all of the models under mixed testing, the model parameters are fairly stable as long as the data is fairly stable.

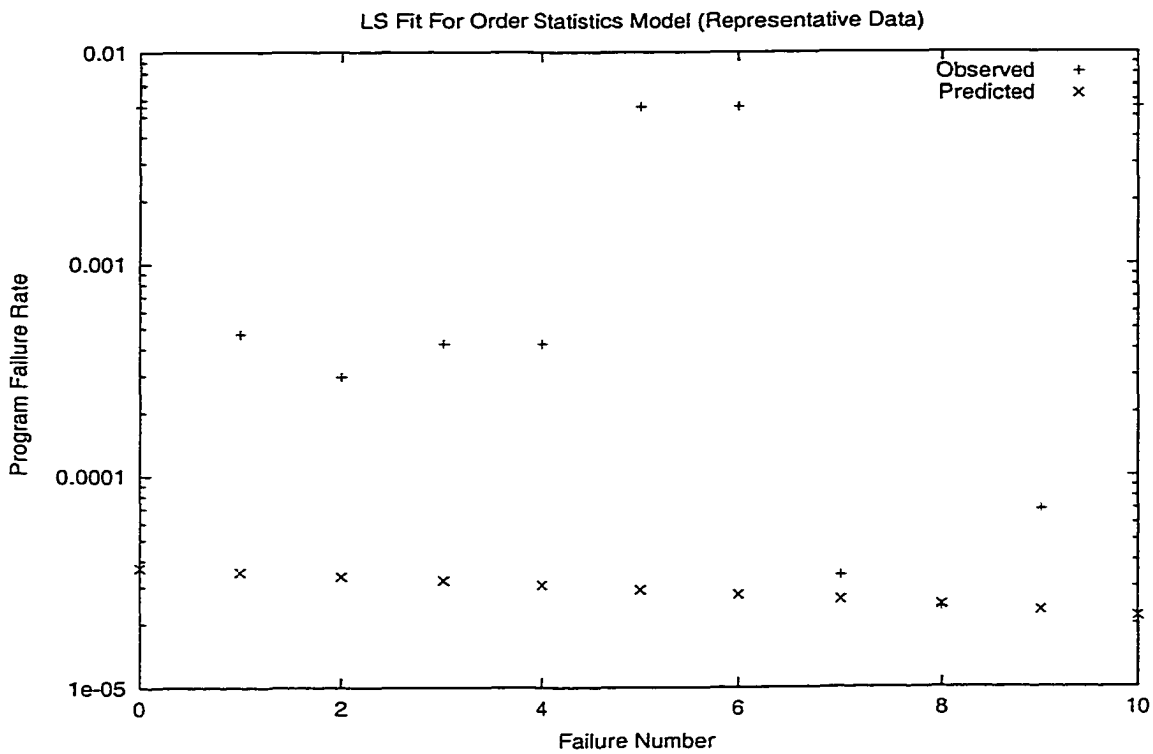

FIG. 128. Best Fit For Order Statistics Model (Mixed Data)

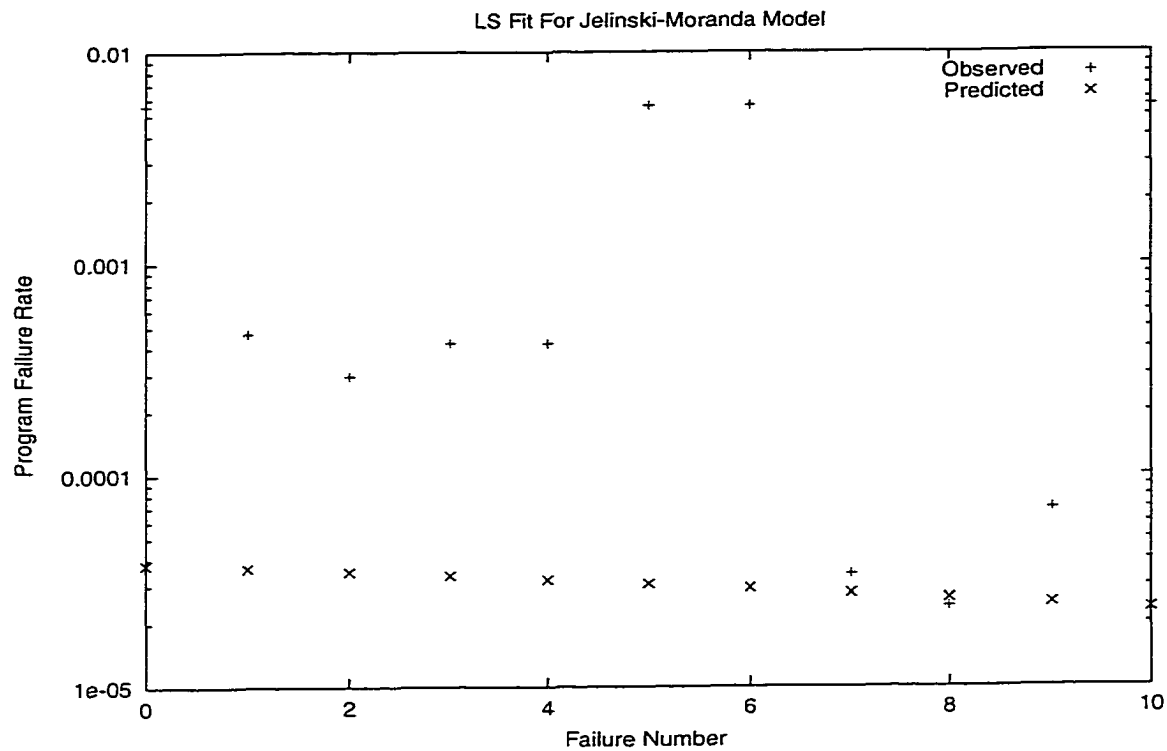

FIG. 129. Best Fit For Jelinski Moranda Model (Mixed Data)

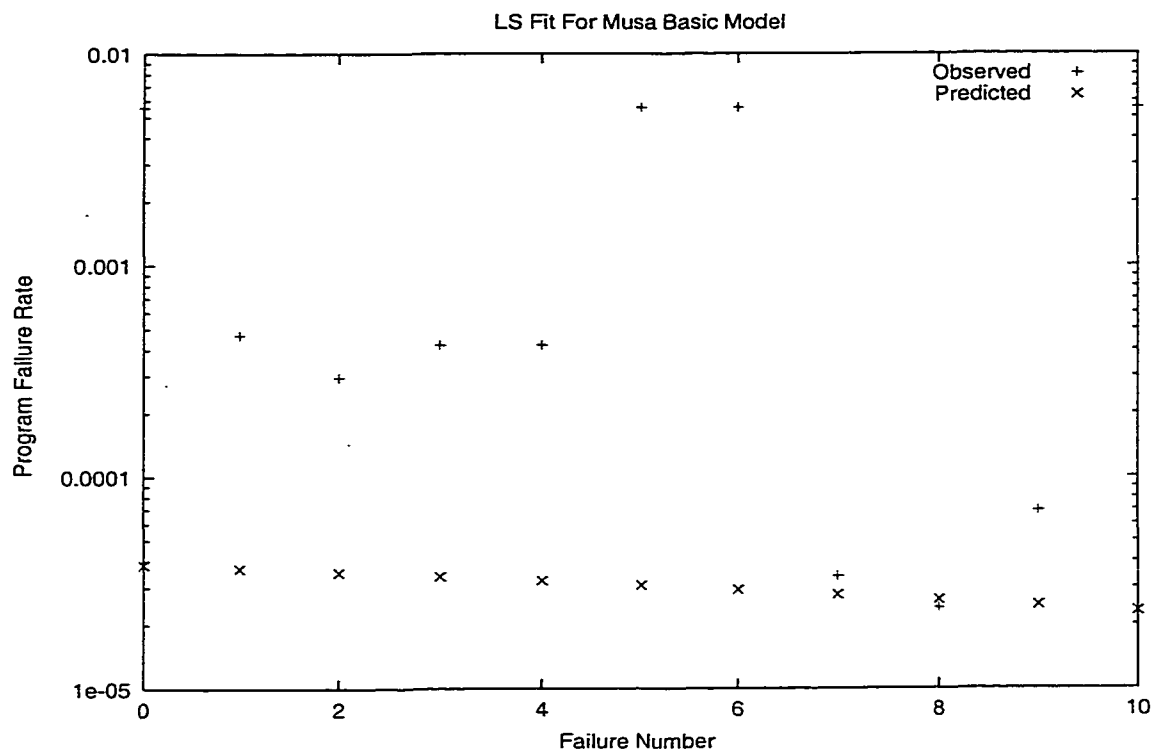

FIG. 130. Best Fit For Musa Basic Model (Mixed Data)

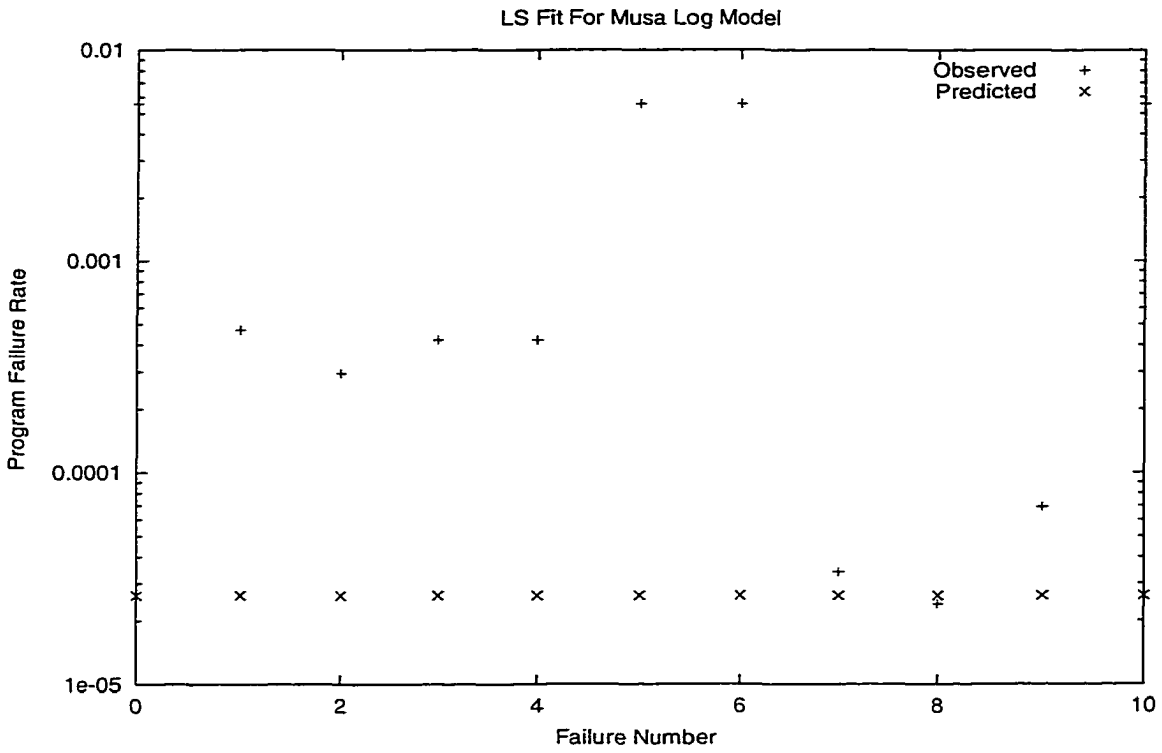

FIG. 131. Best Fit For Musa Log Model (Mixed Data)

Model	JM	MB	ML	OS
Error	6.44925	6.45965	6.20296	6.45437

FIG. 132. Relative Error For The Best Fits Under Mixed Testing

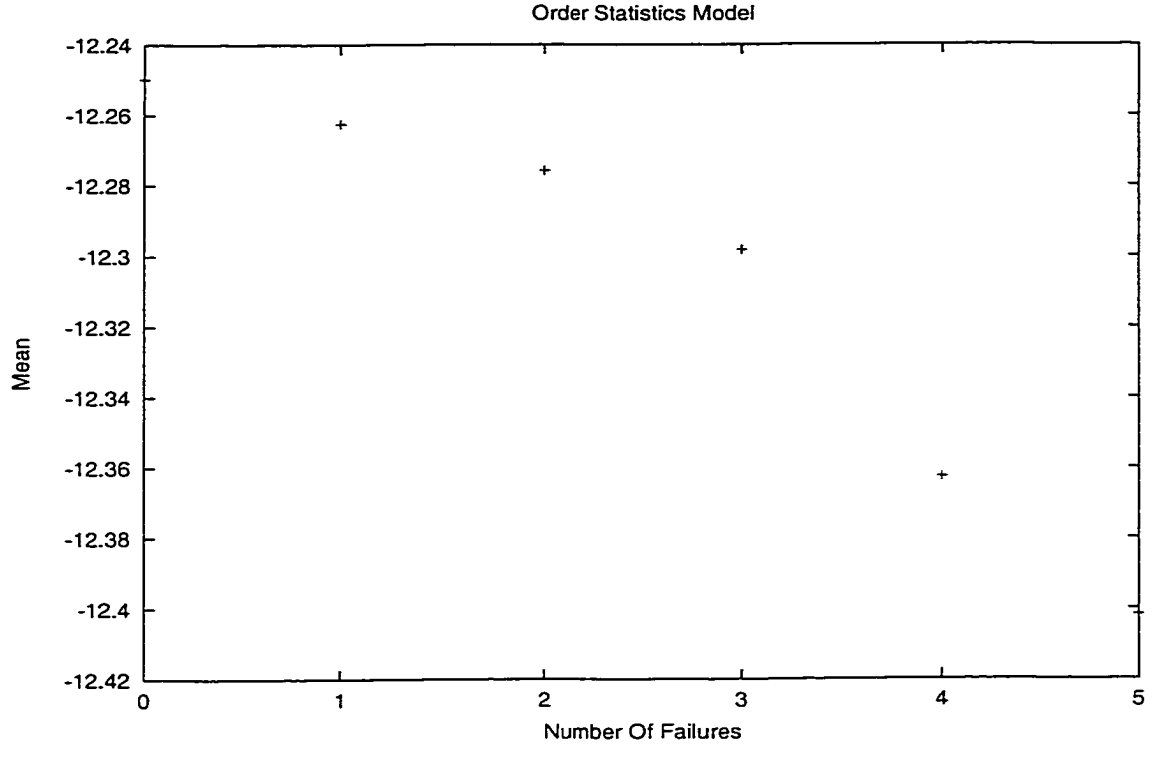

FIG. 133. OS Model Mean Progression With Mixed Data

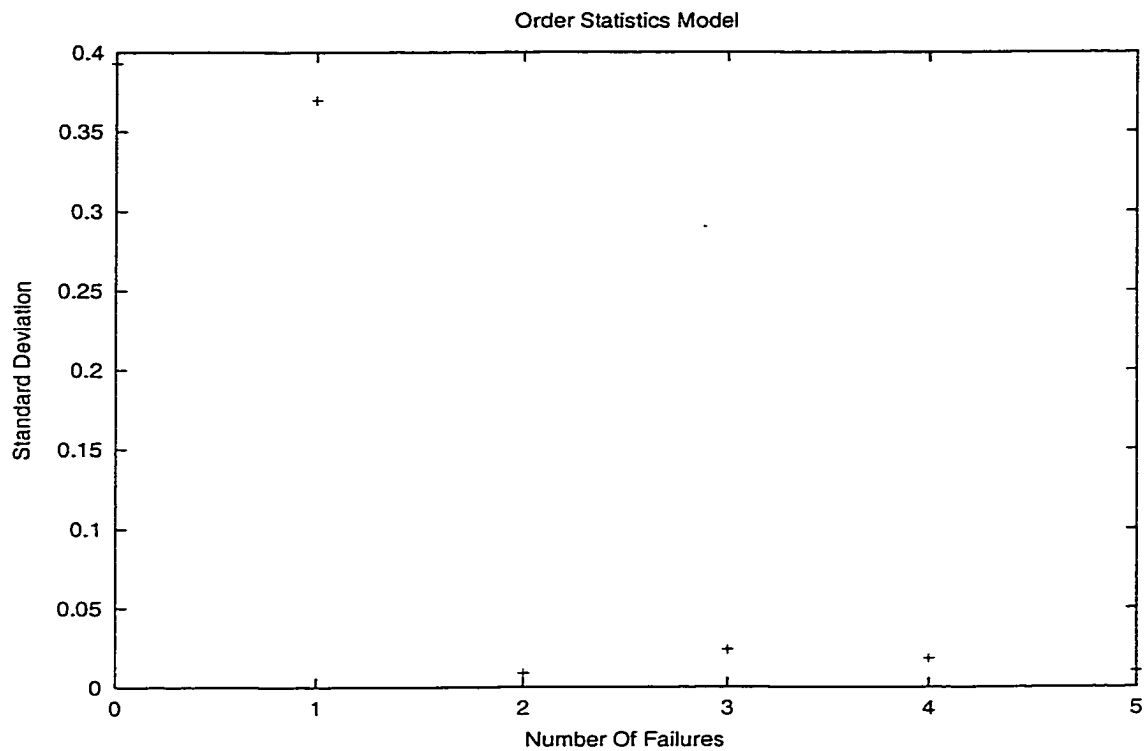

FIG. 134. OS Model Standard Deviation Progression With Mixed Data

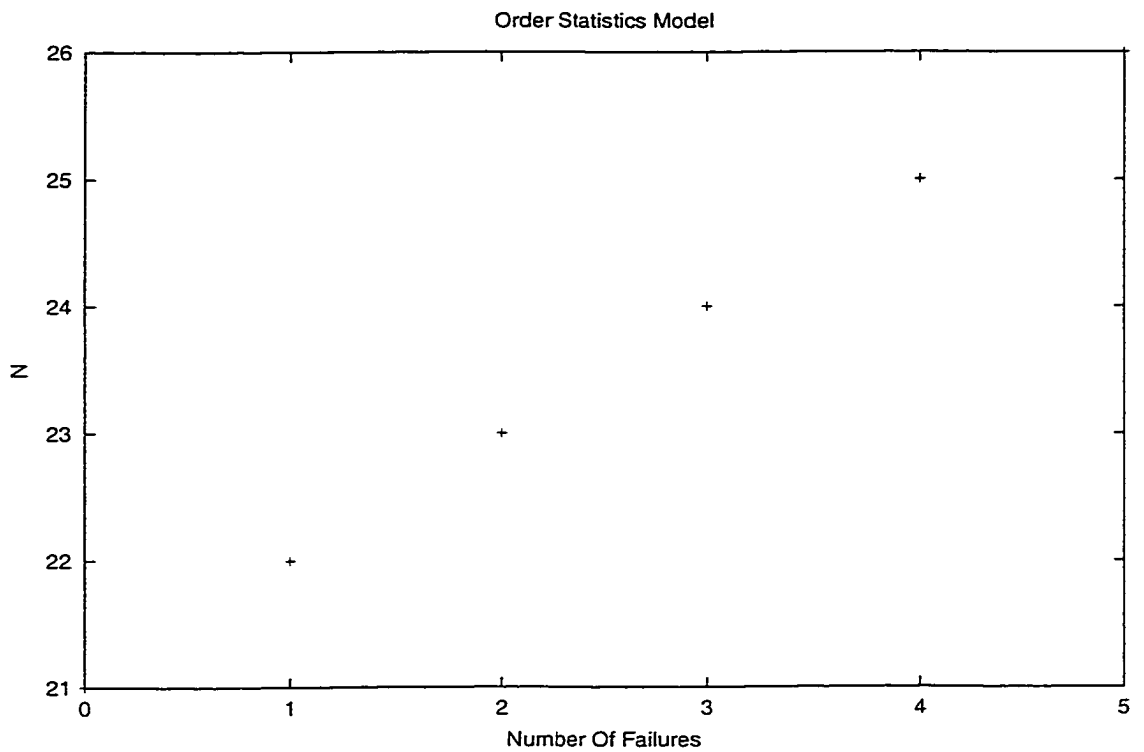

FIG. 135. OS Model N Progression With Mixed Data

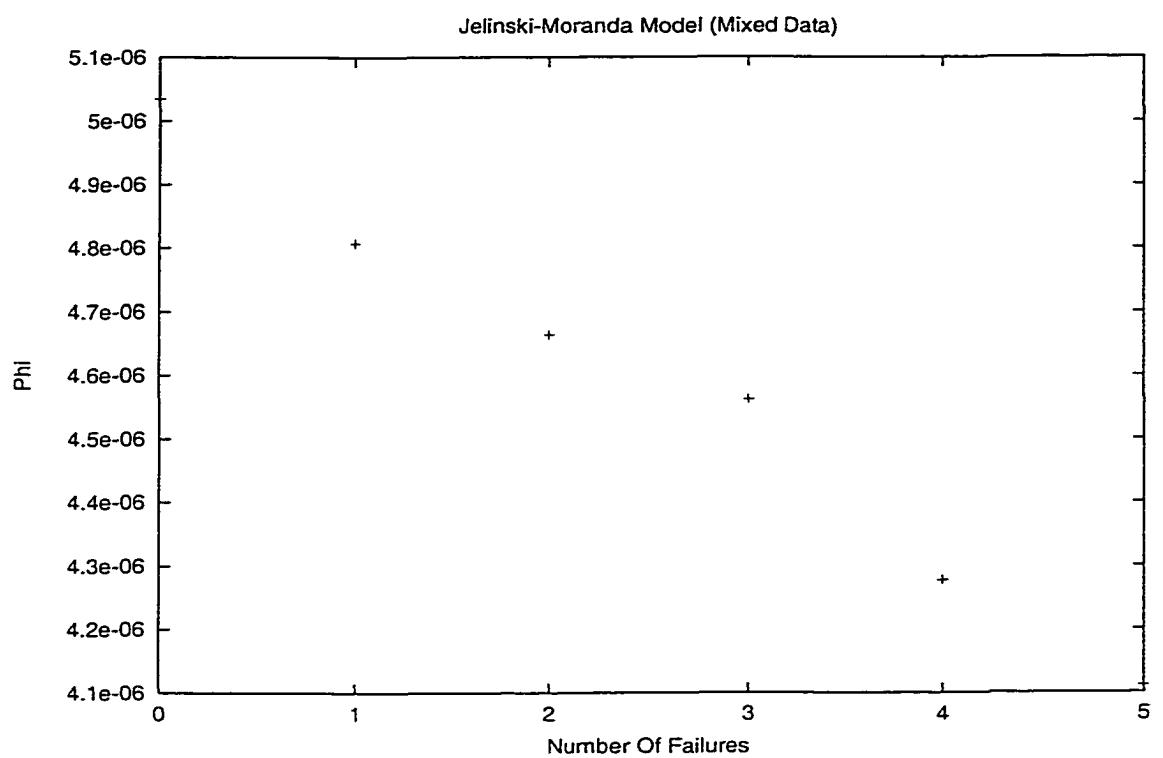

FIG. 136. Jelinski-Moranda Model Phi Progression With Mixed Data

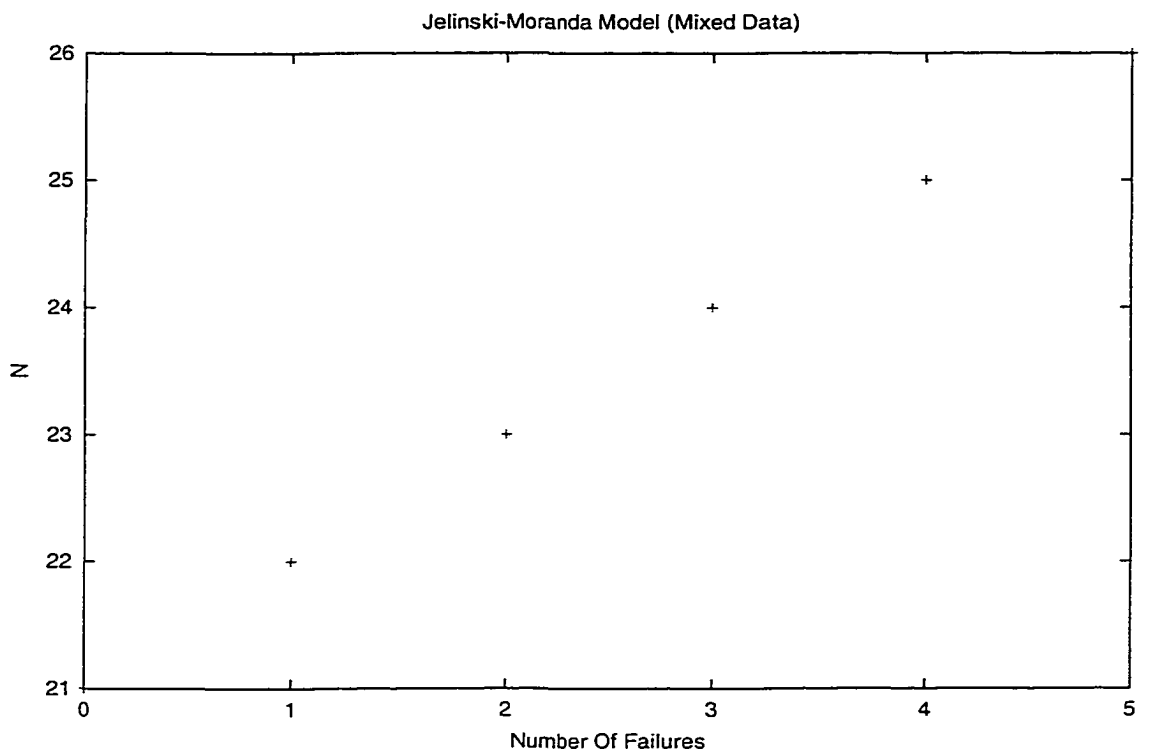

FIG. 137. Jelinski-Moranda Model N Progression With Mixed Data

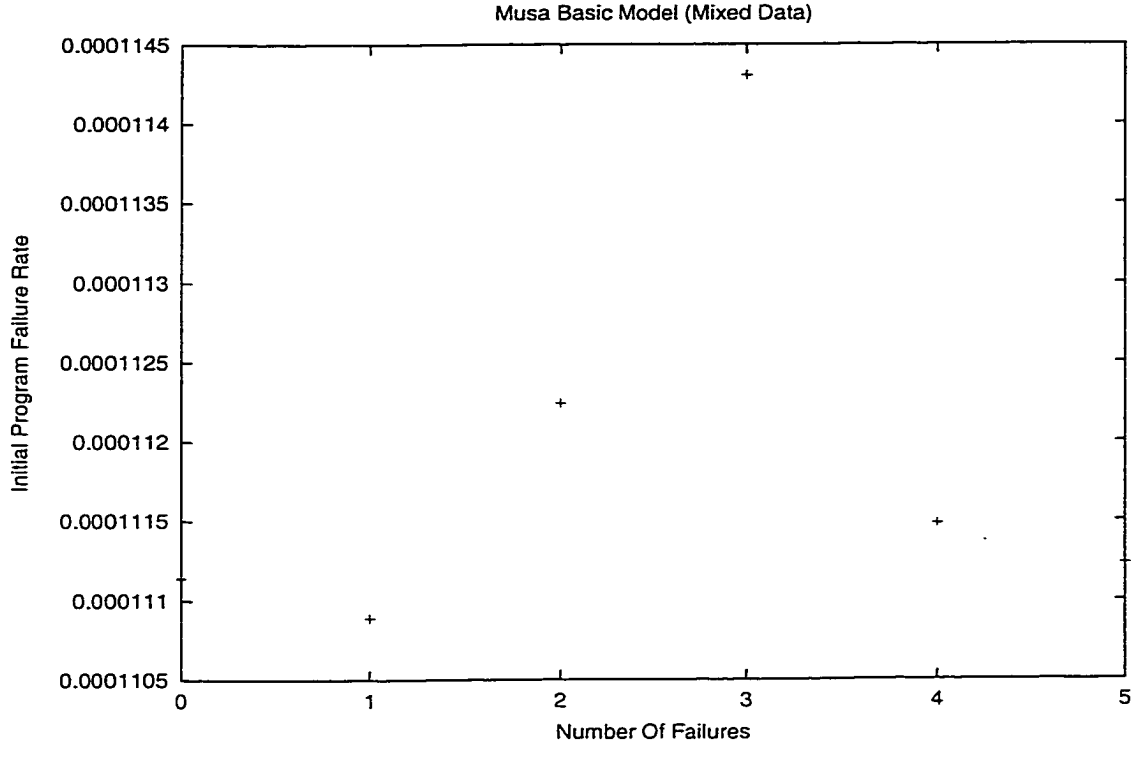

FIG. 138. Musa Basic Model Initial Failure Rate Progression With Mixed Data

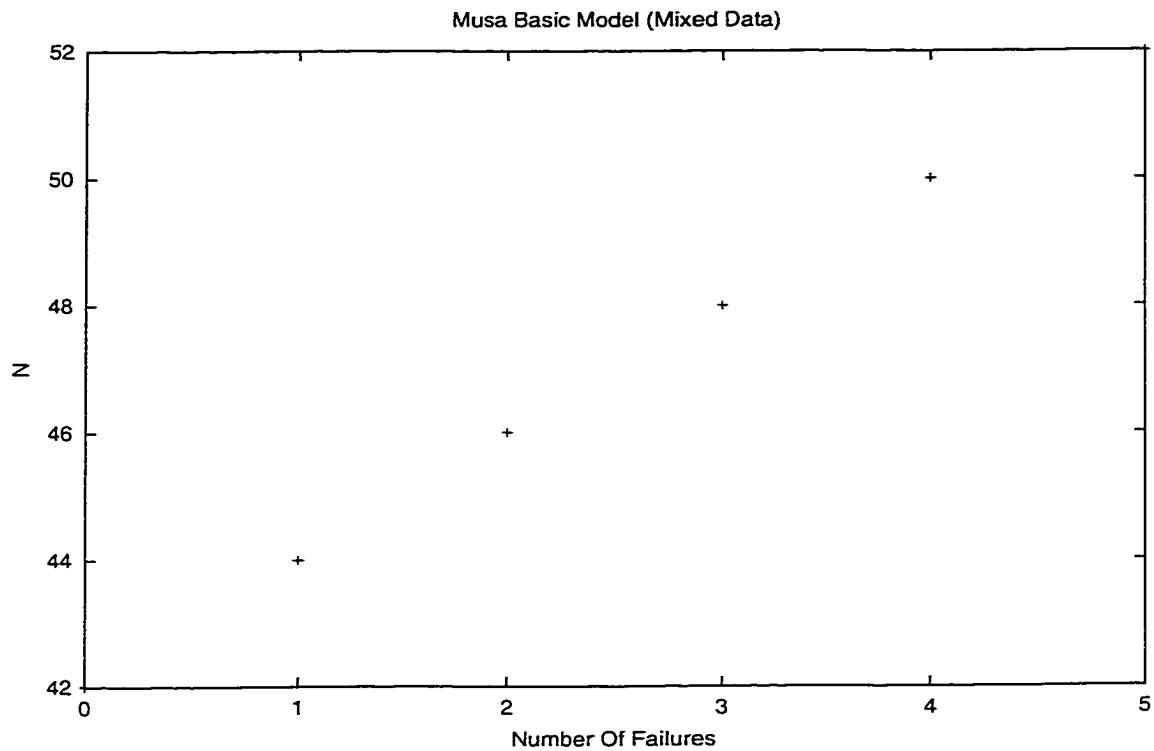

FIG. 139. Musa Basic Model N Progression With Mixed Data

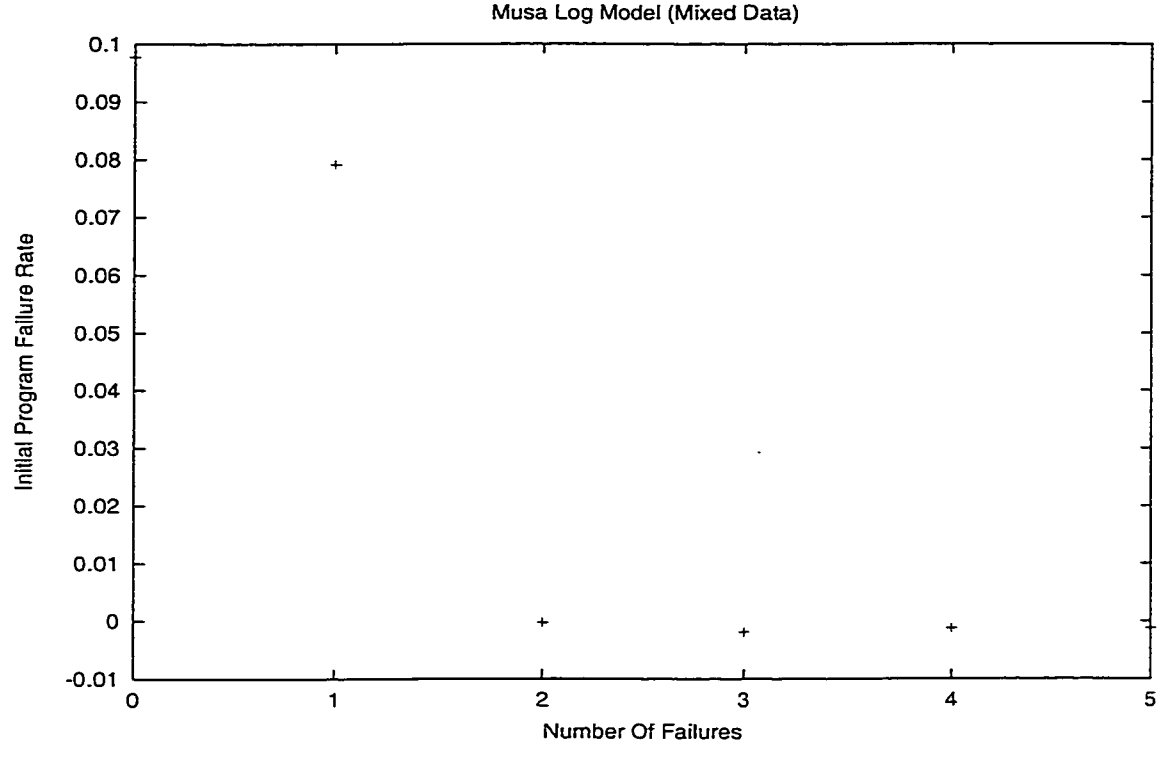

FIG. 140. Musa Log Model Initial Failure Rate Progression With Mixed Data

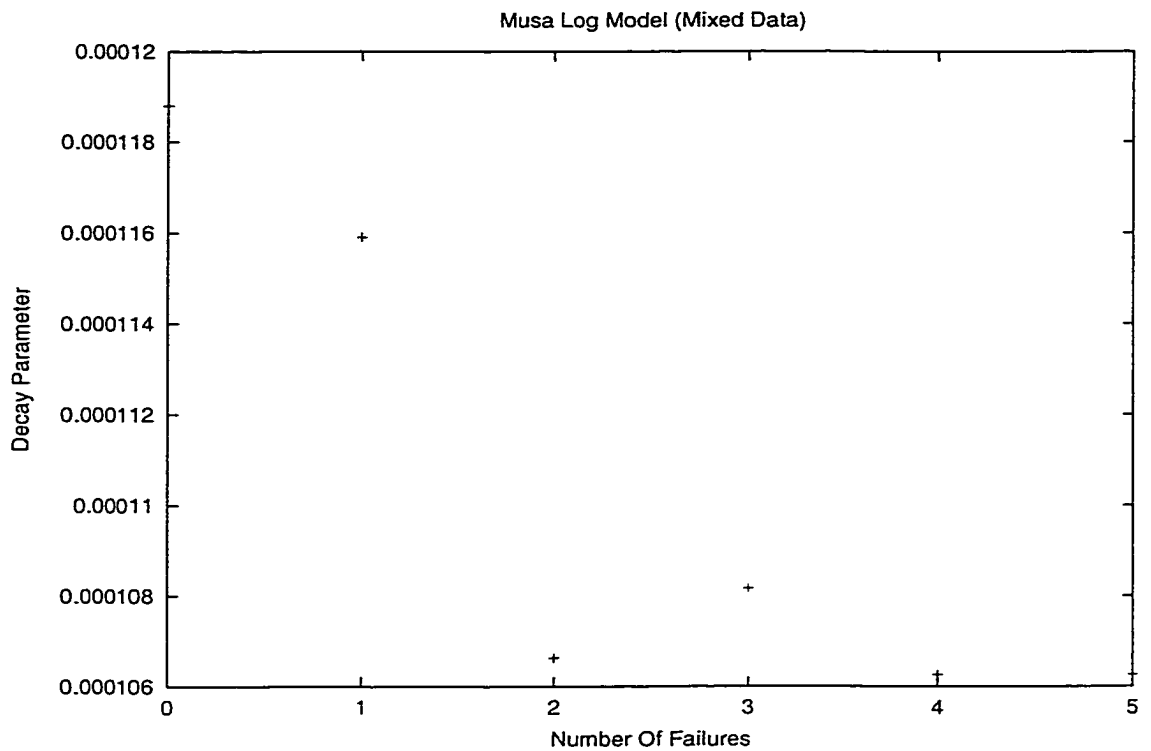

FIG. 141. Musa Log Model Decay Progression With Mixed Data

12.5.2 Conclusions

During this experiment, in addition to the results detailed above, some basic observations were made about the general nature of the software reliability testing process. These observations, in combination with observations made in previous experiments, support a move away from using program failure rates for reliability testing.

First, the amount of statistical noise present in the program failure rate data is apparent simply by looking at the failure data obtained during representative testing. It is hard to imagine any model being able to fit to this data. Similar results were observed in the previous experiments. All of the representative data sets examined during the course of this research exhibited this problem. Since all models performed similarly for all data sets during this research, it appears that any models based on the use of data obtained during testing will not provide any better estimates.

Second, several artifacts of the testing process were observed that caused misleading results because of the use of observed program failure rates for making reliability estimates. Specifically, we observed that two of the thirteen faults in the test software (Faults 11 and 12) cause an incorrect variable initialization when reading in the history values for the associated tags at system startup. Since these history values are only used during the first hour of the software's execution, these faults will only manifest as failures during the first hour of testing. Thus, the interfailure time will always be low for such a manifestation, and a large fault failure rate would be assumed for these faults. This problem is made worse by the fact that during testing the test software will be restarted quite frequently, giving these faults more of an opportunity to manifest and thereby introduce misleading data into the reliabililty model. During actual use the software, a restart only occurs a few times a year. During the actual estimation of the fault failure rates conducted at the start of this experiment, it was found that these faults have low fault failure rates (0.000067204 and 0.000000093, respectively).

Therefore, we have direct evidence that the use of observed program failure rates can lead to poor reliability estimates and predictions. This evidence supports

our suggestion that the emphasis in software reliability modelling be shifted from the use of program failure rates to fault failure rates wherever possible. The Order Statistics Model developed in this dissertation represents a first step in this direction.

During the course of this experiment, all of the models performed similarly for the observed failure data. However, by using the mixed method approach to testing developed earlier in this dissertation, the models required only about half of the number of test cases to provide their estimates.

During this experiment, several existing models were modified to allow them to provide estimates of fault failure rates in addition to the estimates of program failure rates that they already provide. This modification enabled these models to use the data from the mixed method approach to testing, with similar results to those obtained by the Order Statistics Model.

Chapter 13

Conclusions And Future Directions

In this chapter, the results of the research described in this disseration are summarized and future areas of research are suggested.

13.1 Results Of This Work

This section describes the results of this work with respect to each of the goals defined in Chapter 4.

13.1.1 Development Of A Mixed Method Approach To Testing

Prior to this work, software reliability testing was conducted using only representative testing methods. In this work, a mixed method approach to testing was developed that employs both representative and directed testing methods. Each type of testing method is used when it is most efficient to do so, in order to accelerate the reliability testing process. During the final two experiments discussed in this dissertation, using the mixed method approach to testing required far fewer test cases than were required by representative testing.

13.1.2 Development of Techniques To Allow Reliability Estimation Regardless Of The Testing Method Used

Observed failure times during testing can only be used for reliability estimation if testing is conducted using representative methods. Since the mixed method approach to testing uses directed testing methods for a portion of the testing process, failure data gathered using this process cannot be used for reliability modelling.

One of the goals of this research was to develop techniques for obtaining failure rate data that does not depend on the way that a program is tested. This goal was accomplished by switching the emphasis of data collection from quantities observed during testing to quantities obtained during debugging.

Specifically, the random variable of interest for reliability estimation is switched from observed program failure rates to individual fault failure rates. Several methods for estimating fault failure rates were suggested and several of these methods were used in the experiments conducted during this work.

13.1.3 Development Of A Software Reliability Model Capable Of Using Directed Testing Data

Traditional software reliability models require that failure data take the form of observed failure times during representative testing. During the course of this research, a software reliability model based on order statistics was developed. This model differs from existing models, in that it allows data from both representative and non-representative testing to be used as input. All of the experiments conducted during this work confirmed that the Order Statistics Model provides estimates and predictions as good as or better than the existing models that it was compared to.

13.1.4 Development Of Hybrid Versions Of Existing Models

Several existing models were modified to allow them to use fault failure rate data, in addition to the program failure rate data that they already use. This modification enabled these models to use the data from the mixed method approach to testing to make reliability estimates. These hybrid models performed similarly to the Order Statistics Model. The techniques used to convert these models could also be used with other models.

13.1.5 Evaluation Of The Suitability Of Time Based Data For Reliabilitly Estimation

As each experiment was conducted during this research, it became more and more apparent that observed program failure rates are poorly suited for making accurate reliability estimates and predictions. By their very nature, observed program failure rates will vary widely as testing is conducted. In order to minimize the noise and to provide stable data to reliability models, one must either combine multiple occurrences of a failure to obtain a better picture of the true current program failure rate, or move away from the program failure rate. In any event, it has become clear that using program failure rates based on software failures during testing does not provide good data to use as the basis of reliability modeling.

13.2 Future Directions

The work completed during the course of this research has provided some insight into the software reliability testing process and has suggested some ways to improve this process. As with any research, this work can be extended in several directions. This section outlines some of these potential areas of future research.

13.2.1 Refinement Of Methods Of Fault Failure Rate Estimation

Several techniques for determining fault failure rates were suggested in this dissertation. Some of these techniques were used during this work, but further development of these techniques will allow fault failure rate estimates to be made more accurately and at lower costs, especially in the absence of existing representative data.

13.2.2 Development Of A Model That Uses Only Fault Failure Rates

Another possible direction of future research is the development of a software reliability model that uses fault failure rates exclusively. During the course of this research, it became increasingly clear that program failure rates are not suited for making reliablity estimates because of their noisy nature. A model based solely on fault failure rates should provide more stable and accurate estimates and predictions than any of the existing models that rely on program failure rates.

It would be interesting to design this model in such a way that the software reliability engineering process could parallel the engineering process that is already used by other disciplines. For example, when engineers design a dam, it may be built to withstand a one hundred year flood, but maybe not a five hundred year flood. The reliability testing process could follow a similar procedure. Instead of working with the number of faults remaining in the program, the model would have a parameter that specifies a threshhold value that denotes the smallest fault that is of interest. For example, the analyst would be able to specify that he doesn't care about faults that occur less often than once a year. The output of the model could then be used to determine if the software has met this reliability criteria. In other engineering disciplines, such design methodologies are currently guided by the use of asymptotic order statistics. Thus, this new model would be a variation on the model developed in this dissertation.

Bibliography

[1] S. Brocklehurst and B. Littlewood. New ways to get accurate reliability measures. *IEEE Software*, pages 34–42, 1992.

[2] R. Butler and G. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions On Software Engineering*, pages 3–12, 1993.

[3] R. Cobb and H. Mills. Engineering software under statistical quality control. *IEEE Software*, pages 44–54, 1990.

[4] H.A. David. *Order Statistics*. John Wiley and Sons, Inc., 1970.

[5] M. Friedman and J. Voas. *Software Assessment*. John Wiley Sons, Inc, 1995.

[6] A. Goel and K. Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions On Reliability*, pages 206–211, 1979.

[7] D. Hamlet. Are we testing for true reliability? *IEEE Software*, pages 21–27, July 1992.

[8] D. Hamlet and J. Voas. Faults on its sleeve: Amplifying software reliability testing. *Proceedings Of The International Symposium On Software Testing And Analysis*, pages 89–98, 1993.

[9] Mary Ann Hoppa and Larry W. Wilson. Some effects of fault recovery order on software reliability models. In *Fifth International Symposium on Software Reliability Engineering (ISSRE 94)*, pages 338–342, Los Alamitos, CA, November 1994. IEEE Computer Society press.

[10] Howden. Weak mutation testing and completeness of test sets. *IEEE Transacations On Software Engineering*, SE-8(2):371–379, July 1982.

[11] W. Howden. Functional program testing. *IEEE Transactions On Software Engineering*, SE-6(2):162–169, March 1980.

[12] J. Musa A. Ianno and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill Book Company, 1987.

[13] Z. Jelinski and P. Moranda. Software reliability research. *Statsical Computer Performance Evaluation*, pages 465–484, 1972.

[14] D. Richardson L. Clarke, A. Podgurski and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions On Software Engineering*, 15(11):1318–1332, November 1989.

[15] B. Littlewood. Stochastic reliability-growth: A model for fault removal in coputer programs and hardware-design. *IEEE Transactions On Reliability*, pages 313–320, 1981.

[16] P. Maurer. *Reference Manual For A Data Generation Language Based On Probabilistic Context Free Grammars.* Department Of Computer Science And Enginerring, University Of South Florida.

[17] B Mitchell and S Zeil. A reliability model combining representative and directed testing. *Proceedings of the Internation Conference On Software Engineering*, 1996.

[18] P. Moranda. Predictions of software reliability during debugging. In *Proceedings Annual Reliability and Maintainability Symposium*, pages 327–332, Washington D.C., 1975.

[19] J. Musa. Software reliability data. *Bell Telephone Laboratories*, 1979.

[20] J. Musa. Operational profiles in software reliability engineering. *IEEE Software*, pages 14–32, March 1993.

[21] G. Myers. *The Art Of Software Testing.* John Wiley and Sons, Inc., 1979.

[22] S. Yamada M. Ohba and S. Osaki. S-shaped reliability growth modeling for software error detect ion. *IEEE Transactions On Reliability*, pages 475–478, 1983.

[23] T. Ostrand and M. Balcer. The category-partition method for specifying and generating functional tests. *Communications Of The ACM*, pages 676–686, 1988.

[24] R. Lipton R. DeMillo and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, pages 34–41, 1978.

[25] S. Rapps and J. Weyuker. Data flow analysis techniques for test data selection. In *Proceedings of the Sixth International Conference On Software Engineering*, September 1982.

[26] Y. Malaiya Naixin Li J. Bieman R. Karcich B. Skibbe. The relationship between test coverage and reliability. *IEEE Transactions On Software Engineering*, pages 186–195, 1994.

[27] J. Voas. Pie: A dynamic failure based technique. *IEEE Transactions On Software Engineering*, pages 717–727, 1992.

[28] E. Weyuker. On testing non-testable programs. *The Computer Journal*, pages 465–470, 1982.

[29] L. White. Software testing and verification. *Advances In Computers*, pages 335–391, 1987.

[30] Lee J. White and Edward I. Cohen. A domain strategy for computer program testing. *IEEE Transactions On Software Engineering*, SE-6(3):247–257, May 1980.

[31] C. Wild, S. Zeil, J. Chen, and G. Feng. Employing accumulated knowledge to refine test cases. *Software Testing, Verification, and Reliability*, 2(2):53–68, July 1992.

[32] S. Zeil. Pertubation techniques for detecting domain errors. *IEE Transactions On Software Engineering*, pages 737–746, 1989.

[33] S. Zeil, A. Biser, L. Cai, H. Huang, T. Ireland, B. Mitchell, and G. Walker. A formal specification of the rsdimu inertial navigation system. Technical report, Old Dominion University, 1993.

Vita

Brian Michael Mitchell
Department of Computer Science
Old Dominion University
Norfolk, VA 23529

Brian Mitchell received a B.S. in Mathematics from Hampden-Sydney College in May of 1991. He received a M.S. in Computer Science from Old Dominion University in August of 1993. He works as an independent contractor developing software.

Typeset using LaTeX.