Software Reliability Issues: An Experimental Approach

Mary Ann Hoppa
Old Dominion University

SOFTWARE RELIABILITY ISSUES: AN EXPERIMENTAL APPROACH

by

Mary Ann Hoppa
B.S., B.A., June 1981, Auburn University, Auburn, Alabama
M.S., January 1986, George Mason University, Fairfax, Virginia

A Dissertation submitted to the Faculty of Old Dominion University in Partial Fulfillment of the Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
December, 1995

Approved by:

Dr. Larry W. Wilson (Advisor)

Dr. Michael Dorvak

Dr. C. Michael Overstreet

Dr. J. Christian Wild

Dr. Steven J. Zeil

ABSTRACT

SOFTWARE RELIABILITY ISSUES: AN EXPERIMENTAL APPROACH

Mary Ann Hoppa
Old Dominion University
Advisor: Dr. Larry W. Wilson

In this thesis, we present methodologies involving a data structure called the debugging graph whereby the predictive performance of software reliability models can be analyzed and improved under laboratory conditions. This procedure substitutes the averages of large sample sets for the single point samples normally used as inputs to these models and thus supports scrutiny of their performances with less random input data.

Initially, we describe the construction of an extensive database of empirical reliability data which we derived by testing each partially debugged version of subject software represented by complete or partial debugging graphs. We demonstrate how these data can be used to assign relative sizes to known bugs and to simulate multiple debugging sessions. We then present the results from a series of proof-of-concept experiments.

We show that controlling fault recovery order as represented by the data input to some well-known reliability models can enable them to produce more accurate predictions and can mitigate anomalous effects we attribute to manifestations of the fault interaction phenomenon. Since limited testing resources are common in the real world, we demonstrate the use of two approximation techniques, the surrogate oracle and path truncations, to render the application of our methodologies computationally feasible outside a laboratory setting. We report results which support the assertion that reliability data collected from just a partial debugging graph and subject to these approximations qualitatively agrees with those collected under ideal laboratory conditions, provided one accounts for optimistic bias introduced by the surrogate in later prediction stages. We outline an algorithmic approach for using data derived from a partial debugging graph to improve software reliability predictions, and show its complexity to be no worse than $O(n^2)$. We summarize some outstanding questions as areas for future investigations of and improvements to the software reliability prediction process.

Acknowledgements

The author gratefully recognizes the support of family, faculty, friends and colleagues during the seven years invested in attaining this goal.

Contents

Page

Page

Page

List of Tables

Table **Page**

List of Figures

Chapter One

Introduction

The development of robust models for estimating the reliability of software and for predicting achievable reliability remains one of the foremost computer science research areas. It has been observed that measuring the reliability of a program has proved to be an unexpectedly challenging task for over two decades [4]. Despite persistent research activity, software reliability assessment and prediction continue to be elusive problems. While many predictive software reliability models have been proposed, no one model has yet emerged as universally applicable; nor is it clear if such robustness can be realized. Feedback gained from controlled, repeatable experiments is particularly needed to determine the efficacy of existing models and to devise the means to assess their suitability for a given project.

Typically, predictive software reliability models use a single debugging pass, resulting in one sequence of times to failure as inputs to stochastically predict reliability and the related quantities of failure rate and mean time to next failure. This fails to recognize that, had a different stream of inputs or different testing techniques been used, the order in which the faults are discovered, and hence the sequences of times to failure, might vary radically. Uncertainty about the order of fault recovery is further compounded in that a sample of size one is used to represent the failure rate of the software for each stage of the fault removal process.

When an observed process remains the same over a long period of time, then a great deal or all of the data derived from that process should be used to model it. If, however, there is significant change in the process, it may be possible to more accurately represent it by excluding or weighting some observations. Although such data aging techniques as the moving average and exponential smoothing are frequently used in other fields, it has been only recently that similar approaches for filtering failure data to display trends in accordance with various reliability growth models' assumptions have been addressed [28]. This process is further complicated since, while it is now widely accepted that individual faults fail with different rates, to date little work has been done in examining changes in reliability trends that could be caused by interdependency of faults.

1.1 Purpose

The concerns discussed above initially motivated us to consider the fact that the failure data used in software reliability models are derived from only one of many possible repair processes. Assume data from n failures are being used, and that once a failure has occurred, its repair is installed before proceeding to the next debugging iteration. Then there are $n!$ possible orders in which those faults could be individually identified and repaired — any one of the n faults on the first debugging iteration, followed by any one of

(n-1) remaining faults on the second iteration, and so on until the last known fault is removed.

We address this issue by experimentally considering multiple orders for fault removal. We mitigate the potential for a sequence of single time to failure observations to induce randomness in an algorithm's predictive accuracy by using an average observed failure rate for each iteration rather than a sample of size one. Our approach to choosing specific fault recovery orders is based on a relative fault size criterion rooted in observing the failure rate attributable to each known fault. We experimentally consider the potential for the presence of various combinations of other faults, both known and unknown, to affect the relative fault sizes and size ordering.

While laboratory investigations are useful to evaluate a model's performance and to compare various models, the means must exist for translating the lessons learned into techniques applicable to the domain of software development. This motivated two further efforts. First, we propose and evaluate an alternative way to study models' predictive accuracy when a perfectly reliable benchmark is not available. We go on to study a data aging criterion, path truncation, which controls the computational complexity of the proposed methodologies. Our overall goal is to determine ways for improving software reliability predictions from existing models, and to assist practitioners in choosing model(s) applicable to their project.

1.2 Overview of Paper

In this paper we present results from a number of experiments designed to investigate the issues sketched above. In Chapter 2 we provide background and summarize related work to establish a basis for subsequent discussions. In Chapter 3 we describe the debugging graph data structure which we used as a testbed for our experiments. Chapters 4 through 8 detail the specific experiments we conducted on the following topics:

- establishing relative fault sizes;
- investigating faults whose failure behavior appears to change in the presence of other faults;
- studying the effects of varying the fault recovery order on models' predictive accuracy;
- using a substitute for the gold version program in empirical reliability measurement; and
- choosing subsets of the known failure data to limit the computational effort of debugging graph applications.

Chapter 9 outlines some practical methodologies for real-world application based on the experimental results. Finally, Chapter 10 summarizes our findings and recommends possible avenues of future research.

1.3 Notational Conventions

Subsequent to this chapter, the initial use of any term or acronym listed in the glossary is given in boldface italic type (e.g., ***Mean Time to Failure (MTTF)***), while program and file names are printed in boldface type (e.g., **LICCtrl**). Per standard practice, italic type is used throughout the paper for emphasis.

Chapter Two

Background and Related Research

In this chapter, we define the software reliability problem. After establishing a consistent terminology to be used throughout the thesis, we describe the general software reliability problem from an historical viewpoint as well as its contemporary definition as a prediction system. We cite significant factors that continue to make a generic software reliability model so elusive. We describe the essentials of the four well-known software reliability models used in our experiments. We summarize the focus of this thesis and previous research which led to the concepts it explores.

2.1 Terminology

Let P be a program of finite length which has been written to satisfy a particular set of specifications including a pre-defined execution time deadline. We define ***failure*** as a departure of the external results of P's execution from its requirements on a particular run. A ***run*** consists of a single execution instance of P involving the transformation of an input case to an output (or abnormal termination). A ***fault*** or ***bug***, then, is defective, missing or extra code that is the cause of one or more failures for the program. The collection of input cases for which failures are observed due to some fault is commonly called its ***fail set***.

Software reliability (R) is the probability of a software product operating for a given period of time in a particular environment without exhibiting any failures. In many instances, the number of input cases is proportional to the execution time. We will assume this to be true for the remainder of the thesis. This will allow us to use the average time of computation for an input case as the given time period, and R becomes the probability of success per input. The ***failure rate*** ($F = 1 - R$) expresses the probability that a software product will exhibit a failure during a given time period in its specified environment. Assuming an exponential distribution, the ***mean time to failure (MTTF)*** is 1/F.

We use the term ***fault recovery*** to mean the identification of faults and the implementation of suitable code ***repairs*** whose installation removes those faults from the program. For purposes of discussing this preliminary material, a ***debugging session*** consists of the recovery of some collection of n known faults from a program.

2.2 Historical Context of Reliability

Conventional hardware reliability theory is concerned with determining the probability that physical components perform their required function under controlled conditions for a specific period of time. Progress has been achieved in this discipline through a concentration on the random processes of physical failures and statistical methodologies for capturing how the reliability of a complex hardware system depends on the reliability of its constituent components.

The general failure pattern of hardware has been naturally categorized by three periods of operation:

- the early failure period, in which failures are observed in inherently weak parts that were improperly designed, manufactured or used;
- the constant failure period, or useful life of a system, in which failures occur infrequently, at a random and uniform rate; and
- the wear-out failure period, during which components rapidly deteriorate.

It was found that the reliability function of an entire hardware system can be represented by the product of the individual reliability functions for each component, each obeying the so-called ***bathtub curve*** shown in Figure 1 [30].

Such success in the hardware arena encouraged concurrent and at times analogous research into determining the reliability of software. This newer problem is concerned with how well a given program or software system functions with respect to customer requirements. In classical reliability terms, software reliability can be described as the probability that at a specified time T, the system is operating and will continue to work without failure over a subsequent time interval [19]. A software failure is defined broadly as "not meeting some user requirement." Notwithstanding many significant strides in our understanding of the qualitative behavior of software failures, some experts feel that we still lack the means to definitively quantify them [5].

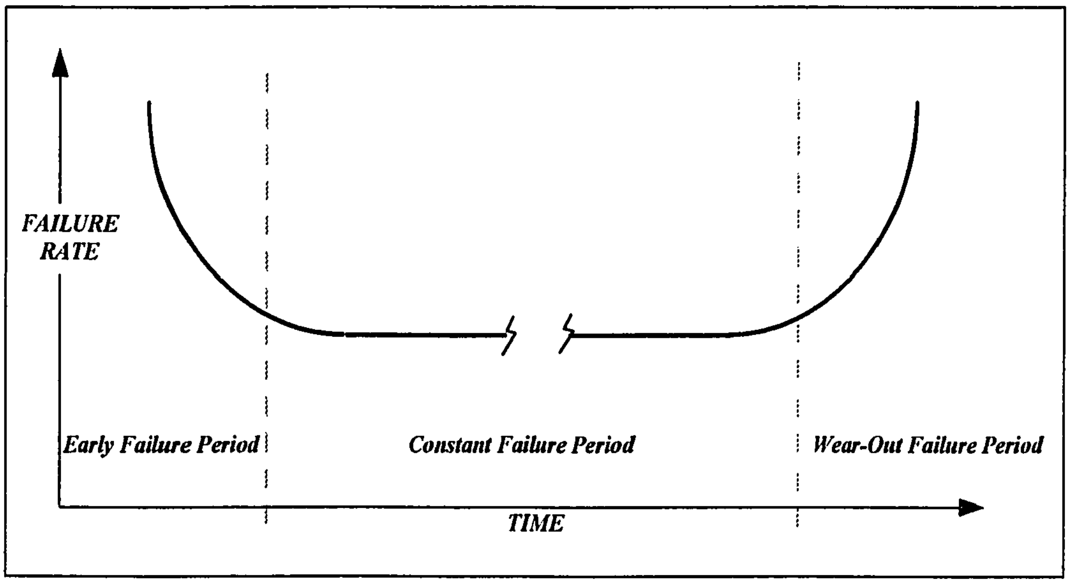

Figure 1. Typical Bathtub Curve of Failure Rate Versus Time

2.3 The Software Reliability Problem

In its simplest form, the software reliability problem has two themes:

- How to make predictions about the future performance of a piece of software, or to assert that some pre-defined level of reliability has been achieved; and
- How to know that any such statements are trustworthy.

Reliability growth models attempt to quantify the operational "goodness" of software based on data gathered during some phase(s) of the software lifecycle.

Although different models vary considerably in the details of their mathematical structure, the basics of the problem can be summarized as follows. The user has available some raw data that are — or can be used to derive — a sequence of execution times t_1, t_2, ..., t_{i-1} between successive software failures. These times are the realizations of random variables T_1, T_2, ..., T_{i-1}. The objective is to use these past observations to predict the future unobserved realizations of T_i, T_{i+1}, ... and so on. That is, at ***stage*** i, when the first (i-1) failures have already been repaired, the goal is to predict the future failure behavior. Details can be found in the literature [2, 5, 6, 13, 15, 16, 17, 18, 19, 20, 22, 26, 29] and later in this paper of various attempts to model these processes.

2.3.1 As a Prediction System

In many cases, the user will be satisfied simply to know the current reliability of the software. Alternatively, the user may wish to predict when a target realiability will be achieved, perhaps to mark a satisfactory end to software debugging. The predictive element of the software reliability problem is sometimes overlooked, even in the technical literature. Authors often "validate" a model by showing it accurately explains past failure behavior. But the ability to capture the past accurately does not necessarily imply an ability to predict accurately [8, 21]. In either case, the software reliability problem can really be regarded as the specification of a prediction system that allows the estimation of future T_i, T_{i+1}, ... from the past times t_1, t_2, ..., t_{i-1}. The system consists of three components:

- the probabilistic model of the T_i's in terms of one or more (unknown) parameters;
- a statistical inference procedure for estimating the parameters based on the known realizations of (past) T_i's; and
- a prediction procedure that uses the probabilistic model and the inference procedure for making probability statements about future T_i's [2].

2.3.2 Obstacles to Effective Modeling

A generic, universally applicable software reliability model has yet to emerge. Likewise, no one has yet succeeded in formulating positive criteria for selecting the use of any of the existing software reliability models in particular cases. Negative criteria exist for disqualifying certain models against a given set of observed data based on numerical nonconvergence [35]. The existing software reliability models fail to account for reliability improvement variance introduced by potentially different debugging orders. Therefore, understanding the nature of debugging variability merits further research to determine whether controlling or accounting for this variability can be used to improve reliability predictions and to identify appropriate software reliability models.

2.3.2.1 Inability to Quantify Reliability of Software

While we can assert with a high level of confidence that every software system of more than trivial complexity contains errors, such faults may lay dormant for arbitrary lengths of time, and we can never affirm that the last design or implementation flaw in the software has been found [1]. Considerable effort has been spent in developing techniques for measuring the number of software faults in a program and predicting how long before the next failure manifests itself under given operational scenarios. As pointed out in Section 2.3, this has resulted in an abundance of mathematical software reliability models.

2.3.2.2 Lack of a Universal Software Reliability Model

Unlike hardware, software failures do not result from aging components. They arise from errors in the specifications of desired functionalities for the subject system, conceptual design errors and implementation inaccuracies such as incorrect combinations of computer language instructions. Despite the large number of models available, still no universally applicable software reliability model exists that can be recommended as giving accurate predictions in all circumstances; nor are we equipped with the knowledge to decide in any given context which of the existing models, if any, would be most appropriate to use [5].

2.3.2.3 Inherent Complexity of Software

The complexity of software is an inherent and essential property [9]. Some of the difficulties of software reliability modeling, then, may be attributed to the nature of software itself. Another factor to consider is that every change to a software system, such as code repairs to fix operational failures, creates a new system having different properties from the original one [32]. In our zeal to correct software flaws as we find them, we may be mutating many of the characteristics one might think to use as a basis for, or as selection criteria of, software reliability models.

2.3.2.4 Randomness of Debugging Data

Another difficulty in reliability modeling is the randomness present in data generated by the debugging process. Debugging activity can be regarded as the recovery of n software faults in some arbitrary order. Suppose one starts with multiple copies of the same undebugged piece of software and applies a different collection of test input cases — or a unique series of random inputs — to each copy. Assume that once a failure is encountered, repairs are made to correct it before debugging continues. It would not be surprising to find a different recovery ordering of the n bugs in each replicate, as well as variance in the reliability improvement after the i^{th} bug is removed in each replicate.

It has been conjectured that such variance fails to be adequately accounted for in existing reliability models, which make predictions based on a single debugging session. Additionally, at each predictive stage, a single realization of time to failure is typically used to formulate the next input to the model, rather than the average of many trials. It has been shown that variance resulting from these practices can play a large role in the poor performance of two well-known software reliability models in the general case [34].

2.3.3 Four Well-Known Models

The models examined in the experiments documented in subsequent chapters are Jelinski-Moranda [17], Geometric De-Eutrophication [23], Basic Musa [22], and Logarithmic Poisson [24], whose assumptions and algorithms are well known from the literature. Their characteristics are summarized here for the sake of completeness, and plots of failure rate versus execution time, which originally appeared in [23, page 327] and [22, page 42], are shown in Figure 2. The models' fundamental formulae for maximum likelihood parameter estimates are also provided. We validated our C programming language implementations of these four models prior to data collection. This analysis is given in Appendix D.

2.3.3.1 Jelinski-Moranda

The Jelinski-Moranda model assumes that all faults contribute equally to the unreliability of the program, so that the plot of failure rate versus time is a step function in which each step essentially represents one "error's worth" of hazard. The model estimates

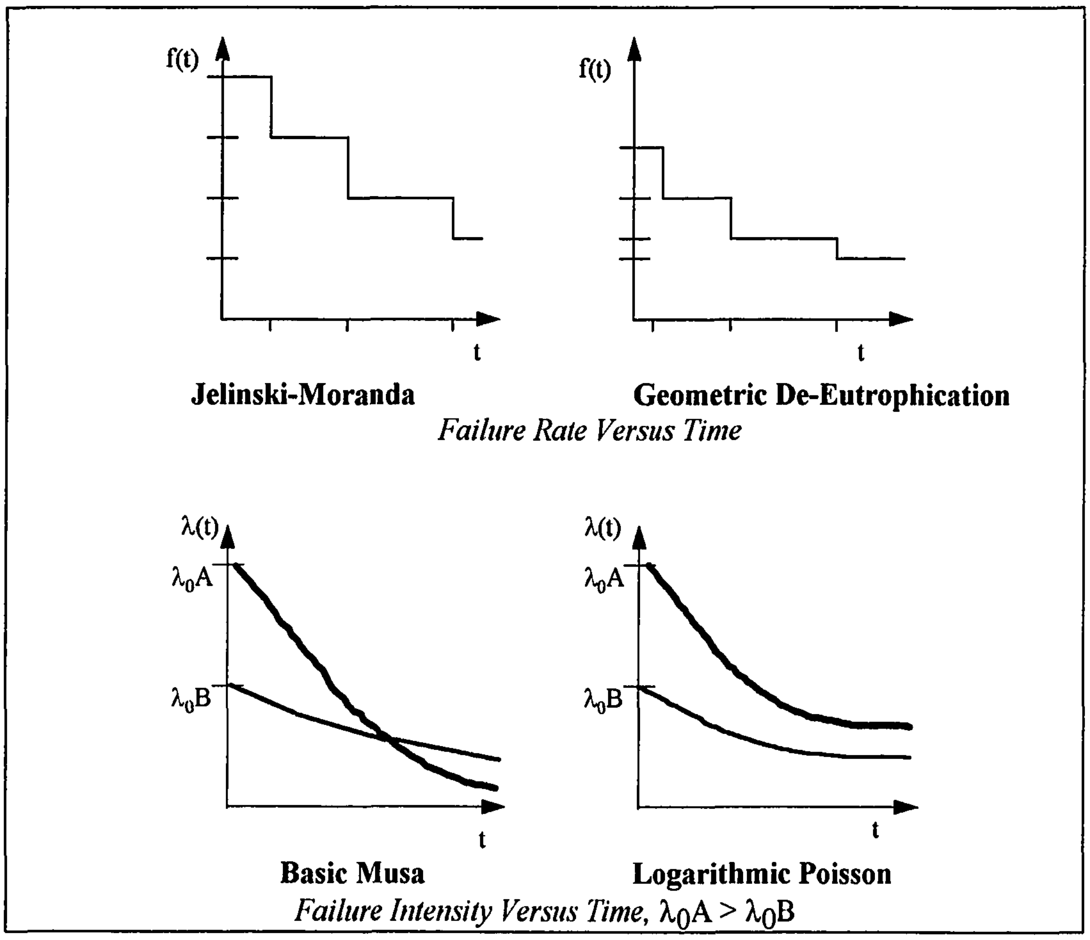

Figure 2. Some Graphical Depictions of Failure Rate Versus Execution Time

the total number of errors in a program, N, by determining a value of N for which the following two functions are equal:

$$\Sigma_{i=1,n}[1 / (\mathrm{N} - (i\text{-}1))] \quad \text{and} \quad n / (N - (\Sigma_{i=1,n}[(i\text{-}1) \cdot X_i)]) / T)$$

In these equations:

N represents the total number of errors in the program;

n represents the number of failures observed;

X_i represents the time at which the i^{th} failure was observed;

T is the sum of all X_i's.

A proportionality constant, ϕ, is then estimated by using the estimator for N in the following formula:

$$\phi = n / ((N \cdot T) - \Sigma_{i=1,n}[(i\text{-}1) \cdot X_i])$$

Thus, after the i^{th} error has been found, the residual number of errors in the software is estimated to be $(N - n)$, while the failure rate F is $(N - i) \cdot \phi$.

2.3.3.2 Geometric De-Eutrophication

In an attempt to describe testing in which an accumulated group of faults is corrected simultaneously or the hazard contributions of faults are not equal, the Geometric De-Eutrophication model instead assumes a plot of failure rate versus time in which the step size decreases in a geometric sequence with each subsequent fault removal. D represents the initial detection rate. It holds until the first error is found, at which time the rate becomes $k \cdot D$, where $0 < k < 1$. In general, the detection rate is $k^i \cdot D$ after the ith error has been found, with the detection rates forming a converging geometric series.

The model estimates a value for the proportionality constant k for which the following two functions are equal:

$$(n + 1) / 2 \quad \text{and} \quad (\Sigma_{i=1,n}[\, i \cdot k^i \cdot X_i\,]) / (\Sigma_{i=1,n}[k^i \cdot X_i\,])$$

In these equations:

n represents the number of failures observed;

X_i represents the time at which the ith failure was observed.

D can then be estimated by using k's estimate in the following formula:

$$D = n / (\Sigma_{i=1,n}[k^{i-1} \cdot X_i\,])$$

F is easily estimated using the formula $D \cdot k^n$.

2.3.3.3 Basic Musa

The Basic Musa model, the continuous analogue to Jelinski-Moranda, assumes that failure intensity decreases by a constant amount regardless of which failure is repaired, with the physical interpretation that all errors are equally likely to occur, but are embedded in a continuous rather than a stepwise function. A value b_l, which represents the ratio of initial failure intensity over the total number of bugs in the program, is estimated by solving the following:

$$m_e / b_l - m_e \cdot t_e / (\exp(b_l \cdot t_e) - 1) - \Sigma_{i=1,me}[t_i\,] = 0$$

In this equation:

m_e represents the number of bugs removed;

t_i represents the time the i^{th} bug was removed;

t_e represents the time at which testing ended.

An estimator for b_0, the total number of bugs in the software, is obtained by using b_1's estimator in the following formula:

$$b_0 = m_e / (1 - \exp(-b_1 \cdot t_e))$$

The number of bugs removed by time t is then given by the function:

$$u(t) = b_0 \cdot (1 - \exp(-b_1 \cdot t))$$

The failure rate is estimated using the function:

$$\lambda(t) = b_0 \cdot b_1 \cdot \exp(-b_1 \cdot t)$$

2.3.3.4 Logarithmic Poisson

The Logarithmic Poisson model, as the continuous counterpart to Geometric De-Eutrophication, exhibits a failure intensity decrement which becomes exponentially smaller with subsequent fault removals, so that the first repair yields substantial improvement, while the effects of later repairs are much smaller. A value b_1, which estimates the product of the initial failure intensity and an intensity decay parameter, is obtained by solving the following:

$$1 / b_1 \cdot (\Sigma_{i=1,me}[1 / (1 + b_1 \cdot t_i)]) - m_e \cdot t_e / ((1 + b_1 \cdot t_e) \cdot \ln(1 + b_1 \cdot t_e)) = 0$$

In this equation:

m_e represents the number of bugs removed;

t_i represents the time the i^{th} bug was removed;

t_e represents the time at which testing ended.

An estimator for b_0, the inverse of the intensity decay parameter, is obtained by using b_1's estimator in the following formula:

$$b_0 = m_e / \ln(1 + b_1 \cdot t_e)$$

The number of bugs removed by time t is then given by the function:

$$u(t) = b_0 \cdot \ln(1 + b_1 \cdot t)$$

The failure rate is estimated using the function:

$$\lambda(t) = b_0 \cdot b_1 / (1 + b_1 \cdot t)$$

2.4 Focus of Research

Our contribution is two-fold. First, we present new methodologies for improving predictions from existing software reliability models, and for assisting practitioners in choosing relevent model(s) for their project. We approach this problem in the laboratory, where a perfect version of the tested program can be used to assess the failure rates of known bugs. Our experiments control the presentation order of fault recovery data to the models and investigate the potential for observed failure rates to change based on the current debugging state of the software. These methodologies use a data structure called the debugging graph as an analysis aid. Second, we propose means for translating these laboratory approaches into real-world methods by demonstrating the utility of an alternative to using the perfect program to assess reliability and by applying data aging techniques.

2.5 Related Work

Here we detail previous experiments and published research which motivated our investigations.

2.5.1 Replicated Debugging

The idea of repetitively debugging a software module was first advocated by Nagel and Skrivan [25] to provide better estimates of program error rates as well as the error rates associated with individual faults. In this ***repetitive run modeling*** approach, the first debugging replication proceeds until n faults are recovered using randomly generated inputs based on a program usage distribution. The repair corresponding to each fault is identified and saved. On subsequent replications, the software is returned to its initial faulty state and debugged again — using the known repairs — with a different set of randomly generated inputs. The goal is to repeatedly debug the software in order to obtain multiple instances of the sequence of interfailure times. By conducting many replications, Nagel and Skrivan determined, among other things, that faults occur with unequal error rates.

The significance of recovering bugs in different orders on separate replications inspired Wilson and Shen's debugging graph model discussed later in this paper [33]. Wilson and Shen also studied the effects of collecting and averaging multiple observations of the current time to next failure at each debugging stage via simulated replications, in which multiple times to failure for a given debugging stage were randomly drawn from the assumed underlying distributions of some well-known models. Their observation of improved predictive behavior as the sample size increased at each stage motivated our interest in the potential effects of the single sample convention now practiced [34].

2.5.2 Fault Interactions

Dunham [11] reported some preliminary investigations of one kind of fault interaction called ***compensation***. Two faults are compensatory if certain failures occur when either fault is in the program, but not when boths faults are present. The interaction can range from ***partial compensation***, in which some of the respective faults' failures still can be observed, to ***full compensation***, in which none of the respective faults' failures are manifested, when the faults are simultaneously present. Based on these investigations, the interaction phenomenon was advanced as a possible explanation for apparent changes in reliability improvement seen when installing certain repairs in the context of different bug sets, as well as for the varying quality of predictions given by existing software reliability models.

2.5.3 Data Aging

Data aging techniques are based on the assumption that older data may not be as representative of a current and future process as more recent data. Although methods such as the moving average and exponential smoothing are frequently used in other fields, only recently have reliability researchers begun investigating ways to choose a subset of all observed data for modeling the software failure process. Schneidewind reasoned that changes in reliability trends could be caused by such factors as fault dependencies and variation in the time between failure occurrence and fault correction; so by excluding or giving lower weight to earlier failures it might be possible to obtain more accurate predictions of future failures [28].

Using his Non-Homogeneous Poisson Process software reliability model and the Space Shuttle on-board flight software as case studies, Schneidewind examined three ways to determine an optimal value of s, an index in the range $1 \leq s \leq t$, which is the starting value of equal-length failure count intervals:

- Choose all the failures in the execution intervals from 1 to t. This method is used if it is assumed that all historical failure counts from 1 through t are representative of the future failure process.
- Exclude counts from 1 to s-1. This method is used if it is assumed that only the historical failure counts from s through t are representative of the future failure process.
- Use an aggregate count from 1 to s-1 and individual counts from s to t. This method is used if it is assumed that the historical cumulative failure count from 1 through s-1 and the individual failure counts from s through t are representative of the future failure process.

Schneidewind investigated the application of various criteria for estimating an optimal value of s which produces the most accurate predictions. For example, he treated the failure count interval index as a parameter by substituting model functions for data vectors and optimizing on functions derived from maximum likelihood estimation techniques. Alternatively, he used weighted least squares to maintain constant variance in the presence of the decreasing failure rate assumed by his model, as well as the familiar mean square error test. His investigations found that all proposed aging criteria produced better predictions for the Space Shuttle software than using all the failure data, whether those predictions were cumulative failures or times to next failure, and he suggested that other software reliability models could benefit from using data aging. Our experiments examine whether data aging is applicable to keeping the computational complexity of the debugging graph methodologies reasonable.

2.5.4 Recalibration

By allowing the user to estimate the relationship between the software reliability predicted by a model and the true reliability figure, the process of ***recalibration*** offers a very general way by which predictions can be improved. Much of the work in this area has taken place at the Centre for Software Reliability in London. Brocklehurst et al. subsumed some preliminary model adaptation work and proposed a systematic model recalibration approach to analyze and improve models' predictions by estimating the relationship between predicted and true accuracy. The fundamental technique is summarized here; details and mathematical justifications can be found in [5].

Having observed t_1, t_2, ..., t_{i-1}, a good estimate of $F_i(t)$ is sought; i.e., $P(T_i < t)$. The equivalent reliability statement of the preceding is $R_i(t) = 1 - F_i(t)$. An existing reliability model is used to calculate an estimate or a *predictor* of $F_i(t)$, denoted here as

$F_i(t)^*$. The difficulty of analyzing "how close" $F_i(t)^*$ is to the true $F_i(t)$ lies in the fact that $F_i(t)$ will always be an unknown. The deviations between the estimated and true values are viewed as two varieties: ***bias***, consistent deviation between prediction and reality; and ***noise***, large variability in the difference between prediction and reality [2]. The only information available to assess and account for these differences are the single samples of each of the random variables T_i whenever the software fails.

The informal approach to this analysis is that the user inspects the pairs $\{ F_i(t)^*, t_i\}$ to see if there is any evidence the the t_i's are not realizations of random variables from the $F_i(t)^*$'s, since such a departure would suggest that there are significant differences between the predicted and actual behavior. The assessment is done by drawing a ***u-plot***; an example is shown in Figure 3. For this graph, *n* previously calculated predictor $F_i(t)^*$'s — *possibly relabeled to be non-decreasing* — each with a value between 0 and 1, are located on the x-axis. The corresponding y values start at 0 and increase by 1 / (*n*+1) for each subsequent data point. The maximum absolute vertical difference between the plotted points and the line of unit slope can be used to make statements about the percentage level significance of the departure of the predicted values from reality. The relative amounts of plotted points above and below the line of unit slope can be used to make general statements about how optimistic or pessimistic the predictor tends to be.

A key assumption in the recalibration technique is that the relationship between the estimated $F_i(t)^*$'s and the true $F_i(t)$ of the random variable T_i's can be represented by a sequence of functions which change only slowly in most cases. Graphical techniques may

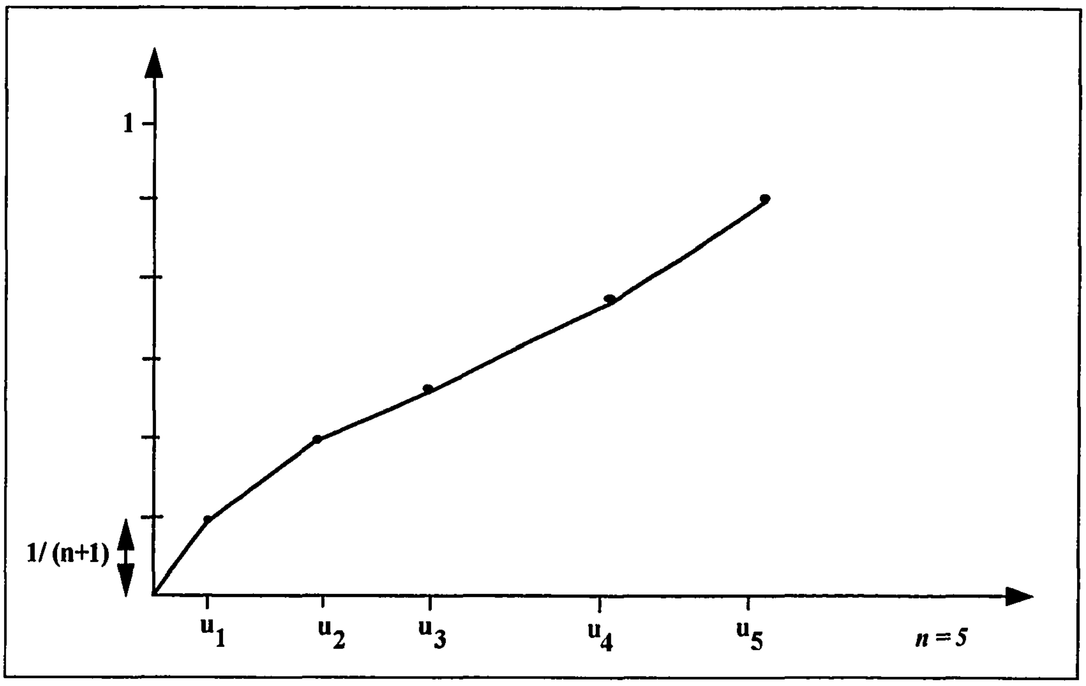

Figure 3. Example u-plot Recalibration Function

be used to increase confidence in this assumption by examining the u-plot data for trend. The simplest form of the recalibration function is then realized by using the u-plot with steps joined up to form a polygon. Various smoothing techniques may also be applied to assure continuity of the estimator plot's derivative.

After the basic prediction system is used to make a "raw" prediction of $F_i(t)^*$, this value is then located on the x-axis of the estimator u-plot and projected up to the polygon to interpolate the value needed to recalibrate the raw prediction. As information about the actual failure history increases, the estimator plot is suitably modified. Brockelhurst et al. used simulated data to validate their approach, and they also discuss techniques for choosing the best of available prediction systems for a given data source.

The recalibration approach exhibits several concepts we likewise have addressed in our studies. The technique supports an implicit reordering of the failure data to construct the recalibration function, which we address explicitly. Further, as failure history increases, the predictor function adapts to account for the effects of long-term trends and localized behaviors, which we address through a combination of failure data reordering and aging.

2.5.5 Preliminary Debugging Graph Investigations

The construction of two levels of a partial debugging graph (see Figure 4), then called an ***error graph***, was the subject of a 1990 Master's Degree project by two students in the Computer Science Department at ODU, White and Harbison [31]. Their investigations were intentionally limited so the project could be completed within a sixmonth time frame, and it was intended to lead to further research. Although they did not advance any formal hypotheses about the kinds of information from the graph that might be useful, they alluded to a desire to determine a unique path through the graph to use for standardizing data for existing reliability models. They also hoped to find a substitute for the ***oracle*** or ***gold version*** program (see Section 3.2.1) that is required for replicated debugging. This would represent a significant step forward since, outside the laboratory, one is not guaranteed the existence of a highly reliable, independently developed version of the software under test.

To constrain the size of the data collection problem, White and Harbison selected one of three independently developed versions of a software solution to a well-known launch interceptor problem having 12 known bugs. One of the bugs with a nearly 100% failure rate was repaired prior to data collection, and another bug with an extremely low incidence of detection was ignored, resulting in a total of ten significant bugs in the study. Data were collected using an existing oracle and instrumentation developed specifically for the subject software. A partial debugging graph was constructed by estimating reliability figures for variants of the program having either just one of the ten repairs installed, or nine of the ten repairs installed, using either 100,000 or 1,000,000 test cases.

White and Harbison then examined the potential application of delta graphs and surrogate oracles to eliminate to need for the oracle. Both techniques involve substituting a partially debugged software version for the independently developed, highly reliable baseline or "gold" program typically used to assess the performance of the software under test. The ***surrogate oracle*** refers to an arrangement in which the substitute program consists of the software under test with all known repairs installed. The ***delta graph*** is used to approximate the debugging graph and to show incremental improvement of the tested software. For example, when the bug subsets of the substitute oracle and the tested variant differ by just one element, this provides a technique for approximating the size of the bug represented by the difference of the subsets.

In inspecting their partial debugging graph, White and Harbison advanced several conjectures. They proposed that data derived from the debugging graph could be used to rank the bugs in a size order based on their relative reliability growth figures. They felt that some numerical relationship may exist between the deltas calculated for the delta graph and the reliability growth figures calculated in the debugging graph; however, as statistical analysis was beyond the scope of their research, they did not attempt to derive such a relationship. The reliability figures calculated using the surrogate oracle versus the gold version program were close enough to merit futher investigation, but appeared to contain some unexpected discrepancies which prevented any firm conclusions.

White and Harbison also noted an unexpected lack of correspondence in the changes in reliability produced by isolating the installation of a single repair at the start of debugging versus the removal of that same repair at the end of the debugging process. Although they suspected fault interactions (see 2.5.2), some errant effects in the test environment could not be discounted as possible sources of this behavior. By implementing a new test environment with all extraneous instrumentation from past experiments removed, a goal of our experiments is to definitively determine the utility of sizing the bugs, using the surrogate oracle and possible evidence of fault interactions.

Chapter Three

The Debugging Graph

In the following paragraphs, we describe the debugging graph, a data structure we used as the basis for studying the effects of various fault recovery orders on the predictive accuracy of software reliability models. We outline the general approach to building such a graph, then provide specific details for the subject software of our experiments.

3.1 Description

Suppose a program contains *n* known faults labeled 1...*n* respectively. There are *n!* possible orders in which the *n* faults could have been individually located and repaired. The debugging graph, as shown in Figure 4, is useful for representing these *n!* orders [25, 33]. The rows, or ***levels***, of the debugging graph are labeled from 0 to *n*, with row *i* representing stage *i* of a debugging process where *i* of the *n* bugs have been repaired. The term ***variant*** references any version of the original program with some subset of the known repairs installed. Each graph node represents a variant and is labeled

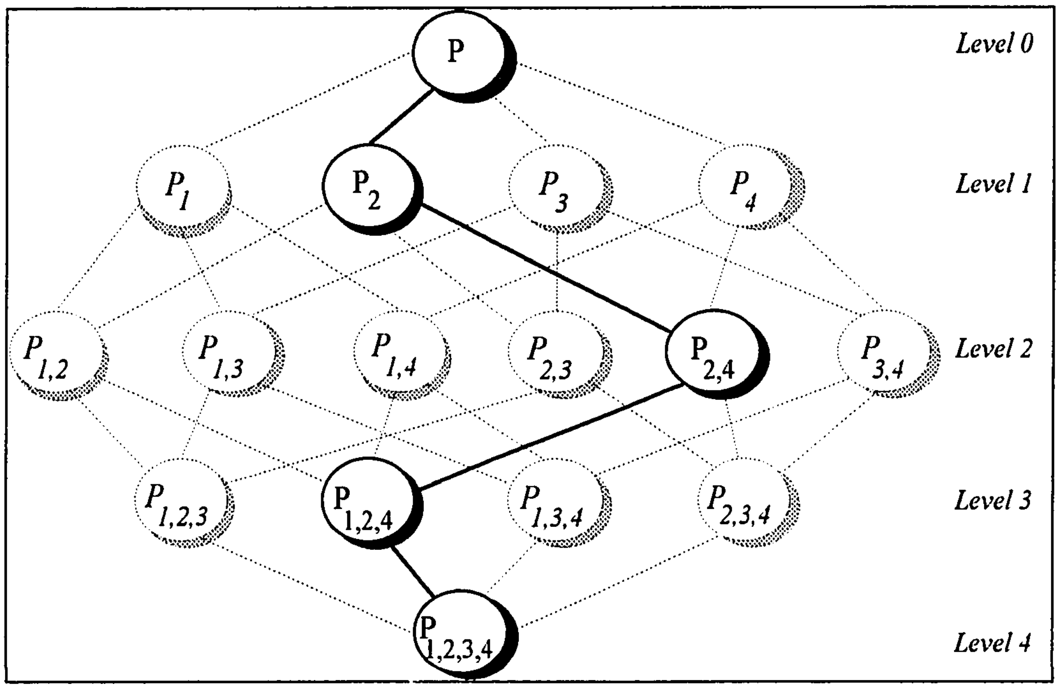

Figure 4. Debugging Graph for $n = 4$
With One Debugging Session Indicated

with P subscripted by the subset of $\{1,2,\ldots,n\}$ corresponding to the faults repaired in that variant.

Level 0 consists of a single node, labeled P, which represents the variant with no repairs installed. Likewise, level n constains a single node, labeled $P_{1\ldots n}$, which represents the software with all n known repairs installed. In general, at level m, $1 \leq m < n$, there are $m!/(n! \cdot (n-m)!)$ nodes and a total of 2^n nodes in the debugging graph. An edge in the graph represents a repair for one fault and connects one node to another one level lower, where the subscripts of the adjacent nodes differ in exactly one element. In general, at level m, $1 \leq m < n$, each node has $(n-m)$ edges joining it to nodes at level $(m+1)$. This results in a total of $2^{(n+1)}$ edges in the debugging graph. A ***debugging session*** removes all known bugs and is represented by a path in the debugging graph from P to $P_{1\ldots n}$ that follows edges through exactly one node at each of the levels 0 through n. Graph nodes and edges can also be labeled with significant quantities, as we will describe below.

3.2 Construction

3.2.1 Generic Approach

To physically realize the debugging graph, variants of the original software must be created containing subsets of the known repairs, and organized in such a way as to be amenable to testing. As described in 3.2.2.3, a directory structure may be set up to hold the program variants and to enable easy navigation among them. UNIX shell scripts as described in Appendix C can be used to automate the construction of the program variants by assembling “flawed” and “repaired” pieces of code. A suitable means for subjecting the variants to test cases must also be devised.

As an example, a ***control program*** is described in 3.2.2.2 which subjects variants to a random, but repeatable, input stream and tests their outputs for correctness via a process illustrated in Figure 5. To obtain empirical reliability estimates, the control program subjects each variant to a large set of inputs, generated randomly according to the prescribed usage distribution. In a laboratory setting, the number of inputs producing expected outputs can be determined by using an error detector which is implemented by comparing a variant’s outputs to those produced by a gold version of the program when both are subjected to the same input stream. Since the gold version always produces the correct outputs, a variant’s reliability can be estimated via such an error detector using the following calculation:

R = (number of “expected” outputs) / (total number of inputs).

Here, “expected” behavior means the variant’s outputs agree with the gold version’s outputs for the same input case. Thus each node in the debugging graph can be labeled with the empirically determined reliability of its corresponding variant. To motivate later

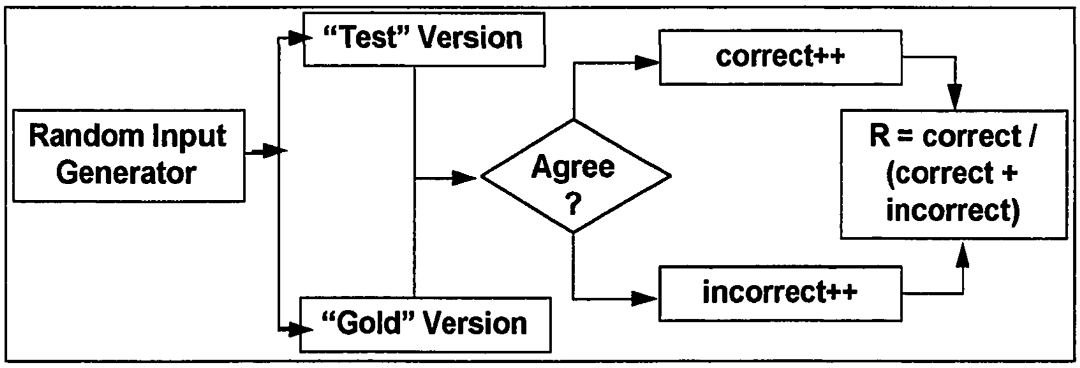

Figure 5. Empirical Reliability Calculation Procedure

notational conventions, we will refer to the error detector constructed using the gold version program as the ***gold oracle***.

3.2.2 Subject Software Details

We set out to construct a debugging graph for software developed under realistic, although controlled laboratory, conditions. We identified a body of code commonly called the ***Lauch Interceptor Condition (LIC)*** application. This software has been the subject of prior NASA-sponsored research projects [11, 12, 25, 27] as well as some preliminary investigations of the debugging graph [10, 31].

3.2.2.1 Launch Interceptor Condition (LIC)

LIC simulates part of a radar tracking system that generates a launch interceptor signal based on input tracking coordinates. The interceptor launch key is normally considered locked. Input parameters determine which combinations of 15 individual launch interceptor conditions are relevant and which are satisfied. Only if all relevant conditions are met will the unlocking signal be issued [11]. LIC exists in three independently developed FORTRAN code versions of between 400 and 600 SLOC in length created by experienced, professional programmers. There are documented debugging histories for each version as well as a gold version whose reliability is assumed to be perfect.

3.2.2.2 Control Program Development

The LIC software was ported from the NASA Langley Research Center's AIRLAB facility in Hampton, Virginia to the ODU Sun network. A new testing environment was set up to eliminate features of the LIC instrumentation which had been added in previous years to satisfy experimental criteria deemed orthogonal to our interests, and to allow exploitation of new operating system features for execution time speedup.

A simplified view of the so-called control program logic, **LICCtrl**, is shown in Figure 6. The control program in essence directs the generation of a large, random but repeatable input stream, performs the empirical R calculation described above and records and tallies results. The program is designed to run two separate subroutines: the *gold* version LIC solution; and a selected LIC *test* variant containing some combination of known bugs and repairs. On each iteration, both subroutines are exercised using the same input values. The subroutines' output values can easily be compared for equality. Output

```
read runtime parameters;
for( desired number of runs ) {
        generate next set of input values;
        load common block;
        call gold subroutine;
        load common block;
        call test subroutine;
        compare gold and test results;
        tally & record;
}
output summary statistics;
exit(0);
```

Figure 6. Overall LICCtrl Program Logic

agreement corresponds to a successful or "expected" outcome; while output disagreement and/or abnormal terminations of the subroutines denote a failing or "unexpected" outcome. Results are recorded and tallied over the total number of runs specified. Fail sets and output data can also be recorded. The **LICCtrl** implementation is discussed in more detail in Appendices A and B.

By using the same random input stream for each gold-test pair, empirical reliability figures and fail sets can be calculated for each debugging graph node. For this preliminary experiment, we chose a LIC variant with 12 known bugs. We found that two of the bugs were actually artifacts of the previous LIC test environment and for that reason were not interesting to study. So those repairs were installed prior to collecting any data, leaving 10 known faults of interest for debugging graph analysis. We constructed a debugging graph for $n = 10$ using 1 million input cases to estimate R for each of the 1024 nodes, in addition to capturing fail sets and erroneous output cases for all variants.

3.2.2.3 Test Environment Design

It is probably apparent at this point that the debugging graph testbed (intentionally) produced a substantial amount of data which, without adequate preparation and forethought, might overwhelm the analysis component of the experiments. The organization of this data was fundamental to enabling its location and access to become relatively minor aspects of these and future investigations.

We designed a UNIX directory structure which mirrored the debugging graph structure and enabled easy navigation through the "graph." This consisted of a minimal collection of "hard" directories and subdirectories, as well as "soft links" which made the directory structure react as though it were completely connected. Special directories were created to contain universally accessed shell scripts and code fragments. Automated scripts were developed to create not only the directory structure, but also to assemble, compile, link and run each test variant, and to later examine the contents of files containing the collected data. To optimize execution speed, runs were distributed across the ODU Sun network to processors which had their own local disk. A high "nice" level and "off" computing hours were used to minimize disruption to other system clients. The test environment configuration is described in Appendix C.

3.2.2.4 Data Collection Component

In its initial FORTRAN implementation as ported from NASA, **LICCtrl** required approximately 30 wall-clock hours to complete a single, 100,000 trial run. Considering that 1024 nodes were needed for the LIC debugging graph, and 1 million trials were considered minimal, this was unacceptably slow. Appendix A describes how we reduced the **LICCtrl** runtime to approximately four wall-clock hours per node, thereby supporting a massive data collection effort in a reasonable amount of time. We include this information to suggest ways in which similar debugging graph data collection environments could be set up for other software specimens.

3.3 Conclusions

In subsequent chapters, we describe various experiments using data derived from the debugging graph database. The experiments were designed to address various issues in software reliability prediction as cited in the previous chapter. For purposes of this laboratory work, a complete debugging graph was constructed for the subject software and we extensively explore and analyze its data. The reader should note, however, that a primary goal of the later experiments is controlling computational complexity. Chapter 8 describes one technique for doing so, while Chapter 9 discusses a methodology based on the partial debugging graph whose expense is polynomial in the worst case.

Chapter Four

Fault Sizing

In planning an experiment to study the effects of various fault recovery orders (see Chapter 6), we first needed to establish criteria for selecting interesting debugging paths to examine. We formulated path construction criteria (enumerated in Table 9) based on various fault "size" arrangements (e.g., largest-to-smallest, smallest-to-largest, etc.). This chapter describes our efforts to consistently "size" known faults, a necessary preprequisite to conducting the later experiment.

4.1 Experiment Description

We applied a criterion used in earlier experiments which estimates relative fault sizes according to their observed failure rates (i.e., estimated fail set size) [10]. This was accomplished by using data from fixed levels in the debugging graph in a procedure we call ***static relative size ranking***.

Table 1. Static Relative Size Rankings at Levels 1 and 9

Level 1		*Level 9*	
Reliability	***Bug Ordering***	***Bug Ordering***	***Reliability***
94.9859	1	1	45.3771
44.1205	2	2	97.2267
43.2728	5	4	98.8365
43.1036	4	5	98.9915
42.8039	3	3	99.8519
42.6722	6	6	99.9705
42.6674	8	8	99.9987
42.6672	10	9	99.9990
42.6671	7	10	99.9990
42.6671	9	7	99.9991

4.1.1 Static Relative Size Ranking at Levels 1 and 9

As shown in Table 1, we used data from level 1 in the debugging graph to arrange the faults in order of non-increasing size based on their failure rates. The rates were inferred based on the R values associated with each of the program variants containing a single repair, with a largest-to-smallest fault size order corresponding to a non-increasing sequence of the variants' R values. We avoided any special handling when the R values are equal, such as bugs 7 and 9, and simply used the original debugging order in such cases (e.g., 7, 9).

To validate the level 1 ordering, we repeated this process using the level 9 program variants, this time achieving a largest-to-smallest fault size order with a non-decreasing sequence of their R values. As shown in the highlighted region of the table body, the ordinal placement of five of the bugs could change depending on the level at which they were considered.

Table 2. Delta Reliabilities at Levels 1 and 9

Level 1 $R(P_i) - R(P)$	*Repair* i	*Level 9* $R(P_{1...10}) - R(P_{\{1...10\}-\{i\}})$
51.3188	1	54.6220
1.4534	2	2.7724
0.1368	3	0.1472
0.4365	4	1.1626
0.6057	5	1.0076
0.0051	6	0.0286
0.0000	7	0.0000
0.0003	8	0.0004
0.0000	9	0.0001
0.0001	10	0.0001

4.1.2 Delta R Study at Levels 1 and 9

To investigate this interesting result further, Table 2 illustrates the reliability changes, or ***deltas***, realized by installing each of the ten known repairs in the initial program P (level 1) versus removing only one of the known repairs from $P_{1...10}$ (level 9). At level 1, the observed failure behavior for a given fault — inferred by observing the effects of installing its repair — is subject to the influence of all other faults, *both known and unknown*, in the program. At level 9, the observed failure behavior for a given fault — inferred by observing the effects of removing its repair — *is not* subject to the influence of other *known* faults, but *is* influenced by *as yet undiscovered* faults in the program as well as the other installed repairs. Based on Table 2, the only faults which appear to behave identically at both levels are 7 and 10, with 8 and 9 exhibiting not much difference. This was surprising; we had expected that the overall size rankings would be the same at the two levels, with perhaps some minor differences in the specific deltas.

A further unexpected outcome of this analysis was the *magnitude* of the differences in several cases. Intuitively, we expected that installing a repair in the (as yet) unrepaired program would result in an analogous delta as instead installing that same repair at some later time during the debugging process. This is the situation we simulated by looking at the two debugging graph levels; but the observed outcome did not meet our intuitive expectations. As examples, Table 2 shows that the repairs numbered 2, 4 and 6 exhibit reliability changes at level 9 which are respectively about two, three and five times those which occur at level 1. So for the subject software, we are getting far less reliability improvement when any of these repairs is installed in the presence of all other faults, both known and unknown, than when we install that repair in the presence of only the unknown faults.

4.1.3 Static Relative Size Ranking at Levels 2 and 8

We repeated static relative size ranking at two more graph levels to verify that the conflicting rankings observed at levels 1 and 9 were not just exceptional cases. We assumed that during the course of a normal debugging session bug 1 would most likely be discovered and repaired first, corresponding to its overwhelming ranking as the "largest" of the known faults. We examined the reliability figures of appropriate variants in the debugging graph database to determine the relative sizes of the nine remaining faults at level 2, given that the repair associated with bug 1 had already been installed. These results are presented together with the level 1 data in Table 3.

Table 3. Static Relative Size Rankings at Levels 1 and 2

Level 1		***Level 2, with Repair 1 installed***	
Reliability	***Bug Ordering***	***Bug Ordering***	***Reliability***
94.9859	1	▨	▨
44.1205	2	2	97.6960
43.2728	5	4	96.0921
43.1036	4	5	95.9302
42.8039	3	3	95.1160
42.6722	6	6	94.9966
42.6674	8	8	94.9862
42.6672	10	10	94.9860
42.6671	7	7	94.9859
42.6671	9	9	94.9859

This time only the size rankings of bugs 4 and 5 were reversed between the two levels. Still, this indicated that at least some of the remaining known faults changed relative size ranks. We inferred from this that the absence of bug 1 — that is, the installation of its repair — has side effects on the failure behavior associated with bugs 4 and 5, since their failure behavior changed between the two adjacent levels of the debugging graph.

Next, we juxtaposed size rankings between levels 8 and 9. As previously explained, the level 9 data cites the reliability figures of the variants containing repairs for all but one known fault. At level 8, we looked at variants having all but two known faults repaired, one of them bug 1. The comparison figures are shown in Table 4. From these data it appears that the presence of bug 1 — that is, the absence of its repair — has side effects on the failure behavior associated with bugs 7, 9 and 10 as well as bugs 4 and 5, all

Table 4. Static Relative Size Rankings at Levels 8 and 9

Level 8, with Bug 1 installed		***Level 9***	
Reliability	***Bug Ordering***	***Bug Ordering***	***Reliability***
▨	▨	1	45.3771
43.8760	2	2	97.2267
44.7166	5	4	98.8365
44.9167	4	5	98.9915
45.2299	3	3	99.8519
45.3641	6	6	99.9705
45.3767	8	8	99.9987
45.3770	10	9	99.9990
45.3771	7	10	99.9990
45.3772	9	7	99.9991

Table 5. Some Delta Reliabilities at Levels 2 and 8

Level 2 $R(P_{1,i}) - R(P_1)$	***Repair*** i	***Level 8*** $R(P_{2...10}) - R(P_{\{2...10\}-\{i\}})$
▨	1	▨
2.7101	2	1.5011
0.1301	3	0.0781
1.1062	4	0.4604
0.9443	5	0.6605
0.0107	6	0.0130
0.0000	7	0.0000
0.0003	8	0.0004
0.0000	9	0.0001
0.0001	10	0.0001

of whose relative rankings potentially changed at level 8 from those observed at the adjacent level 9.

4.1.4 Delta R Study at Levels 2 and 8

In a manner analogous to the level 1 and 9 comparisons, Table 5 presents the deltas realized by installing two repairs at a time in the initial program P — specifically, the repair for bug 1 along with each of the remaining repairs (level 2) — versus removing two repairs — the repair for bug 1 along with each of the other repairs (level 8). We expected to see similar relative rankings and magnitudes of these values at the inspected levels of the debugging graph. While our expectations about relative size rankings were fulfilled — evaluation at levels 2 and 8 ordered the bugs identically — the magnitudes of the deltas were still quite different in several cases. In particular, bugs 2, 3 and 4 were measured as roughly twice as large at level 2 versus level 8.

4.1.5 Relative Size Determinations

At this point we have determined four different relative sizes and corresponding rankings for the bugs at each of the debugging graph levels 1, 2, 8 and 9. These data are coalesced in Table 6. Shown this way, it is easy to see that the size rankings are slightly different at the four levels. Further, although bugs 7 and 10 are alone in being assigned the same "absolute" sizes at all four levels, nearly half the bugs — namely 1, 2, 3 and 6 —

Table 6. Some Bug Sizes

bug	*Level 1*	*Rank*	*Level 2*	*Rank*	*Level 8*	*Rank*	*Level 9*	*Rank*
1	51.3188	1	▨	1	▨	1	54.6220	1
2	1.4534	2	2.7101	2	1.5011	2	2.7724	2
3	0.1368	5	0.1301	5	0.0781	5	0.1472	5
4	0.4365	4	1.1062	3	0.4604	4	1.1626	3
5	0.6057	3	0.9443	4	0.6605	3	1.0076	4
6	0.0051	6	0.0107	6	0.0130	6	0.0286	6
7	0.0000	9	0.0000	9	0.0000	10	0.0000	10
8	0.0003	7	0.0003	7	0.0004	7	0.0004	7
9	0.0000	10	0.0000	10	0.0001	9	0.0001	8
10	0.0001	8	0.0001	8	0.0001	8	0.0001	9

appear in the same relative ranks at all four levels despite rather large differences in the magnitude of their associated deltas in each case.

The varying measurement of bug size at different levels of the debugging graph impacts our experiments in the following sense: having chosen a particular size-based fault recovery criterion to construct a path through the graph, the specific order in which the faults are removed will vary depending on which graph level was used to make the relative size determinations. A question is whether predictive accuracy would be skewed enough to affect our evaluation conclusions by differences in the exact R values in a given type of path's sequence (i.e., largest-to-smallest, smallest-to-largest, and so on). In subsequent experiments, we will address this concern by constructing paths using two size orderings — level 1 and level 9 — to conduct our studies.

4.1.6 Path Inspections

Finally, we inspected the paths through the debugging graph that would be followed by iteratively installing repairs using the largest-to-smallest size rankings of the faults as calculated at both levels 1 and 9. For each path, this determined a sequence of reliability figures associated with program variants resulting from iteratively inserting into the software the repairs corresponding to the size-ordered faults. Two paths resulted from each of the two fault orders, because we considered inserting in both possible orders the repairs for fault pairs whose corresponding program variants have equal reliability figures (i.e., bugs 7 and 9 at level 1, and bugs 9 and 10 at level 9).

The data are shown in Figure 7. We note that *none* of these paths produces a sequence of reliability figures that exhibits no negative reliability growth. It is interesting that omitting the repairs for bugs 7 and 9, when they appear as the last two installations in any of these paths, actually results in a higher overall reliability figure than installing all the known repairs. We find this to be an interesting pathology, since all the known repairs are presumed to be correct and we therefore expect the insertion of any one to produce at least some R growth.

4.2 Analysis

In the past, such anomalous behaviors as those pointed out above have been attributed to partial or incorrectly done repairs. But the LIC repairs are presumed to be the "correct" ones, having been validated in previous experiments and checked again in

Level 1-Based Paths
Initial R: 42.6671

1	2	5	4	3	6	8	10	7	9
94.9859	97.6960	98.6781	99.8228	99.9700	99.9987	99.9991	99.9992	99.9990	99.9991

1	2	5	4	3	6	8	10	9	7
94.9859	97.6960	98.6781	99.8228	99.9700	99.9987	99.9991	99.9992	99.9991	99.9991

Level 9-Based Paths
Initial R: 42.6671

1	2	4	5	3	6	8	9	10	7
94.9859	97.6960	98.8260	99.8228	99.9700	99.9987	99.9991	99.9990	99.9990	99.9991

1	2	4	5	3	6	8	10	9	7
94.9859	97.6960	98.8260	99.8228	99.9700	99.9987	99.9991	99.9992	99.9991	99.9991

Figure 7. Largest-to-Smallest Paths

this one. So the inconsistencies in these data lead us to alternative conclusions. Individual faults do appear to fail with different rates [25]; but the standard view that the failure rate is somewhat a "given" characteristic for each fault within a particular program is perhaps misleading. We find that the rate experimentally associated with a given fault can be subject to the program's ***debugging state*** at the time the assessment is made. We believe this phenomenon is a physical manifestation of interaction effects that needs to be taken into account during software reliability modeling.

Why should fault sizing anomalies be a concern? Besides complicating our efforts to construct debugging paths based on a size criterion, this behavior means that even when considering a repair which is known to be "correct," one cannot always expect it to have the same reliability growth effect, since that effect depends on *when* the repair is installed during the debugging process. As we noted above, there may even be a subset of the known repairs which, when installed, results in a higher reliability figure than installing all known repairs. The obvious question is how existing models can produce accurate predictions while remaining "fuzzy enough" to account for such variance when only one realization of the debugging process is used to make predictions — and we have no assurance as to which of the $n!$ possible paths will be presented as input.

4.3 Conclusions

Examination of various levels of the debugging graph and attempts to "size-rank" the software's known faults revealed that not only do individual faults fail with different rates, but the rate experimentally associated with a given fault is a function of the program's debugging state at the time assessment is made. We attribute this effect to interactions among faults, both known and unknown, which may result in unexpected changes in the failure behavior of the considered program variants. We conjecture that these effects occur dynamically during the debugging process and explore this assumption further in the next chapter.

Chapter Five

Fault Interactions

In the previous chapter, we explained how to relatively size rank faults based on the changes in reliability caused by installing single faults and/or single repairs in isolation. We found that size ranking, while seemingly straightforward, was complicated by unexpected inconsistencies in these changes, which we attributed to the debugging state of the software at the time the analysis was performed. Further experimentation to study this phenomenon certainly seems warranted, since there is documented interest in so-called fault interactions elsewhere in the research literature.

As a starting point for future work in this area, we present some preliminary ways that we examined suspected fault interactions more closely in the context of the debugging graph database. To construct the most probable debugging path, we tried to construct a path using a greedy, largest-remaining-bug-next criterion which dynamically re-evaluated relative fault sizes. We subsequently performed some detailed investigations of the bug pairs 7, 9 and 9, 10 since their anomalous behaviors during size rankings in this and the previous chapter lead us to believe they are interacting in some manner.

5.1 Dynamic Relative Size Ranking

It seems apparent from the previous chapters that the behavior of a fault may vary according to which other faults and repairs are present in the software. This observation, as summarized in 4.1.5, implies that each fault has the potential to appear larger or smaller depending on the program's debugging state when the size assessment is made. We conjecture that this is due to interaction effects. As an alternative to relying on the static relative size ranking approach at a fixed level in the debugging graph, we used our database derived from the ten bugs of interest to incrementally step through the debugging graph. At each level, we chose a program variant that resulted in the maximum positive growth in R upon the installation of any one of the remaining repairs. We then added that bug's number to the path — a "greedy" path construction algorithm.

This process corresponds to iteratively removing the i^{th} bug with the greatest failure rate, with that assessment performed at debugging graph level i. Since the relative size rankings of the remaining bugs were in this manner recalculated with each iteration — a process we call ***dynamic relative size ranking*** — we conjectured the greedy algorithm could produce a path that would result in more accurate predictions from the models, since it is the most probable path.

5.1.1 The "Greedy" Path

We encountered difficulty in carrying out the seemingly straightforward greedy path construction approach. The algorithm proceeded smoothly through the first eight iterations, adding fixes 1, 2, 4, 3, 5, 6, 8 and 10 to the path. Then only two known bugs remained — 7 and 9 — and installing neither repair would result in positive reliability growth. This process is illustrated in Figure 8.

5.1.2 The "Not-So-Greedy" Path

As an alternative to the greedy path algorithm, which we were unable to execute to successful completion, we used the database derived from ten bugs of interest to explore the debugging graph in search of a path that at least exhibited no negative reliability growth—that is, a ***non-decreasing R path***. The algorithm basically followed a "not-so-greedy" approach, using dynamic relative size ranking to add on each iteration the first repair which resulted in no negative reliability growth, until no such choices remained. Then, backtracking was pursued, trying different path alternatives at previous steps in the algorithm until a non-decreasing R path was successfully created, or no alternative sub-

R	*Next Fix*
42.6671	initial R
94.9859	1
97.6960	2
98.8260	4
98.9628	3
99.9700	5
99.9987	6
99.9991	8
99.9992	10

Next Fix Choices	
R	Next Fix
99.9990	7
99.9991	9

Figure 8. "Greedy" Path Attempt

paths remained, whichever occurred first.

We conjectured that two factors would help us to account for unforeseen effects of bug interactions at subsequent levels of the graph: providing for backtracking; and relaxing the requirement for positive R growth with each path increment—only *no negative growth* would be sought. Additionally, the non-decreasing R path would be a practical approximation of the theoretical view of MTTFs as increasing through time.

The non-decreasing R path algorithm in fact proceeded smoothly; the fourth path attempted using this algorithm was successful. The sequence in which repairs were inserted and the corresponding R values of the program variants along this path are shown in Figure 9. Although an exhaustive search was not conducted, further exploration of the debugging graph revealed that many other non-decreasing R paths exist.

In the remainder of this chapter we detail some deeper analysis of the bug pairs 7, 9 and 9, 10 whose failure behavior appear to be afflicted by interaction phenomena.

5.2 Bugs 7 and 9

First we consider the programatic nature of bugs 7 and 9. Bug 7 is related to the calculation of one bit of the ***conditions met matrix*** in LIC. The programmner used an incorrect formula to compute the distance between two points. Bug 9 is related to the calculation of another bit of the same matrix. In this case, the programmer had forgotten

Initial R: 42.6671

1	2	4	3	5	6	7	10	8	9
94.9859	97.6960	98.8260	98.9628	99.9700	99.9987	99.9987	99.9988	99.9990	99.9991

Figure 9. A "Not-So-Greedy" Path

to check for a special vector relationship.

Elements of the conditions met matrix are logically combined by LIC to determine a "launch" or "no launch" output according to a specific unlocking sequence, in which a particular combination of significant conditions as represented in the matrix is assessed. The logical connectors used in this calculation are part of the (random) input to the problem, so that either or both of the conditions represented by the repaired code fragments might not even have any effect on the output calculation, let alone each other. From this we conclude that, while both involve part of the matrix calculation, failures associated with bugs 7 and 9 are not logically linked.

The reliability figures associated with variants containing various combinations of bugs 7 and 9 and their repairs were examined for anomalies. Fail sets were also inspected for unexpected occurrences. The results of this analysis are explored in the next two sections.

5.2.1 Repair Anomalies

We noted from the debugging graph database that adding the repair for bug 7 at level 2 — that is, after repairing any one of the other remaining known bugs — produced no improvement in the respective reliability figures. That is, in general, $R(P_n) = R(P_{n,7})$ whenever n is chosen from $\{1,...,6,8,...10\}$. The effects of bug 7's repair were in effect hidden by any other single repair together with all the other faults, both known and unknown. The same relationship held when repairing bug 9 at level 2 in all but one case:

Table 7. Fail Sets Associated with Bugs 7 and 9

Known Bugs Present			
Bug 7	***Bug 9***	***Bugs 7 & 9***	***None***
57636	57636	57636	57636
63454	63454	63454	63454
189981	189981	189981	189981
191909	191909	191909	191909
237569	237569		237569
792841	792841	792841	792841
823542	823542	823542	823542
	839343		
898202	898202	898202	898202
900157	900157	900157	900157

repairing bug 9 after bug 6 resulted in negative R growth. Hence, we deduce that bug 9's repair "broke" certain input cases that had previously (appeared to) perform correctly.

5.2.2 Fail Set Study

At levels 8 and 9, we used the debugging graph database to determine the specific random input case numbers on which failures occurred (i.e., disagreement between the outputs of the oracle and the variant under test). As mentioned in the background material, such collections are commonly called fail sets. Table 7 shows some interesting failure anomalies, and leads us to offer some conjectures:

- The software has precisely the same fail set regardless of whether bug 7 is repaired, in the presence of all remaining repairs (i.e., the leftmost and rightmost columns in the table are identical). This may indicate interactions of bug 7 with as yet undiscovered faults in the program, producing "coincidentally correct" results for some of these input cases.

- Installing bugs 7 and 9 simultaneously in the presence of all remaining repairs produces a fail set that is smaller than the fail set associated with either bug 7 or bug 9 individually. In fact, the installation of bug 7 appeared to repair two of bug 9's bad cases (237569 and 839343). This may indicate that the two bugs compensate for each other and/or interact with other faults which remain to be found.

5.3 Bugs 9 and 10

Turning to the second pair of bugs we investigated, we note once again that bug 9 is related to the calculation of one bit of the conditions met matrix (see 5.2). Bug 10 occurred because the programmer had incorrectly specified a value of 3.1414927 for the parameter PI; the correct value is 3.1415927. There is a logical relationship between these bugs, however, since the portion of the code involving bug 9 uses PI in its calculation. On the surface, this leads us to suspect that even if bug 9 is repaired, whenever bug 10 is still present at least some of the calculations involving PI will be significantly enough incorrect to observe as failures. Similar analysis as that described above was conducted for bugs 9 and 10 to see if some of the same observed effects manifested themselves, or if the logical relationship of this bug pair made it distinctly different from the former case.

5.3.1 Repair Anomalies

First, recall we previously noted that adding the repair for bug 9 after repairing any one of the remaining known bugs produced no change in the respective reliability figures

Table 8. Fail Sets Associated with Bug 9 and Bug 10

Known Bugs Present			
Bug 9	***Bug 10***	***Bugs 9 & 10***	***None***
57636	57636	57636	57636
63454	63454	63454	63454
189981	189981	189981	189981
191909	191909	191909	191909
237569	237569		237569
792841	792841	792841	792841
823542	823542	823542	823542
839343	839343	839343	
898202	898202	898202	898202
900157	900157	900157	900157

at level 2, in all but one case: repairing bug 9 after bug 6 produced negative reliability growth. By contrast, installing the repair for bug 10 after any one of the remaining known bugs produced equal growth in all cases; specifically, $R(P_{n,10}) - R(P_n) = 0.0001\%$ whenever n is chosen from $\{1,...,9\}$. This is consistent with our expectations; at least some of the failures present at both levels 1 and 2 may be attributable to PI having the wrong value, while repairing the value produces consistent improvement.

5.3.2 Fail Set Study

At levels 8 and 9, we again used the debugging database to determine the specific random input case numbers on which failures occurred, as shown in Table 8. Since bugs 9 and 10 are logically related in the code, we saw interesting interactions taking place, and we conjecture the following:

- In the presence of all other known repairs, the software has precisely the same fail set regardless of whether bug 9 or bug 10 is present, (i.e., the two leftmost columns are identical in the table). This may indicate that the repairs must be installed simultaneously for either to show positive effect.

- In the presence of all other known repairs, when both bugs 9 and 10 are simultaneously present, one input case (237569) appears to be repaired which had previously failed when either bug was individually present. This may indicate compensation between the two bugs, or perhaps interactions of these bugs with as yet undiscovered faults in the code.
- In the presence of all other known repairs, installing the repairs for bugs 9 and 10 — as shown in the column labeled "None" — once again "broke" the software for the previously mentioned input case (237569); but it repaired another case (839343) which had failed for the other fault/repair combinations considered. Installing the repairs simultaneously, "as they should be," may be revealing that the anomalous input case (237569) produced the correct output only coincidentally in the other program variants considered.

5.4 Analysis

We observed some interesting manifestations of fault interactions, with unexpected positive and negative effects. Sometimes, the removal of a repair (bug installation) has no effect, or even improves, the software's observed behavior, given a particular combination of other repair and/or bug installations. The presence of the bug, in effect, repairs or compensates for the erroneous behavior associated with certain inputs for another.

Viewed another way, the insertion of a given repair (bug removal) can make no difference, or even degrade, the software's empirically determined reliability along a given path — despite the perceived size of the bug it repairs. We could say that a bug whose presence improves performance (i.e., a repair whose installation degrades performance) behaves as though it has "negative" size, since repairing it produces the opposite of the expected effect.

Unexpected behaviors in the R values observed in the debugging graph provides a hint of interaction potential. The ability to use the debugging graph data collection environment to inspect fail sets enables us to penetrate to a deeper level of analysis than simply looking at raw reliability numbers. We looked at two pairs of faults we thought were interacting; bugs 7 and 9 did not appear to be explicitly related in the code, while bugs 9 and 10 did. Both exhibited anomalies in observed R values at specific level of the debugging graph, as well as in the fail sets associated with various fault/repair combinations.

We advanced some conjectures about what the anomalies observed in the R values and the fail sets might imply about these faults' interaction potential; unfortunately, our ability to make stronger conjectures based on the LIC experiment is complicated by the fact that as yet undiagnosed faults remain in the code. There may be other interacting faults in the LIC software. It is unclear at the present time how best to identify them, although some of the above observations might offer starting points for future work.

We found it significant that the "greedy" path — which was constructed to near completion according to the maximum R growth criterion and dynamic re-evaluation of relative bug size — and the "not-so-greedy" path both ranked the faults in yet different orders than those produced by static relative size ranking at level 1 and level 9. We believe this observation to be symptomatic of the effects of bug interactions on the empirically determined failure rates. Further, subject to the effects of data accuracy and bug interactions, clearly it is not always be possible to construct a "greedy" path that reflects maximum positive reliability growth between adjacent graph levels of the debugging graph; it was not successful and we will therefore not include such a path in the experiments subsequently described in this thesis.

5.5 Conclusions

Software reliability modeling is an abstraction of the failure process in that a sequence of MTTFs (and the associated sequence of R values) is used for predictions, with no inspection of the underlying fail sets. Zero, low or negative reliability growth, perhaps attributable to interactions, is not explicitly accounted for in existing models; yet it is unclear that any of them are "fuzzy" enough to implicitly account for some of the unexpected behaviors we observed in the LIC failure data, behaviors attributed not just to inherent randomness in the data but also to fault interactions.

Contradictory results when inspecting fail sets when the bugs are considered both in isolation or in the presence of some or all of the other bugs, both known and unknown, should probably not be too surprising, considering earlier indications of the apparent effects of bug interactions. During the early to middle stages of the debugging process, a possibly significant number of bugs remain unrepaired and may influence our attempts to capture the failure behavior of known faults. Similarly, in many cases a perfect end product is not an achievable goal, so that some faults will always remain undiscovered in the tested software. The debugging graph, constructed using an appropriately instrumented control program, offers many opportunities for further study of interaction phenomena.

Some researchers have claimed that interaction effects are negligible, and that in most cases faults which mask one another would have been discovered and eliminated in the earlier software development stages [22]. The analysis presented in this chapter points out that this is not necessarily the case; perhaps further work is needed in this area. At least one remaining challenge is determining whether the debugging graph model offers a means for distinguishing how logically related and unrelated faults/repairs can be represented to the prediction process to better account for their side effects on empirically observed reliability changes. We note that the methodologies described later in this thesis mitigate interaction effects by controlling debugging order.

Chapter Six

Fault Recovery Order

Since intuitively the debugging process is most likely to recover bugs in a largest-to-smallest order, we conjectured that recovering the faults in various "size" orders would yield an appropriate basis for comparing the models' predictive performance.

6.1 Experiment Description

The debugging graph was used as a basis for formulating input data for the predictive models along selected debugging paths. The models' predictive performances were compared along these paths.

6.1.1 Path Selection Criteria

For model comparison purposes, we formulated fault recovery criteria as enumerated in Table 9 to simulate various debugging sessions. As noted in 4.1.1, the size rankings of the bugs differed at the two graph levels we used to analyze them, thus, where path pairs are specified in the table, the second path number and the alternative graph level used for size ranking are shown in parentheses.

Table 9. Description of Debugging Paths

Paths	***Construction Method***
1 (3)	largest-to-smallest order using level 1 (9) size rankings
2 (4)	smallest-to-largest order using level 1 (9) size rankings
5	five largest repairs in non-increasing size order using level 1 size rankings, followed by the remaining five repairs in non-increasing size order using level 9 size rankings
6	five smallest repairs in non-decreasing size order using level 1 size rankings, followed by the remaining five repairs in non-decreasing size order using level 9 size rankings
7 (9)	alternate the largest remaining repair followed by the smallest remaining repair using level 1 (9) size rankings
8 (10)	alternate the smallest remaining repair followed by the largest remaining repair using level 1 (9) size rankings
11 (12)	medium, small and large repairs in mixed order using level 1 (9) size rankings
13	original repair order (1, 2, 3, etc.)

Paths using largest-to-smallest construction criteria (e.g., 1, 3, 5) are considered ***intuitive path*** examples; whereas those using a smallest-to-largest approach (e.g., 2, 4, 6) are considered ***counter-intuitive paths***. Paths 7 through 10 were included to stress the predictive models by making the incremental reliability improvements oscillate between relatively large and small changes. Paths 11 and 12 recover faults in mixed size orders, while path 13 represents faults repaired in the original recovery order.

6.1.2 Comparison Path Data

Table 10 enumerates the debugging paths of interest based on the fault recovery criteria discussed in 6.1.1. The paths were constructed by reading the corresponding R values from the debugging graph based on the gold oracle, whose construction was described in 3.2.

6.1.3 Comparing Models' Performance

The four models of interest — Jelinski-Moranda, Geometric De-Eutrophication, Basic Musa and Logarithmic Poisson — were implemented in the C programming language and executed on Sun SparcStations. The basic formulae and implementation validations are discussed in Appendix D. The comparison procedure explained in 6.1.3.1 and 6.1.3.2 is illustrated in Figure 10.

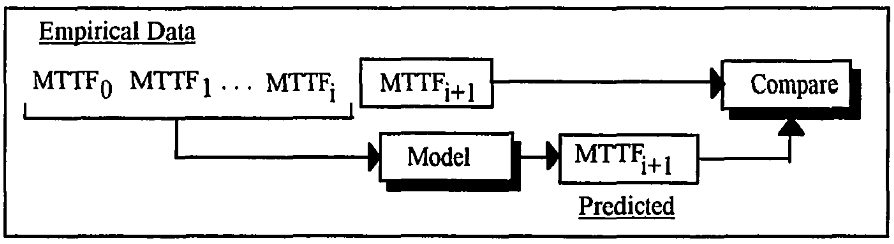

Figure 10. MTTF Comparison Procedure

Table 10. Repair Numbers and R Values for Comparison Paths (Gold Oracle)

Initial R: 42.6671

Path 1									
1 94.9859	2 97.6960	5 98.6781	4 99.8228	3 99.9700	6 99.9987	8 99.9991	10 99.9992	7 99.9990	9 99.9991
Path 2									
9 42.6671	7 42.6671	10 42.6672	8 42.6675	6 42.6725	3 42.8093	4 43.2533	5 43.8760	2 45.3771	1 99.9991
Path 3									
1 94.9859	2 97.6960	4 98.8260	5 99.8228	3 99.9700	6 99.9987	8 99.9991	10 99.9992	9 99.9991	7 99.9991
Path 4									
7 42.6671	9 42.6671	10 42.6672	8 42.6675	6 42.6725	3 42.8093	5 43.4255	4 43.8760	2 45.3771	1 99.9991
Path 5									
1 94.9859	2 97.6960	5 98.6781	4 99.8228	3 99.9700	6 99.9987	8 99.9991	10 99.9992	9 99.9991	7 99.9991
Path 6									
9 42.6671	7 42.6671	10 42.6672	8 42.6675	6 42.6725	3 42.8093	5 43.4255	4 43.8760	2 45.3771	1 99.9991
Path 7									
1 94.9859	9 94.9859	2 97.6960	7 97.6960	5 98.6781	10 98.6782	4 99.8229	8 99.8233	3 99.9705	6 99.9991
Path 8									
9 42.6671	1 94.9859	7 94.9859	2 97.6960	10 97.6961	5 98.6782	8 98.6786	4 99.8233	6 99.8519	3 99.9991
Path 9									
1 94.9859	7 94.9859	2 97.6960	9 97.6960	4 98.8260	10 98.8261	5 99.8229	8 99.8233	3 99.9705	6 99.9991
Path 10									
7 42.6671	1 94.9859	9 94.9859	2 97.6960	10 97.6961	4 98.8261	8 98.8264	5 99.8233	6 99.9705	3 99.9991
Path 11									
3 42.8039	6 42.8090	4 43.2530	8 43.2533	5 43.8760	10 43.8761	2 45.3772	7 45.3772	1 99.9990	9 99.9991
Path 12									
3 42.8039	6 42.8090	5 42.8093	8 43.4255	4 43.8760	10 43.8761	2 45.3772	9 45.3771	1 99.9991	7 99.9991
Path 13									
1 94.9859	2 97.6960	3 97.8328	4 98.9628	5 99.9700	6 99.9987	7 99.9987	8 99.9991	9 99.9990	10 99.9991

6.1.3.1 Iterative Prediction Process

We used the relationship MTTF = 1 / (1 - R) to generate input sequences for each path of interest using the empirically calculated R values for its variants as shown in Table 10. These sequences were input to each model, so that along a chosen debugging path, the experimentally generated consecutive MTTFs enumerated 0 through *i* were used to predict the $(i+1)^{st}$ failure time for each *i* from 1 to 10. (Note: The 0^{th} MTTF is derived using the R value for the unrepaired program, labeled in Table 10 as "*Initial **R***.") As an example, data for MTTF Prediction Stage 4 are based on using $MTTF_0$ through $MTTF_3$ as inputs to predict $MTTF_4$.

6.1.3.2 Normalized Comparison Data

As a measure of a model's predictive accuracy at each MTTF Prediction Stage of the iteration, we performed a normalized comparison by taking the ratio of each estimated MTTF as predicted by a model to the empirical MTTF calculated from an appropriate variant's R value (i.e., predicted MTTF / empirical MTTF). In a sense, the denominator of the comparison ratio serves as the true "reliability basis" against which the predicted values are compared, and we will so refer to it in the remainder of this thesis. Additionally, the resulting ratios, which fall in the range $(0, +\infty)$, were subjected to the $\mathbf{log_{10}}$ function to map them into the interval $(-\infty, +\infty)$.

The intuitive appeal of these converted ratios can be summarized as follows. A ratio close in value to one is coverted by $\mathbf{log_{10}}$ to a value close to zero and is interpreted as indicating an accurate prediction. Ratios greater than one are mapped to positive values by **$\log_{10}$**, and those between zero and one are mapped to negative values; they respectively indicate optimistic and pessimistic predictions. The ratio continuum and its interpretation are illustrated in Figure 11.

As an example, a predicted MTTF that is nearly an order of magnitude greater than the actual MTTF (i.e., the true reliability basis) produces a ratio value near ten, which **$\log_{10}$** converts to a value near one. This implies that the software would *probably fail* ten times (or one order of magitude) *sooner* than expected based on the estimated MTTF value; hence the model's prediction is optimistic.

The prediction ratios for the four models, as mapped into the interval $(-\infty, +\infty)$, are shown in Tables 11 through 14. To establish a consistent labeling convention that will make later discussions easier, we have added parenthetically to the graph titles the notation "(Gold / Gold)." This is meant to indicate that both the input values to the models and the denominator of the normalized comparison ratios, which serves as the true reliability basis, were derived from the table containing the gold oracle's assessment of the variants' reliability values (i.e., Table 10).

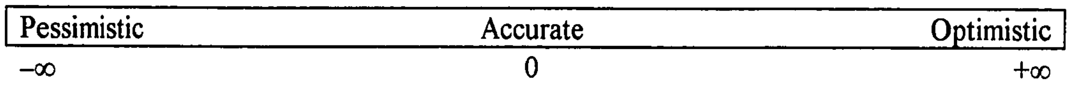

Figure 11. Predictive Performance Continuum
for $\log_{10}$(Estimated MTTF / Actual MTTF)

Table 11. Jelinski-Moranda Prediction Ratios (Gold / Gold)
? no solution; N is infinite or no R growth present
?? no solution; N is finite

Path	***MTTF Prediction Stage***								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	?	??	??	??	??	??	??	??	??
2	?	?	?	?	?	?	-6.577e-3	-0.01495	-4.791
3	?	??	??	??	??	??	??	??	??
4	?	?	?	?	?	?	-5.91e-3	-0.01447	-4.791
5	?	??	??	??	??	??	??	??	??
6	?	?	?	?	?	?	-5.910e-3	-0.01447	-4.791
7	?	?	??	??	??	0.6335	??	??	??
8	?	??	??	??	??	??	??	??	??
9	?	?	??	??	??	??	??	??	??
10	?	??	??	??	??	??	??	??	??
11	1.002e-3	-2.68e-3	5.512e-4	-3.328e-3	1.65e-4	-9.961e-3	-3.116e-3	-4.736	??
12	1.002e-3	7.03e-4	-4.166e-3	-3.957e-3	-1.125e-4	-0.01006	-3.11e-3	-4.782	??
13	?	??	??	??	??	??	??	??	??

Table 12. Geometric De-Eutrophication Prediction Ratios (Gold / Gold)

Path	***MTTF Prediction Stage***								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0.7205	0.6122	-0.1848	-0.2794	-0.8821	0.2363	0.8131	1.421	1.809
2	0	-8.686e-7	-2.171e-6	-3.822e-5	-1.05e-3	-3.828e-3	-6.584e-3	-0.015	-4.796
3	0.7205	0.5607	-0.1373	-0.2639	-0.8757	0.2385	0.8144	1.375	1.81
4	0	-8.686e-7	-2.171e-6	-3.822e-5	-1.05e-3	-5.148e-3	-5.921e-3	-0.01451	-4.796
5	0.7205	0.6122	-0.1848	-0.2794	-0.8821	0.2363	0.8131	1.375	1.81
6	0	-8.686e-7	-2.171e-6	-3.822e-5	-1.05e-3	-5.148e-3	-5.921e-3	-0.01451	-4.796
7	1.058	0.4452	0.4771	0.2191	0.3271	-0.5004	0.04236	-0.4995	-1.498
8	-1.058	0.4129	0.266	0.4891	0.3285	0.4445	-0.3728	0.01902	-1.934
9	1.058	0.4452	0.4771	0.1675	0.3109	-0.4469	0.06136	-0.486	-1.491
10	-1.058	0.4129	0.266	0.4891	0.2769	0.4197	-0.3283	-0.6629	-1.512
11	9.986e-4	-2.68e-3	5.373e-4	-3.342e-3	1.229e-4	-0.01001	-3.282e-3	-4.745	-0.2057
12	9.986e-4	7.026e-4	-4.167e-3	-3.973e-3	-1.616e-4	-0.01011	-3.287e-3	-4.77	-0.1552
13	0.7205	0.8269	0.3964	-1.058	-1.058	0.3514	0.7884	1.3433	1.732

Table 13. Basic Musa Prediction Ratios (Gold / Gold)
** indicates software predicted to be perfect*

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.6387	-0.7177	-0.2023	10.71	*	2.31	0.5342	-0.318	-0.7588
2	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8485	-0.9079	-0.966	-5.783
3	-0.6387	-0.7692	0.4839	10.54	*	2.31	0.5341	-0.2486	-0.6554
4	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8498	-0.9066	-0.966	-5.783
5	-0.6387	-0.7177	-0.2023	7.186	*	2.31	0.5342	-0.2486	-0.6554
6	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8498	-0.9066	-0.966	-5.783
7	-0.301	-0.8148	-0.5752	-0.9403	-0.6394	-1.718	0.6391	5.754	*
8	-1.359	-0.1545	-0.9398	-0.4044	-1.019	-0.5121	-0.4968	1.26	*
9	-0.301	-0.8148	-0.5752	-0.9918	-0.5429	-1.667	0.5925	5.586	*
10	-1.359	-0.1545	-0.9398	-0.4044	-1.07	-0.4112	0.19	12.52	*
11	-0.3011	-0.4805	-0.6021	-0.7038	-0.7782	-0.8569	-0.903	-5.692	2.777
12	-0.3011	-0.4771	-0.6068	-0.7024	-0.7782	-0.8569	-0.903	-5.737	2.341
13	-0.6387	-0.5034	-0.9221	10.22	*	1.213	0.5632	-0.1732	-0.5603

Table 14. Logarithmic Poisson Prediction Ratios (Gold / Gold)
? indicates no solution for desired precision

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.0777	-0.7139	-0.5969	-2.276e-3	0.07112	0.1829	0.1315	0.07551	7.594e-3
2	-0.301	-0.4771	-0.6021	?	-0.7792	?	-0.9079	-0.966	?
3	-0.0777	-0.7647	-0.4151	-3.892e-3	0.07037	0.1826	0.1311	0.07507	0.01262
4	-0.301	-0.4771	-0.6021	?	-0.7792	?	-0.9066	-0.96601	?
5	-0.0777	-0.7139	-0.5969	-2.276e-3	0.07112	0.1829	0.1315	0.07551	0.01318
6	-0.3010	-0.4771	-0.6021	?	-0.7792	?	-0.9066	-0.966	?
7	-0.02012	?	-0.5602	?	?	?	-0.2565	?	-0.19
8	-1.359	-0.04593	-0.3787	-0.2889	-1.0144	?	-0.8971	-0.2283	-0.2793
9	-0.02012	?	-0.5602	?	-0.5578	?	-0.2594	?	-0.1915
10	-1.359	-0.04593	-0.3787	-0.2889	-1.065	-0.4138	-0.7157	-0.2309	-0.1244
11	?	-0.4805	-0.6021	-0.7038	-0.7782	-0.8569	-0.9031	-5.605	?
12	?	-0.4771	?	-0.7024	-0.7782	-0.8569	-0.9031	?	?
13	-0.0777	-0.5017	?	-0.6354	0.07882	0.1863	0.1176	0.07434	0.01353

Each row in a table represents predictions along the debugging path cited in its left-most column. Six significant digits were carried in the calculations to reflect the precision of the MTTF values input to the models; converted ratio values were rounded to four significant digits to enhance the tables' readability. We marked exceptional cases with symbols and annotated them in the captions. It should be noted that, were a model predicting perfectly along a given path, the corresponding row of table entries would contain all zeros. Paths that result in pessimistic predictions contain predominately negative values; whereas predominately positive values indicate predictive optimism..

6.2 Analysis

Despite its assumption that "all bugs are created equal," — which often causes it to be dismissed an impractical for realistic applications — the Jelinski-Moranda model's predictions were initially quite good for the mixed recovery order paths (11, 12). Although it otherwise generally failed at prediction, the algorithm performed *consistently* over the intuitive paths, assessing a finite number of bugs after the first few iterations; whereas counter-intuitive paths proved to be more challenging, probably due to low-to-no reliability growth.

For several counter-intuitive paths (2, 4, 6), the Geometric De-Eutrophication algorithm provided the most accurate overall predictions, possibly since the effects of inconsistent changes were postponed in those cases. It also gave good predictions for mixed recovery order paths (11, 12) until the last two predictive stages, again possibly due to latent numerical effects from the largest fault. Along the remaining paths, the model appeared to try and correct itself at some point; but unfortunately the predictions tended to grow either increasingly optimistic or pessimistic thereafter.

The behavior of the Basic Musa model was *inconsistent* from path to path as well as along any given path. Its predictions were either very optimistic or pessimistic, and it often incorrectly predicted perfect software after an early, large reliability improvement step. Interestingly, however, the influence of the very large repair (i.e., the fix for bug 1) appeared to be mitigated if it were inserted early.

The primary challenge in using the Logarithmic Poisson model was determining parametric values which "fit" the functions, given the input data precision and host computer accuracy; this problem was particularly evident for counter-intuitive and mixed paths. However this model performed extremely well on the intuitive paths (1, 3, 5) — it was the only model which did so — and the overall performance on the original order path (13), unlike that of the other models, was quite respectable. Another general characteristic observed was close coincidence of portions of some paths with the Basic Musa model's corresponding predictions (e.g., paths 11 and 12).

The data derived from comparing the four reliability models clearly show that along a given debugging path, the predictive performance of these models can vary greatly. As an example, Figure 12 plots the comparison ratios along path 1 for the four models. We also note that, despite differences in specific prediction ratios at each stage, paths constructed using the same criteria (e.g., 1 and 3, 2 and 4), exhibit similar predictive accuracy overall for any given model. That is, the selection of a criterion for path

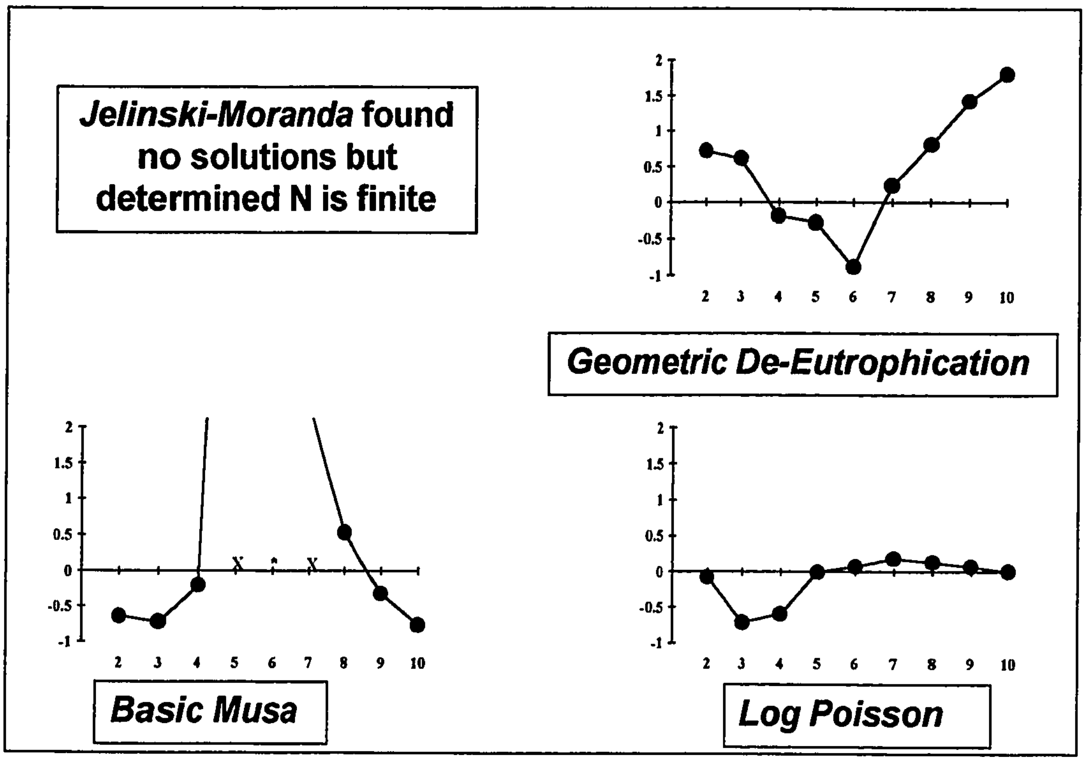

Figure 12. Prediction Ratios Compared Along a "Largest-to-Smallest" Path
X indicates "outlier" (value off scale)
** indicates software predicted to be perfect*

construction appears to be more important than the specific methodology used to assess the relative fault sizes.

Additionally, just because a model appears to "fit" a given path's data well in terms of its predictive performance, there is no guarantee that the model would still appear as appropriate had those faults been recovered in a different order. Figure 13 markedly contrasts the performance of the Geometric De-Eutrophication model along paths 1, 4 and 7. In other words, if the practitioner evaluates models based on a single realization of the debugging process, with the faults recovered and corrected in a single order, (s)he might reject a model that could perform quite well using data from a different recovery order.

6.3 Conclusions

Based on the data presented in this chapter, we conclude that the Basic Musa model would probably not perform well as a predictor for the LIC software. Although the Jelinski-Moranda and Geometric De-Eutrophication models performed extremely well along certain paths, this appears to be an artifact of the models expoiting some aberration in a particular path's data, such as the postponement of effects from a large repair. When presented with "largest-to-smallest" data which comply with our intuitive interpretation of the fault recovery process and the model's underlying assumptions, the Logarithmic Poisson model featured quite accurate predictions once several data points accumulated.

In this experiment we also observed that recovery order is an important factor in general which influences the potential performance of the tested software reliability models. Based on this work it appears that, when using the average of large samples for

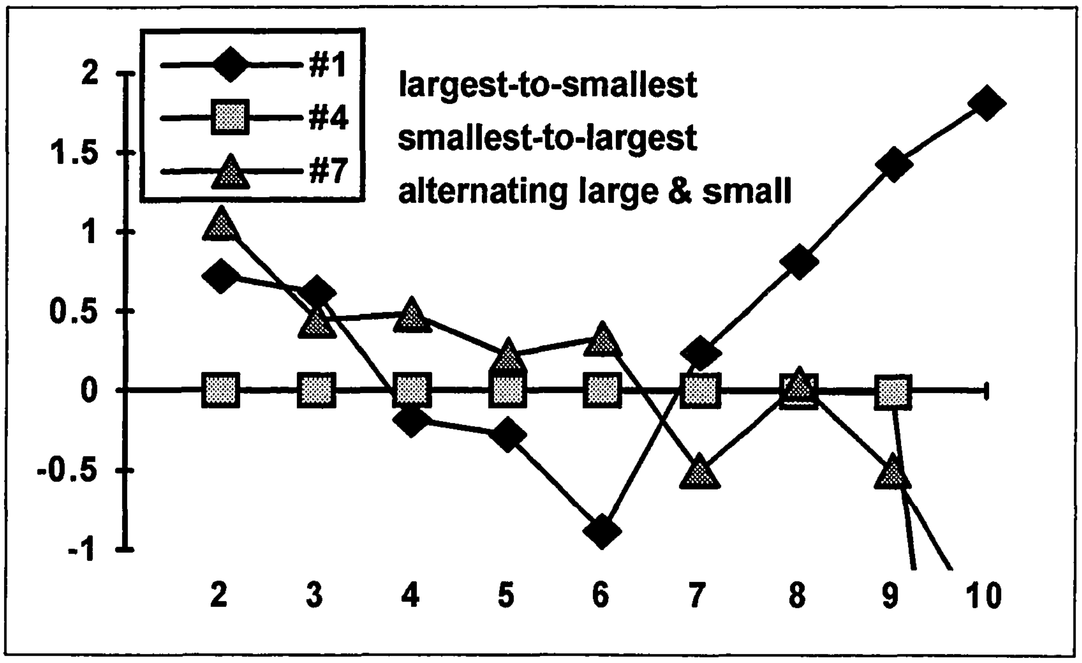

Figure 13. Prediction Ratios Compared Along Three Paths
for Geometric De-Eutrophication Model

the interfailure times, if we can control the recovery order, then we can expect more accurate predictions from these models. Further, it appears that the choice of a fault recovery criterion (e.g., largest-to-smallest, smallest-to-largest) is more important than the specific debugging graph level at which relative fault sizes are measured. This supports our conjecture that controlling fault recovery order helps to mitigate some of the failure data's noise and bias which in part may be attributable to randomness and/or interaction effects.

Chapter Seven

Surrogate Oracle

In 6.3 we concluded that fault recovery order is a significant effect which needs to be accounted for in the predictive modeling process. To make use of this information in software development, we need a means for moving our methodologies out of the realm of controlled laboratory experiments and into the production process. To this end, we investigated a potential substitute for the gold version program used for error detection. We repeated selected parts of the previous experiment using a surrogate error detector constructed using the test program with all known repairs installed.

7.1 Experiment Description

We conjectured that predictive results consistent with using data based on a gold oracle could be obtained using data based on a ***surrogate oracle***. We therefore substituted $P_{1...n}$, the program with all known faults repaired, for the gold version program in the error detector. To test the feasibility of the surrogate oracle approach, we compared predictive results based on data collected using a gold oracle versus those collected using this revised error detector.

Earlier, we described the construction of a debugging graph for the ten significant LIC bugs ($n = 10$) using one million input cases to estimate R via the gold oracle for each of the 1024 nodes (see 3.2). For this new experiment, we performed another data collection by constructing part of a second debugging graph based on the surrogate oracle. This required running each LIC test variant which appears on one of the 13 paths of interest described in the previous experiment in tandem with $P_{1...n}$ to empirically calculate the required reliability figures.

7.1.1 Path Selection Criteria

To evaluate the performance of the surrogate oracle, we compared the predictive results based on it versus those based on the gold oracle. Hence, we used the same paths formulated for the previous experiment as described in Table 9.

7.1.2 Comparison Path Data

Table 15 shows the R values for these 13 paths as measured using the surrogate oracle. We compared sequences of R values associated with corresponding gold- and surrogate-oracle-based data collections (i.e., Tables 10 and 15) and found them to agree quite closely. To illustrate their close coincidence graphically, we chose four representative paths — 1, 4, 11 and 13 — which respectively illustrate intuitive, counter-intuitive, mixed and original repair orders. Since the granularity of plotting the actual R

Table 15. Repair Numbers and R Values for Comparison Paths (Surrogate Oracle)

Initial R: 42.6673

Path 1									
1	2	5	4	3	6	8	10	7	9
94.9866	97.6968	98.6789	99.8236	99.9708	99.9994	99.9998	99.9999	99.9999	100.
Path 2									
9	7	10	8	6	3	4	5	2	1
42.6673	42.6673	42.6674	42.6677	42.6728	42.8096	43.2536	43.8763	45.3775	100.
Path 3									
1	2	4	5	3	6	8	10	9	7
94.9866	97.6968	98.8268	99.8236	99.9708	99.9994	99.9998	99.9999	100.	100.
Path 4									
7	9	10	8	6	3	5	4	2	1
42.6673	42.6673	42.6674	42.6677	42.6728	42.8096	43.4258	43.8763	45.3775	100.
Path 5									
1	2	5	4	3	6	8	10	9	7
94.9866	97.6968	98.6789	99.8236	99.9708	99.9994	99.9998	99.9999	100.	100.
Path 6									
9	7	10	8	6	3	5	4	2	1
42.6673	42.6673	42.6674	42.6677	42.6728	42.8096	43.4258	43.8763	45.3775	100.
Path 7									
1	9	2	7	5	10	4	8	3	6
94.9866	94.9866	97.6968	97.6968	98.6789	98.6790	99.8237	99.8241	99.9713	100.
Path 8									
9	1	7	2	10	5	8	4	6	3
42.6673	94.9866	94.9866	97.6968	97.6969	98.6790	98.6794	99.8241	99.8528	100.
Path 9									
1	7	2	9	4	10	5	8	3	6
94.9866	94.9866	97.6968	97.6968	98.8268	98.8269	99.8237	99.8241	99.9713	100.
Path 10									
7	1	9	2	10	4	8	5	6	3
42.6673	94.9866	94.9866	97.6968	97.6969	98.8269	98.8272	99.8241	99.8528	100.
Path 11									
3	6	4	8	5	10	2	7	1	9
42.8041	42.8091	43.2531	43.2534	43.8761	43.8762	45.3774	45.3774	99.9999	100.
Path 12									
3	6	5	8	4	10	2	9	1	7
42.8041	42.8091	43.4252	43.4256	43.8761	43.8762	45.3774	45.3775	100.	100.
Path 13									
1	2	3	4	5	6	7	8	9	10
94.9866	97.6968	97.8336	98.9636	99.9708	99.9994	99.9994	99.9998	99.9999	100.

values would obscure any differences between the gold and surrogate paths, we instead plotted the paths' R value differences (*surrogate - gold*) in Figure 14. These plots show that the differences between the gold- and surrogate-oracle-based data range between 10^{-3} to 10^{-4} percent.

7.1.3 Comparing Models' Performance

In this experiment we considered the predictive accuracy of the four models of interest when using input data derived from the partial debugging graph based on the surrogate oracle. The accuracy was assessed against two empirical reliability measures: one using the gold oracle and the second using the surrogate oracle.

7.1.3.1 Iterative Prediction Process

We repeated the iterative prediction process described in 6.1.3.1. This time we used the empirical R values shown in Table 15 to generate input sequences for each path of interest.

7.1.3.2 Normalized Comparison Data

As a measure of each model's predictive accuracy, we again performed a normalized comparison as described in 6.1.3.2 by iteratively taking the ratio of each estimated MTTF as predicted by a model to an empirical MTTF calculated from the appropriate variant's R value, and rescaling the resulting values by applying the $\mathbf{log_{10}}$ function. The logarithms of the prediction ratios for the four models are shown in Tables 16 through 19, with exceptional cases indicated by symbols and annotated in the captions.

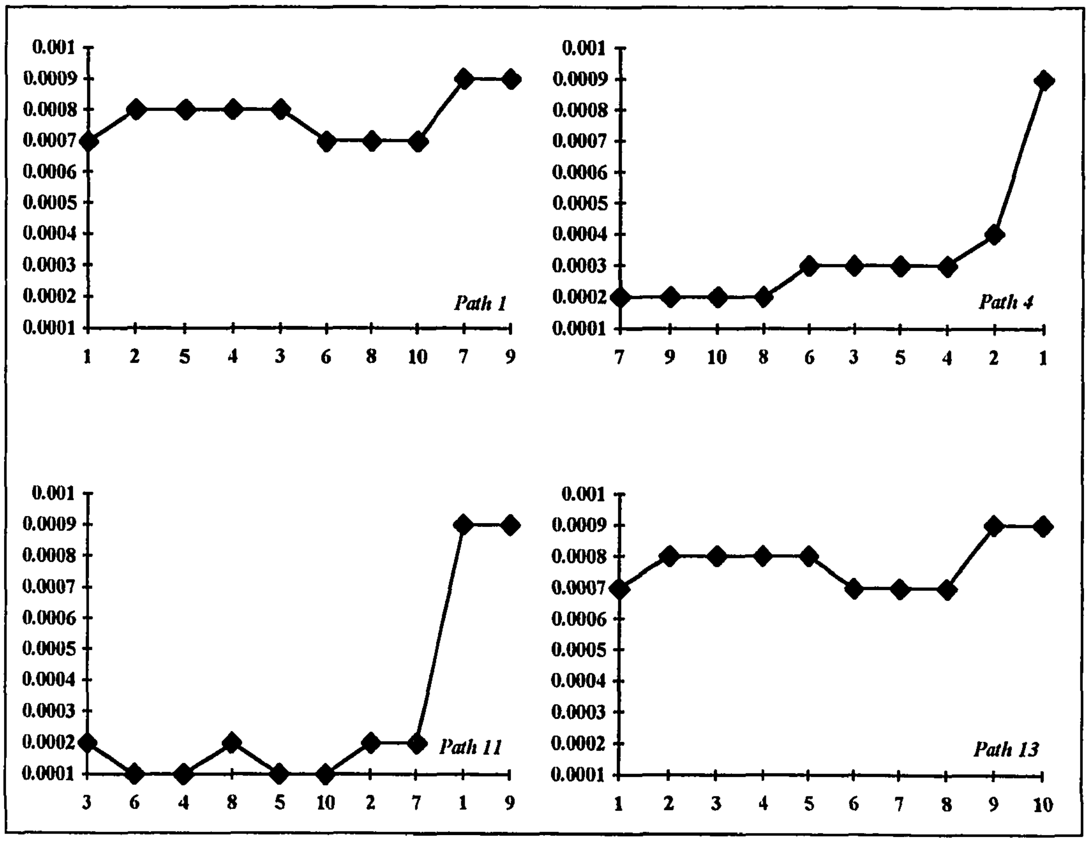

Figure 14. R Value Differences Along Some Paths (Surrogate - Gold)

From this experiment, two series of tables resulted corresponding to the two reliability measures mentioned above. We labeled the two groups of tables "a" and "b" (e.g., Table 16a, Table 16b) and also added parenthetically the information concerning the source of the input data to the models and the reliability basis for each data set as we discussed them in 6.1.3.2. In both sets of tables, the input data to the models were derived from the partial debugging graph based on the surrogate oracle (i.e., Table 15), so the first part of the parenthetical label lists "Surrogate."

The "a" series tables are marked as "(Surrogate / Surrogate)," since for them we also used the surrogate oracle as the "reliability basis" in calculating the denominator of the normalized comparison ratios. The "b" series tables are instead marked "(Surrogate / Gold)" since the gold oracle data (i.e., Table 10) were used as the reliability basis in the ratios. Thus, the "a" series tables provide a measure of the internal consistency of the models' predictions when only a surrogate oracle is used; whereas the "b" series tables measure how well the surrogate figures support predictions which are consistent with those based on the gold oracle.

Several observations should be noted before assessing these data.

- A value of zero at any Prediction Stage in the "a" series tables implies that the surrogate oracle's data produce consistent predictions as measured against its empirical approximation of the software's reliability.
- In the "b" series tables, when zero appears as an entry, it denotes perfect predictive accuracy based on the gold version.

Table 16a. Jelinski-Moranda Prediction Ratios (Surrogate / Surrogate)

? no solution; N is infinite or no R growth present

?? no solution; N is finite

Path #	MTTF Prediction Stage								
	2	3	4	5	6	7	8	9	10
1	?	??	??	??	??	??	??	??	??
2	?	?	?	?	?	?	- 6.577e-3	-0.01496	▨
3	?	??	??	??	??	??	??	??	?
4	?	?	?	?	?	?	-5.91e-3	-0.01447	▨
5	?	??	??	??	??	??	??	??	?
6	?	?	?	?	?	?	-5.91e-3	-0.01447	▨
7	?	?	??	??	??	0.6362	??	??	??
8	?	??	??	??	?	??	??	??	??
9	?	?	??	??	??	??	??	??	??
10	?	??	??	??	??	??	??	??	??
11	1.002e-3	-2.68e-3	5.512e-4	-3.329e-3	1.65e-4	-9.963e-3	-3.117e-3	-5.736	??
12	1.002e-3	-3.999e-3	5.642e-4	-1.602e-3	1.132e-3	-9.401e-3	-2.798e-3	*	?
13	?	??	??	??	??	??	??	??	??

Table 16b. Jelinski-Moranda Prediction Ratios (Surrogate / Gold)

? no solution; N is infinite or no R growth present

?? no solution; N is finite

Path #	MTTF Prediction Stage								
	2	3	4	5	6	7	8	9	10
1	?	??	??	??	??	??	??	??	??
2	?	?	?	?	?	?	-6.575e-3	-0.01495	-4.791
3	?	??	??	??	??	??	??	??	?
4	?	?	?	?	?	?	-5.908e-3	-0.01447	-4.791
5	?	??	??	??	??	??	??	??	?
6	?	?	?	?	?	?	-5.908e-3	-0.01447	-4.791
7	?	?	??	??	??	0.6382	??	??	??
8	?	??	??	??	?	??	??	??	??
9	?	?	??	??	??	??	??	??	??
10	?	??	??	??	??	??	??	??	??
11	1.002e-3	-2.679e-3	5.512e-4	-3.328e-3	1.65e-4	-9.961e-3	-3.115e-3	-4.736	??
12	1.002e-3	7.03e-4	5.642e-4	-1.601e-3	1.132e-3	-9.399e-3	-2.794e-3	-4.782	?
13	?	??	??	??	??	??	??	??	??

Table 17a. Geometric De-Eutrophication Prediction Ratios (Surrogate / Surrogate)
** indicates software predicted to be perfect*

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	0.7205	0.6122	-0.1865	-0.2891	-1.207	-0.08122	0.4241	1.081	▨
2	0	-8.686e-7	-2.171e-6	-3.865e-5	-1.05e-3	-3.828e-3	-6.5841e-3	-0.01499	▨
3	0.7205	0.5606	-0.1389	-0.2736	-1.201	-0.07949	0.4251	-∞	*
4	0	-8.686e-7	-2.171e-6	-3.865e-5	-1.05e-3	-5.148e-3	-5.921e-3	-0.01451	▨
5	0.7205	0.6122	-0.1865	-0.2891	-1.207	-0.08122	0.4241	-∞	*
6	0	-8.686e-7	-2.171e-6	-3.865e-5	-1.05e-3	-5.148e-3	-5.921e-3	-0.01451	▨
7	1.058	0.4451	0.4771	0.2190	0.3271	-0.502	0.04208	-0.5094	▨
8	-1.058	0.4129	0.266	0.4891	0.3284	0.4445	-0.3744	0.01792	▨
9	1.058	0.4451	0.4771	0.1674	0.3109	-0.4485	0.06105	-0.4958	▨
10	-1.058	0.4129	0.266	0.4891	0.2768	0.4197	-0.3299	0.03666	▨
11	9.995e-4	-2.68e-3	5.369e-4	-3.342e-3	1.225e-4	-0.01001	-3.281e-3	-5.699	▨
12	9.995e-4	-4.0e-3	5.3776e-4	-1.626e-3	1.089e-3	-9.445e-3	-2.953e-3	-∞	*
13	0.7205	0.827	0.3963	-1.07	-1.381	0.3657	0.5444	0.8491	▨

Table 17b. Geometric De-Eutrophication Prediction Ratios (Surrogate / Gold)
** indicates software predicted to be perfect*

Path	MTTF Prediction Stage								
#	2	3	4	5	6	7	8	9	10
1	0.7206	0.6125	-0.1845	-0.2773	-0.8715	0.572	1.327	2.081	2.602
2	1.3029e-6	8.686e-7	-8.686e-7	-3.648e-5	-1.047e-3	-3.826e-3	-6.581e-3	-0.01498	-4.796
3	0.7206	0.5609	-0.1369	-0.2618	-0.8652	0.5737	1.328	2.036	*
4	1.3029e-6	8.686e-7	-8.686e-7	-3.648e-5	-1.047e-3	-5.146e-3	-5.919e-3	-0.01451	-4.796
5	0.7206	0.6125	-0.1845	-0.2773	-0.8715	0.572	1.327	2.035	*
6	1.3029e-6	8.686e-7	-8.686e-7	-3.648e-5	-1.047e-3	-5.146e-3	-5.919e-3	-0.01451	-4.796
7	1.058	0.4453	0.4773	0.2193	0.3273	-0.5001	0.04405	-0.4975	-1.489
8	-1.058	0.413	0.2661	0.4892	0.3287	0.4448	-0.3725	0.02056	-1.934
9	1.058	0.4453	0.4773	0.1677	0.3112	-0.4466	0.06302	-0.4839	-1.482
10	-1.058	0.413	0.2661	0.4892	0.2771	0.42	-0.3279	-0.6614	-1.92
11	1.0e-3	-2.68e-3	5.378e-4	-3.341e-3	1.233e-4	-0.01001	-3.28e-3	-4.745	0.9012
12	1.0e-3	7.03e-4	5.386e-4	-1.625e-3	1.089e-3	-9.443e-3	-2.95e-3	-4.77	*
13	0.7206	0.8272	0.3967	-1.058	-1.045	0.7014	1.198	1.849	2.346

Table 18a. Basic Musa Prediction Ratios (Surrogate / Surrogate)
** indicates software predicted to be perfect*

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.6388	-0.7176	-0.196	7.397	*	6.677	3.119	0.5983	*
2	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8485	-0.9079	-0.966	*
3	-0.6388	-0.769	0.4927	7.223	*	6.676	3.119	*	*
4	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
5	-0.6388	-0.7176	-0.196	7.397	*	6.677	3.119	*	*
6	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
7	-0.301	-0.8149	-0.5751	-0.9404	-0.6392	-1.72	0.6419	5.94	*
8	-1.359	-0.1544	-0.9399	-0.4043	-1.019	-0.5118	-0.4905	1.268	*
9	-0.301	-0.8149	-0.5751	-0.9919	-0.5426	-1.668	0.5954	5.769	*
10	-1.359	-0.1544	-0.9399	-0.4043	-1.075	-0.4109	0.1989	1.205	*
11	-0.3011	-0.4805	-0.6021	-0.7038	-0.7782	-0.8569	-0.9031	-6.691	*
12	-0.3011	-0.4818	-0.6021	-0.7024	-0.7782	-0.8569	-0.9031	*	*
13	-0.6388	-0.5033	-0.9223	1.562	*	1.275	2.667	2.562	*

Table 18b. Basic Musa Prediction Ratios (Surrogate / Gold)
** indicates software predicted to be perfect*

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.6387	-0.7174	-0.1941	7.409	*	7.33	4.022	1.598	*
2	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8485	-0.908	-0.966	*
3	-0.6387	-0.7687	0.4947	7.235	*	7.33	4.022	*	*
4	-0.301	-0.4771	-0.6021	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
5	-0.6387	-0.7174	-0.1941	7.409	*	7.33	4.022	*	*
6	-0.301	-0.4771	-0.602	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
7	-0.301	-0.8148	-0.5749	-0.9401	-0.6389	-1.718	0.6439	5.952	*
8	-1.359	-0.1544	-0.9397	-0.4041	-1.018	-0.5116	-0.4885	1.271	*
9	-0.301	-0.8148	-0.5749	-0.9916	-0.5423	-1.666	0.5974	5.781	*
10	-1.359	-0.1544	-0.9397	-0.4041	-1.07	-0.4106	0.2009	0.5071	*
11	-0.3011	-0.4805	-0.6021	-0.7038	-0.7782	-0.8569	-0.9031	-5.691	*
12	-0.3011	-0.4771	-0.6021	-0.7024	-0.7782	-0.8569	-0.9031	*	*
13	-0.6387	-0.5031	-0.9219	10.57	*	1.61	3.321	3.562	*

Table 19a. Logarithmic Poisson Prediction Ratios (Surrogate / Surrogate)
? indicates no solution for desired precision
** indicates software predicted to be perfect*

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.0778	-0.7125	-0.5962	-1.668e-3	0.07336	0.2184	0.197	0.165	*
2	-0.301	-0.4771	?	-0.699	-0.7792	-0.8485	-0.9079	-0.966	*
3	-0.0778	-0.763	-0.4146	-3.278e-3	0.07261	0.2181	0.1967	*	?
4	-0.301	-0.4771	?	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
5	-0.0778	-0.7125	-0.5962	-1.668e-3	0.07336	0.2184	0.197	*	?
6	-0.301	-0.4771	?	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
7	-0.02018	?	-0.56	?	?	?	-0.2559	?	*
8	-1.359	-0.0459	-0.3788	-0.2887	-1.013	?	-0.8964	-0.2278	*
9	-0.02018	?	-0.56	?	-0.5576	?	-0.2588	?	*
10	-1.359	-0.0459	-0.3788	-0.2887	-1.063	-0.4136	-0.7153	-0.2289	*
11	-0.3011	-0.4805	-0.6021	?	-0.7782	-0.8569	-0.9031	-6.078	?
12	-0.3011	-0.4818	-0.6021	-0.7024	-0.7782	-0.8569	-0.9031	*	?
13	-0.0778	-0.501	?	-0.6349	0.08095	0.2212	0.1538	0.1409	*

Table 19b. Logarithmic Poisson Prediction Ratios (Surrogate / Gold)
? indicates no solution for desired precision
** indicates software predicted to be perfect*

Path	*MTTF Prediction Stage*								
#	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	-0.07765	-0.7122	-0.5943	0.01007	0.4091	0.8716	1.1	1.165	*
2	-0.301	-0.4771	?	-0.699	-0.7792	-0.8485	-0.9079	-0.966	*
3	-0.07765	-0.7627	-0.4127	8.46e-3	0.4084	0.8713	1.1	*	?
4	-0.301	-0.4771	?	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
5	-0.07765	-0.7122	-0.5943	0.01007	0.4091	0.8716	1.1	*	?
6	-0.301	-0.4771	?	-0.699	-0.7792	-0.8498	-0.9066	-0.966	*
7	-0.02012	?	-0.5598	?	?	?	-0.2539	?	*
8	-1.359	-0.04584	-0.3787	-0.2886	-1.013	?	-0.8945	-0.2252	*
9	-0.02012	?	-0.5598	?	-0.5573	?	-0.2568	?	*
10	-1.359	-0.04584	-0.3787	-0.2886	-1.063	-0.4133	-0.7133	-0.927	*
11	-0.3011	-0.4805	-0.6021	?	-0.7781	-0.8569	-0.9031	-5.078	?
12	-0.3011	-0.4771	-0.6021	-0.7024	-0.7781	-0.8569	-0.9031	*	?
13	-0.07765	-0.5008	?	-0.6231	0.4167	0.557	0.8071	1.141	*

- In producing the "a" series tables, the variant at the end of each path is indistinguishable from the surrogate. It should be noted, then, that the 10th Prediction Stage in these tables is *meaningless* in the following sense: the use of the surrogate oracle results in a "known" MTTF equal to $+\infty$, so the ratio of predicted to actual MTTF is zero. In some cases, the algorithms reached alternative conclusions (e.g., "no solution"), which are recorded in the tables to reflect the implementations' robustness (or lack of it); otherwise, we simply shaded the data position in this column. This notation does not imply a bad prediction was made by the model; rather the lack of a significant entry is due to the lack of a useful empirical value to use for comparison.
- In the "b" series tables, there were no such problems calculating the comparison ratios at the 10th Prediction Stage.

7.2 Analysis

Since our goal was to see how closely predictive performance based on surrogate oracle derived data resembles that based on gold oracle derived data, we do not at this time critique the individual models for their applicability to the specimen software; nor do we discuss the fault recovery order issues previously presented in 6.2. Comparing the prediction ratios based on data derived using the gold and surrogate oracles along a given path for a given model involves juxtaposing corresponding rows in Tables 11 through 14 with those in the "a" and "b" series of Tables 16 through 19.

It should be noted that since both the "a" and "b" series tables were produced using "surrogate" data as inputs to the models, these two sets of tables identically flag any special cases (i.e., "no solutions" or "perfect software"). So along a given path, just the magnitudes of the comparison ratios may differ due to the difference in the reliability measures in a corresponding "a-b" pairings.

First we analyze the "a" series data, which illustrate the quality of the models' predictions using the surrogate oracle as the source for both model inputs and the reliability measure. The models produced more "perfect software" reports in later stages when using data derived from the surrogate oracle. This can be attributed to the fact that the variant at the end of each path is indistinguishable from the surrogate oracle (i.e., the last measured R value is in fact 10 percent). Another general observation is that the "surrogate" data agree qualitatively with the "gold" data for all four models and along every path. This is seen even when the data are sparse, as with the Jelinski-Moranda model. In that particular case, the juxtaposed data are nearly identical to one another; the "gold" and "surrogate" data in both Tables 14 and 16b agree exactly along at least two paths (e.g., 1 and 13), and generally to within 10^{-6} along the others (e.g., 4 and 11).

The "b" series tables measure how well the models predict with respect to reliabilities as measured by the gold oracle when the models' input data are based on the surrogate oracle. The "surrogate" data under this comparison basis prove to agree quite well with the "gold" data for all the models and along all paths until the 6th or 7th Prediction Stage; thereafter, they become increasingly optimistic, much more so than the "gold" data. A notable exception once again is the Jelinski-Moranda model, whose ratios are similar in both Tables 14 and 16b.

In addition to the general observations mentioned above, the following trends are notable for the four models across the three data sets:

- The Jelinski-Moranda continued to surprise us with its accurate predictions along paths 11 and 12, as well as the later stages of paths 2, 4 and 6. This behavior is consistent for all three combinations of data conditions.
- Use of the surrogate oracle's input data produced a few "false perfect" assessments in the Geometric De-Eutrophication model's predictions; none appeared when only gold data were used. Use of the surrogate oracle as the reliability basis made the model appear to be less "wildly" optimistic in the later predictive stages, but had little effect elsewhere on any of the paths.
- Basic Musa predictions are generally at the extremes of the prediction continuum; that is, they are either very pessimistic or very optimistic regardless of the data combinations used. The model on the whole is a poor performer along all the paths. Use of the surrogate oracle's data as either inputs to the model or as the reliability basis injected predictive optimism a stage earlier than what we saw with the other models along some paths, and also caused more "false perfect" assessments to appear near the end of the prediction iterations.
- The Log Poisson model was more successful at making prediction attempts (i.e., fewer "no solutions") whenever the surrogate data were involved versus using

the gold data. Although in general corresponding paths' data agree quite closely through the 6th Prediction Stage, the model became overly optimistic later on, as is evident comparing the total number of "perfect software" predictions at the 9th and 10th Stages in the three data sets.

To illustrate the similarity of the predictions in the data based on the two different oracles, Figures 15 through 17 use bar graphs to depict comparison ratios along the four representative paths (i.e., 1, 4, 11, 13) used earlier (see 7.1.2) for all models except Jelinski-Moranda, due to the lack of significant numbers to plot in this case. The graph legends use the labeling convention we explained earlier (see 7.1.3.2) and report the "(input basis) / (reliability basis)" of the three data sets. Thus the first of the three cases used the gold oracle both to generate inputs to the models and as the reliability basis in the comparison ratios, as discussed in 6.1.3.2; the second and third cases correspond to the "a" and "b" series data discussed above. Also note in some cases the largest data values have been "cropped" to fit in these figures, but this should not affect the overall qualitative comparison of the paths.

In interpreting the bar graphs, keep in mind that positive bars indicate predictive optimism, negative bars indicate predictive pessimism, and the "zero-line" plot indicates predictive accuracy. Since plotted values much smaller than 0.001 are not discernable in these plots, we have enhanced the visibility of the near-zero data by darkening the x-axis in the corresponding data regions wherever such values occur. As an example, the plot for Path 4 in Figure 15 reveals that all three data cases produce extremely accurate

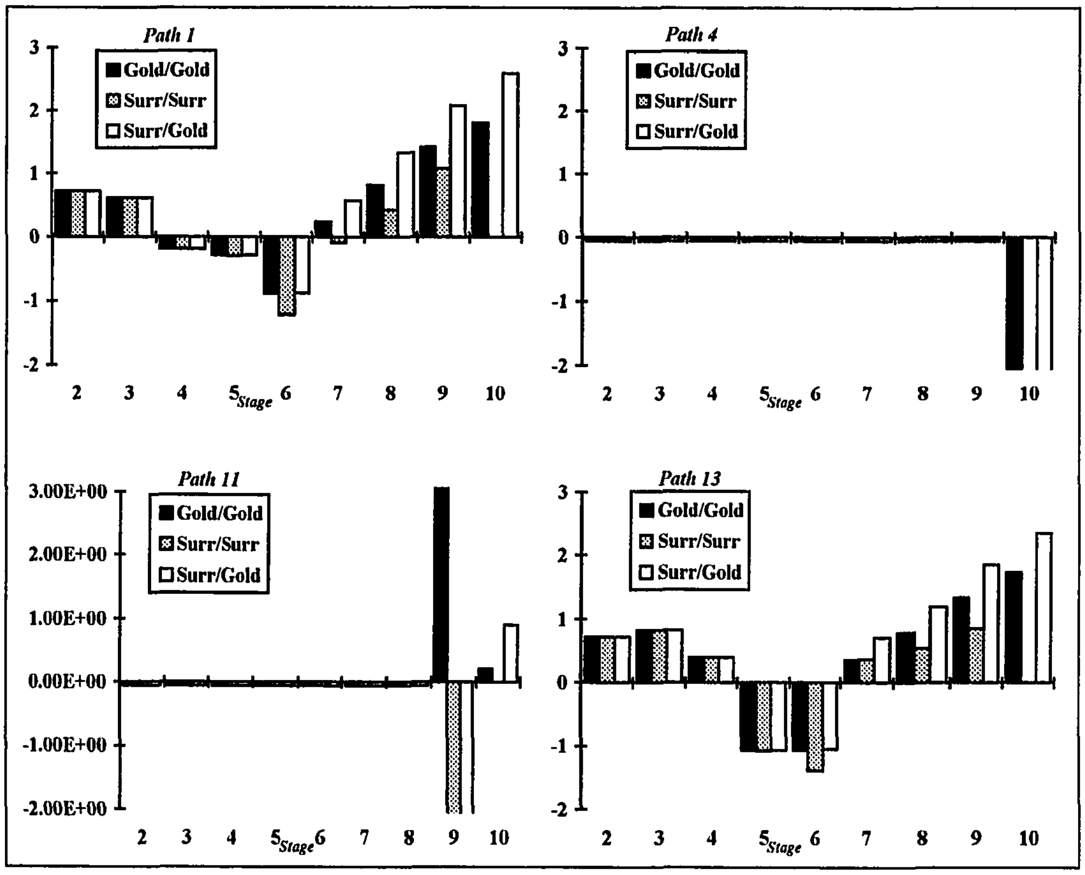

Figure 15. Geometric De-Eutrophication Prediction Ratios Comparisons

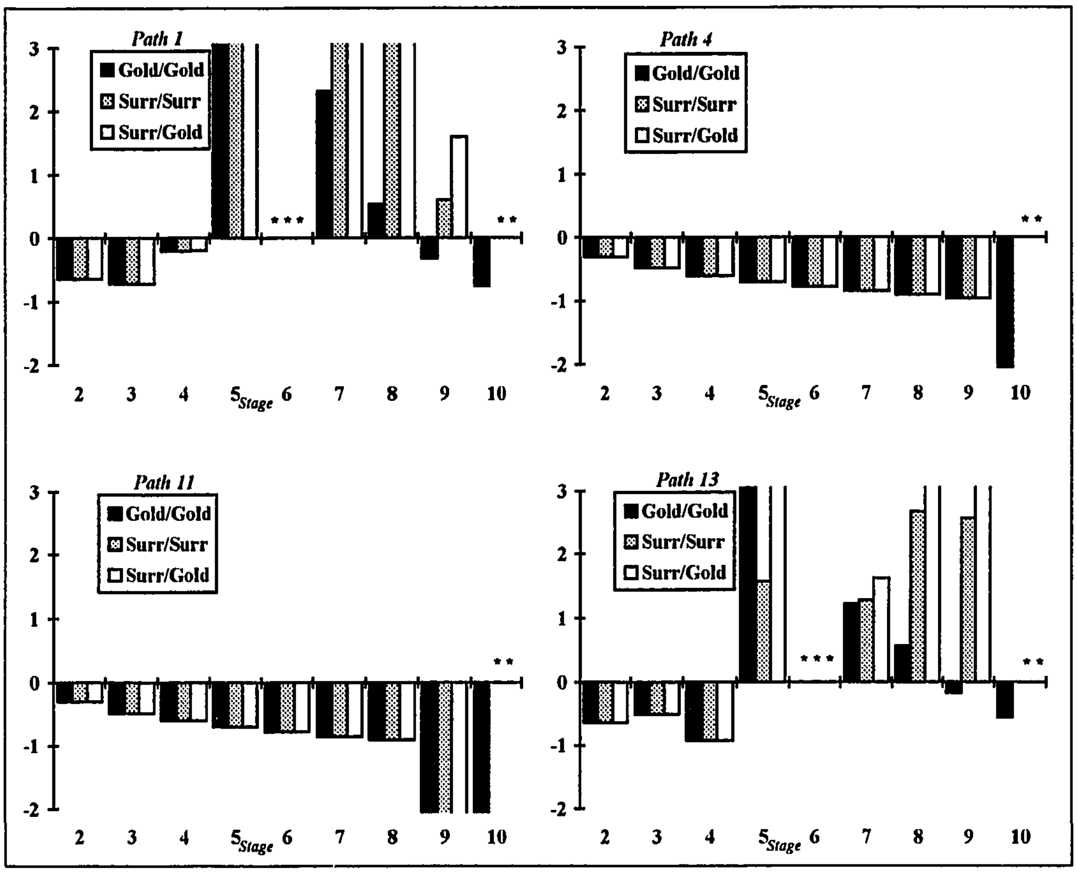

Figure 16. Basic Musa Prediction Ratios Comparisons
* *indicates software predicted to be perfect*

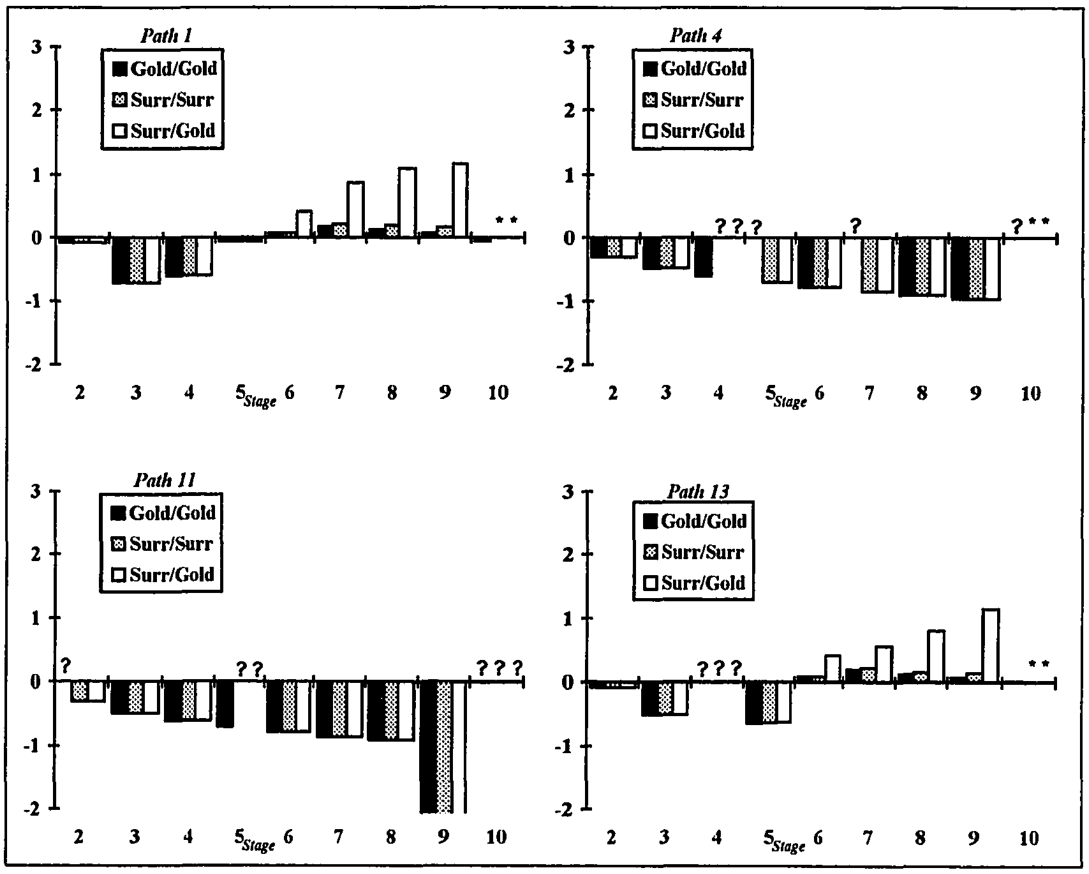

Figure 17. Logarithmic Poisson Prediction Ratios Comparisons
? indicates no solution for desired precision
** indicates software predicted to be perfect*

predictions except at the final Prediction Stage; here in two cases the model predicted failure times more than two orders of magnitude smaller than they "should" be.

The figures show that these three models' predictive performance data are quite similar in all three scenarios, although there are some observable differences. Similarity is to be expected, since the "(Surrogate / Surrogate)" and "(Surrogate / Gold)" data are by design approximations to the ideal laboratory data labeled "(Gold / Gold)." The models predict rather consistently on the counter-intuitive (4) and mixed recovery order (11) paths under all three combinations of data conditions. For the intuitive (1) and original recovery order (13) paths, all three models' data show some regions where predictions are respectable (i.e., the bar data are very small), but there is a general trend toward increasing optimism, magnified by the use of the surrogate data, as the Prediction Stage increases.

To summarize, in the first half of the iterative prediction and evaluation process, using the surrogate oracle to generate the models' inputs and/or as the reliability basis produced results consistent with those realized using the gold oracle to both generate model inputs and as the reliability basis. The primary effect of using the surrogate oracle as the reliability basis in the comparison ratios vice the gold oracle is to mask the relatively wild predictive optimism resulting from the surrogate input data after the 6th or 7th Prediction Stages. This predictive optimism is clearly evident in much greater magnitude when the predictions are assessed against the gold oracle, the "true" reliability basis.

7.3 Conclusions

To move the performance analysis methodologies described in 6.1.3 into the production process, we proposed the use of a surrogate oracle — constructed by instrumenting the specimen software with all known repairs — as a useful alternative to a gold oracle when a perfect version of the tested program is unavailable. Based on our experiments, it appears that using a surrogate oracle is feasible for generating and comparing the predictive performance data of software reliability models, provided one judiciously interprets the comparison data.

Our analysis shows that for the studied models and specimen software, using this surrogate error detector as a basis for creating the models' input data produces predictive results similar to and consistent with those obtained using a gold oracle. The predictions, however, grow increasingly optimistic after the 6th or 7th Prediction Stage, and need to be viewed with caution as we near Stage *n*. The results of our experiment also suggest that one's confidence in the predictions should probably decrease as more and more data are added to the input set, so that controlling the number of input data points may be recommendable. We will address this consideration in the next chapter.

We saw in the prediction assessment component of this study the effect of not being able to distinguish the program $P_{1,\ldots,10}$ from the surrogate oracle; i.e., the last program variant along the surrogate debugging paths is assigned a reliability of 100 percent. In most cases, this did not cause the models to completely fail to make predictions at the later stages; but there was a higher incidence of incorrect "perfect" predictions at and near the final stage. This is an inconvenient artifact of using the surrogate oracle when some number of bugs remain in the program.

Under these conditions, approximation techniques may be useful to correct the R sequence along the debugging path from the inflated values measured by the surrogate oracle. For example, a biasing factor could be derived from the prediction ratios assessed earlier along the debugging path and used to lower the later reliability figures to more realistic values, before predictions to Stage n+1 and beyond are performed. Future experiments should address this general problem as it applies to using a surrogate oracle when predictions are required.

Finally, this experiment revealed that our laboratory work on bug sizes and fault recovery order could have been conducted using just the surrogate oracle, and we would have reached similar conclusions as those based on the gold oracle. Thus the surrogate oracle has immediate uses in laboratory experiments and some potential also exists for further applications.

Chapter Eight

Truncated Paths

A legitimate concern in using the debugging graph as a basis for manipulating software reliability prediction data is that, as the number of known faults increases, the breadth of the graph and the number of levels — that is, the length of the debugging paths — increases. The computational component of the problem grows commensurately, as do storage requirements if failure data are tracked. The notion of data aging offers a means of "trimming" the graph by enabling one to intelligently eliminate from consideration some portion of the known failure data, with an attendant reduction in time and space needs for data collection and subsequent predictive performance analysis.

Further, in 7.3 we mentioned the possible need to control the size of the data set input to the predictive models as a means of mitigating potential skewing of the data due to latent effects in the failure history. In this chapter we describe an experiment in which the models' predictive performance is studied along four representative debugging paths whose data were subjected to an aging criterion.

8.1 Experiment Description

We chose four representative paths from those described in Table 9 to use in this experiment: an intutitive path (1); a counter-intuitive path (4); a mixed recovery order path (11); and the original repair order path (13). For each of the four models, we applied the iterative prediction process to predict MTTF_i at each stage *i* (see 6.1.3.1); only instead of using MTTF_0 through MTTF_{i-1} to formulate each prediction, a window of size *w* was used to limit the number of inputs to *w* values consisting of MTTF_{i-w} through MTTF_{i-1}. (Note: This is the second of Schneidewind's aging approaches as described in 2.5.3.) For example, if $w = 4$, then predicting MTTF_9 involves using MTTF_5 through MTTF_8.

We allowed *w* to range from 2 through 10 for each path; these values are shown in the column labeled "*# Pts*" in Tables 20 through 23. The shaded regions of the tables denote where predictions were not made due to the combination of *w* and Prediction Stage; since in general, when limiting the inputs to the last *w* MTTF values, the first possible prediction occurs at stage *w*. As before, exceptional cases are labeled with symbols and annotated in the captions. The tables are also labeled parenthetically as "(Gold / Gold)" to indicate that the source of both the input data to the models and the reliability basis are data derived from the gold oracle.

Table 20. Jelinski-Moranda Aged Prediction Ratios (Gold / Gold)
? no solution; N is infinite or no R growth present
?? no solution; N is finite

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	?	?	?	?	?	0.1549	?
3		??	?	??	??	??	??	0.265	?
4			??	??	??	??	??	??	0.06168
5				??	??	??	??	??	0.46288
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	?	?	?	-3.665e-3	1.284e-3	-8.274e-3	-4.771
3		?	?	?	?	-4.332e-3	-1.171e-3	-7.419e-3	-4.777
4			?	?	?	-4.683	-2.902e-3	-8.833e-3	-4.779
5				?	?	-4.899e-3	-4.055e-3	-0.01046	-4.781
6					?	?	-4.863e-3	-0.01181	-4.784
7						?	-5.457e-3	-0.01289	-4.786
8							-5.91e-3	-0.01376	-4.788
9								-0.01447	-4.789
10									-4.79

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	1.002e-3	?	3.409e-3	?	4.846e-3	?	0.0121	?	?
3		-2.68e-3	1.167e-3	-2.528e-3	1.63e-3	-8.564e-3	4.124e-3	-4.729	??
4			5.512e-4	-3.069e-3	1.734e-3	-9.354e-3	2.598e-3	-4.731	??
5				-3.328e-3	7.94e-4	-8.946e-3	2.301e-4	-4.732	??
6					1.65e-4	-9.499e-3	-6.854-4	-4.733	??
7						-9.961e-3	-2.023e-3	-4.734	??
8							-3.116e-3	-4.735	??
9								-4.736	??
10									??

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	-0.2917	?	?	?	?	0.301	?
3		??	0.04854	-1.066	??	??	?	0.1445	0.05112
4			??	-0.2434	??	??	??	??	0.05043
5				??	??	??	??	??	0.4882
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Table 21. Geometric De-Eutrophication Aged Prediction Ratios (Gold / Gold)

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	0.7205	0.09643	-0.6315	0.1014	-0.5918	1.203	0.1085	0.1481	-0.1427
3		0.6122	-0.5666	-0.2964	-0.5236	0.8294	1.011	0.2212	-0.04253
4			-0.1848	-0.43	-0.8065	0.7078	1.228	0.7815	0.0654
5				-0.2794	-0.9609	0.4115	1.258	1.379	0.4775
6					-0.8821	0.2258	0.992	1.706	1.054
7						0.2363	0.7957	1.559	1.69
8							0.8131	1.381	1.823
9								1.421	1.726
10									1.809

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	0	-8.686e-7	-1.303e-6	-3.561e-5	-9.996e-4	-3.667e-3	1.233e-3	-8.302e-3	-4.77
3		-8.686e-7	-2.171e-6	-3.648e-5	-1.024e-3	-4.334e-3	-1.211e-3	-7.48e-3	-4.77
4			-2.171e-6	-3.735e-5	-1.036e-3	-4.684e-3	-2.932e-3	-8.903e-3	-4.77
5				-3.822e-5	-1.044e-3	-4.899e-3	-4.078e-3	-0.01052	-4.77
6					-1.05e-3	-5.044e-3	-4.88e-3	-0.01187	-4.796
7						-5.148e-3	-5.47-3	-0.01294	-4.796
8							-5.921e-3	-0.01381	-4.796
9								-0.01451	-4.796
10									-4.796

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	9.986e-4	-3.346e-3	3.383e-3	-4.79e-3	4.791e-3	-0.01177	0.01178	-4.745	4.692
3		-2.68e-3	1.152e-3	-2.534e-3	1.6e-3	-8.578e-3	3.934e-3	-4.721	2.148
4			5.373e-4	-3.08e-3	1.695e-3	-9.375e-3	2.414e-3	-4.721	1.239
5				-3.342e-3	7.515e-4	-8.98e-3	6.297e-5	-4.721	0.7479
6					1.229e-4	-9.54e-3	-8.56e-4	-4.745	0.4336
7						-0.01001	-2.19e-3	-4.745	0.2094
8							-3.28e-3	-4.745	0.03989
9								-4.745	-0.0951
10									-0.2057

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	0.7205	0.3111	-0.2935	-1.219	0.1756	1.363	-0.1597	0.2055	-0.09152
3		0.8269	-0.07962	-1.409	-0.5595	1.482	0.8775	0.1006	0.04822
4			0.3964	-1.353	-0.9766	0.9404	1.75	0.6059	0.03774
5				-1.059	-1.123	0.5287	1.453	1.523	0.394
6					-1.058	0.3459	0.9866	1.904	0.9829
7						0.3514	0.7746	1.519	1.805
8							0.7884	1.309	1.8
9								1.343	1.657
10									1.732

Table 22. Basic Musa Aged Prediction Ratios (Gold / Gold)
** indicates software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6387	-0.5423	-1.174	-1.072	-1.664	-0.4607	-0.3522	-0.2041	-0.3468
3		-0.7177	-1.35	1.554	10.59	0.2074	-0.5283	-0.3802	-0.5229
4			-0.2023	4.566	*	0.7916	-0.4164	-0.5052	-0.6478
5				7.186	*	1.314	-0.1476	-0.6021	-0.7447
6					*	1.815	0.09126	-0.5427	-0.8239
7						2.31	0.3161	-0.4694	-0.8909
8							0.5342	-0.3944	-0.9488
9								-0.318	-0.8796
10									-0.7588

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.301	-0.301	-0.301	-0.3011	-0.3021	-0.3057	-0.3045	-0.3128	-5.084
3		-0.4771	-0.4771	-0.4772	-0.4782	-0.4818	-0.4806	-0.4889	-5.26
4			-0.6021	-0.6021	-0.6031	-0.6068	-0.6055	-0.6138	-5.385
5				-0.6997	-0.7	-0.7037	-0.7024	-0.7107	-5.482
6					-0.7792	-0.7829	-0.7816	-0.7899	-5.561
7						-0.8498	-0.8486	-0.8569	-5.628
8							-0.9066	-0.9149	-5.686
9								-0.966	-5.737
10									-5.783

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.301	-0.3044	-0.301	-0.3058	-0.301	-0.3128	-0.301	-5.038	-0.3468
3		-0.4805	-0.4771	-0.4819	-0.4771	-0.4889	-0.4771	-5.214	0.128
4			-0.6021	-0.6069	-0.6021	-0.6138	-0.6021	-5.339	0.5185
5				-0.7038	-0.7	-0.7107	-0.7	-5.436	0.8869
6					-0.7782	-0.7899	-0.7782	-5.515	1.252
7						-0.8569	-0.8451	-5.582	1.622
8							-0.9031	-5.64	1.999
9								-5.692	2.384
10									2.777

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6387	-0.3276	-0.6211	-1.84	-1.664	-0.301	-0.4607	-0.2553	-0.3468
3		-0.5034	-0.7972	-2.016	*	7.991e-3	-0.6368	-0.4314	-0.5229
4			-0.9221	-2.141	*	0.3176	-0.7618	-0.5563	-0.6478
5				10.218	*	0.614	-0.3795	-0.6172	-0.7447
6					*	0.9108	-0.02983	-0.4991	-0.8239
7						1.213	0.2782	-0.3877	-0.8909
8							0.5632	-0.2797	-0.8188
9								-0.1732	-0.6837
10									-0.5603

Table 23. Logarithmic Poisson Aged Prediction Ratios (Gold / Gold)
? indicates no solution for desired precision

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.0777	?	?	?	?	0.1523	?	?	?
3		-0.7139	?	-0.1268	0.02478	0.2326	?	?	?
4			-0.5969	-0.09896	0.04831	0.2526	?	?	?
5				-2.276e-3	0.02724	0.2118	?	-0.05947	?
6					0.07112	?	0.1396	0.05689	?
7						0.1829	0.1143	0.06503	-0.06914
8							0.1315	0.05223	-0.02464
9								0.07551	-0.02295
10									7.594e-3

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.301	-0.301	-0.301	?	-0.3021	?	-0.3045	?	?
3		-0.4771	-0.4771	?	-0.4782	?	-0.4806	?	?
4			-0.6021	?	-0.6031	?	-0.6055	-0.6138	?
5				?	-0.7	?	-0.7024	-0.7107	?
6					-0.7792	?	-0.7816	-0.7899	?
7						?	-0.8486	-0.8569	?
8							-0.9066	-0.9149	?
9								-0.966	?
10									?

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	-0.3044	-0.301	-0.3058	?	-0.3128	-0.301	-4.952	?
3		-0.4805	-0.4771	-0.4819	?	-0.4889	-0.4771	-5.128	?
4			-0.6021	-0.6069	-0.6021	-0.6138	-0.6021	-5.253	?
5				-0.7038	-0.699	-0.7107	-0.699	-5.349	?
6					-0.7782	-0.7899	-0.7782	-5.429	?
7						-0.8569	-0.8451	-5.496	?
8							-0.9031	-5.554	?
9								-5.605	?
10									?

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.0777	?	?	?	-0.8989	0.1613	?	?	?
3		-0.5017	?	?	0.2	0.3728	?	?	?
4			?	?	0.1113	0.3141	0.1998	?	?
5				-0.6354	?	0.2396	0.1967	0.07829	?
6					0.07882	0.1858	0.1464	0.1234	?
7						0.1863	?	0.08904	0.02956
8							0.1176	?	0.01248
9								0.07434	-5.529e-3
10									0.01353

8.2 Analysis

To assess the effects of data aging on the predictive process, we contrasted the sequence of ratios of predicted to known MTTFs, transformed using **log$_{10}$**, for a given model and path as obtained when using the cumulative approach — *all* known data up to a current predictive stage (see 6.1.3.2) — versus the nine sequences for different *w* values calculated during this experiment. We made the following observations:

- Data aging allowed the Jelinski-Moranda model to attempt a few more predictions on the original repair order path (13). Otherwise, using even the smallest data window produced results consistent with the original predictive performance along these four paths as previously examined. That is, the algorithm assessed a finite number of bugs along the intutitive path (1) after the first few stages, the counter-intuitive path (4) did well only in the later stages, and the mixed recovery order path (11) did well until the last two stages.
- Once again, for the counter-intuitive (4) and the mixed order (11) paths, the Geometric De-Eutrophication model produced extremely accurate overall predictions. Using data aging did nothing to eliminate the later skewing along these paths, but neither did it unreasonably worsen it. Along the other two paths (1, 13), with cumulative data the model appeared to try to make a correction at one point and then grew increasingly optimistic in its predictions. Applying data aging made the accuracy of the predictions result in an oscillating pattern of

optimistic and pessimistic predictions, as though the model were continually trying to make corrections. This resulted in inconsistent predictive performance across the stages regardless of the window size.

- The overall performance of the Basic Musa model remained relatively poor even using data aging, although it was somewhat less wildly optimistic. One observable difference was the increased incidence of "false perfect" predictions when data aging was applied.
- The predictive accuracy of the Logarithmic Poisson model appeared to degrade somewhat when applying the data aging criterion. That is, we saw an increased incidence of "no solutions," indicating an inability to find parametric values which "fit" the functions to the truncated data. The model still did well during the middle and late predictive stages along the intuitive path (1), and respectably along the original order path (13). Unlike when cumulative data were used, along paths 4 and 11, for a given window size the comparison ratios were consistent across the stages. This suggests that a simple correction factor might be useful in realigning the predictions along some paths.

An interesting general observation is that along all four representative paths, given a window size as small as 2, the models perform quite well. In fact, the models generally predict as well or better using a smaller window versus a larger one. That is, at any Prediction Stage, the first few ratios listed in each column are respectably close to the remaining ratios in the column — and this is consistent across a fixed row. To illustrate this point, Figures 18 and 19 show the prediction ratios resulting for paths 1 and 4 based on the gold oracle for all four models, using both the cumulative approach and the aging criterion with $w = 2$ and with a "medium" sized window, $w = 5$. Note that while we have once again enchanced the visibility of the near-zero data by darkening the x-axis in the corresponding regions, the $w = 5$ ratio data do not start until Stage 5.

The path 1 results are somewhat inconsistent among the four models. But in general one can see that windowing in many cases pulls the predictions into better agreement with the empirically measured reliability (i.e., the plotted ratios are closer to zero), and many times better performance is gained using the smaller window. A notable exception is the Logarithmic Poisson model, which has the best aged performance along this path, and with the medium sized window. For path 4, it is evident that the Jelinski-Moranda and Geometric De-Eutrophication cumulative and aged data align quite nicely. The aged Basic Musa and Logarithmic Poisson predictions are more optimistic than their cumulative counterparts, but this actually results in greater predictive accuracy in many cases, with $w = 2$ approaching better accuracy than $w = 5$.

To investigate the combined effects of windowing and the surrogate oracle, we ran the data for the four representative paths derived from the surrogate oracle through the models using the various w values; the results are shown in Tables 24 through 27. Once again, there are two series of tables, labeled "a" and "b." Using the convention "(input basis / reliability measure)," the "a" tables are also labeled "(Surrogate / Surrogate)" and correspond to using the surrogate oracle as the source for the model's input data as well

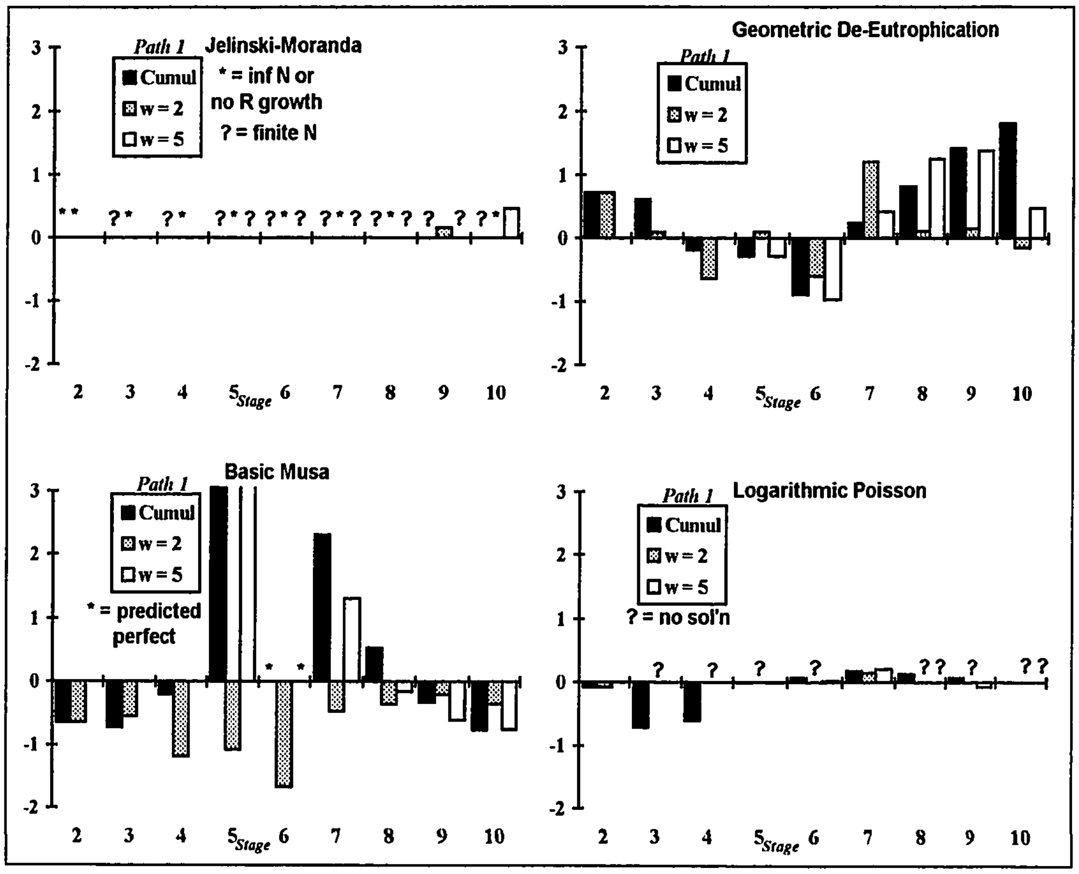

Figure 18. Cumulative Versus Aged Prediction Ratio Comparisons for Path 1 (Gold / Gold)

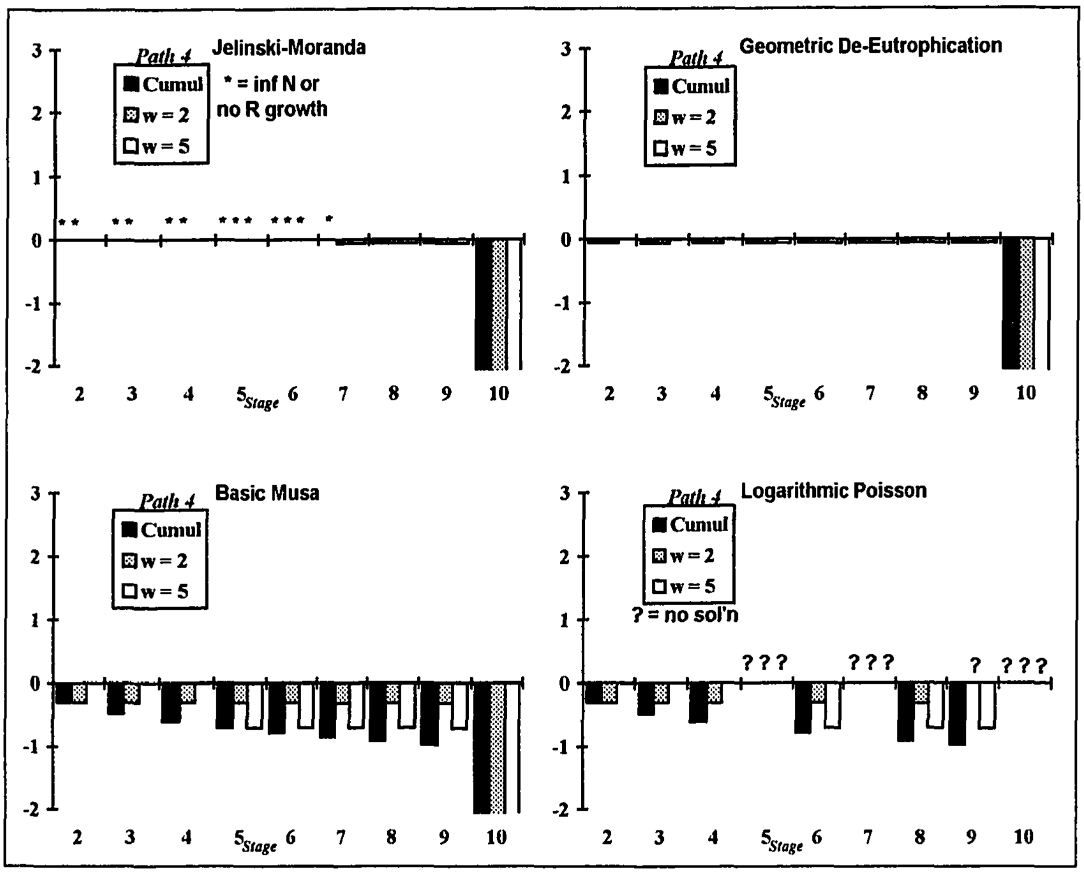

Figure 19. Cumulative Versus Aged Prediction Ratio Comparisons for Path 4 (Gold / Gold)

Table 24a. Jelinski-Moranda Aged Prediction Ratios (Surrogate / Surrogate)

? no solution; N is infinite or no R growth present

?? no solution; N is finite

** software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	?	?	-0.2837	?	?	?	?	*	?
3		??	?	??	??	??	??	?	
4			??	??	??	??	??	??	
5				??	??	??	??	??	??
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Path 4

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	?	?	?	?	?	-3.665e-3	1.284e-3	-8.275e-3	
3		?	?	?	?	-4.331e-3	-1.171e-3	-7.42e-3	
4			?	?	?	-4.683e-3	-2.902e-3	-8.834e-3	
5				?	?	-4.898e-3	-4.055e-3	-0.01046	
6					?	?	-4.863e-3	-0.01181	
7						?	-5.456e-3	-0.01289	
8							-5.91e-3	-0.01376	
9								-0.01447	
10									

Path 11

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	1.002e-3	?	3.409e-3	?	4.846e-3	?	0.0121	?	?
3		-2.68e-3	1.167e-3	-2.528e-3	1.63e-3	-8.565e-3	4.124e-3	-5.729	??
4			5.512e-4	-3.069e-3	1.734e-3	-9.355e-3	2.598e-3	-5.731	??
5				-3.329e-3	7.897e-4	-8.947e-3	2.301e-4	-5.732	??
6					1.65e-4	-9.5e-3	-6.859e-4	-5.733	??
7						-9.963e-3	-2.024e-3	-5.734	??
8							-3.117e-3	-5.735	??
9								-5.736	??
10									??

Path 13

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	?	?	-0.2919	?	?	?	?	?	*
3		??	0.04853	-1.076	??	??	?	?	?
4			??	-0.2488	??	??	??	??	??
5				??	??	??	??	??	??
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Table 24b. Jelinski-Moranda Aged Prediction Ratios (Surrogate / Gold)

? no solution; N is infinite or no R growth present

?? no solution; N is finite

** software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	-0.2817	?	?	?	?	*	?
3		??	?	??	??	??	??	?	1.229
4			??	??	??	??	??	??	2.314
5				??	??	??	??	??	??
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	?	?	?	-3.662e-3	1.288e-3	-8.271e-3	-4.771
3		?	?	?	?	-4.329e-3	-1.169e-3	-7.416e-3	-4.777
4			?	?	?	-4.681e-3	-2.899e-3	-8.831e-3	-4.779
5				?	?	-4.896e-3	-4.053e-3	-0.01045	-4.781
6					?	?	-4.86e-3	-0.01181	-4.784
7						?	-5.454e-3	-0.01289	-4.786
8							-5.908e-3	-0.01376	-4.788
9								-0.01447	-4.789
10									-4.791

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	1.002e-3	?	3.409e-3	?	4.846e-3	?	0.0121	?	?
3		-2.68e-3	1.167e-3	-2.527e-3	1.63e-3	-8.563e-3	4.124e-3	-4.729	??
4			5.512e-4	-3.069e-3	1.734e-3	-9.354e-3	2.602e-3	-4.731	??
5				-3.328e-3	7.94e-4	-8.945e-3	2.301e-4	-4.732	??
6					1.65e-4	-9.499e-3	-6.841e-4	-4.733	??
7						-9.961e-3	-2.022e-3	-4.734	??
8							-3.115e-3	-4.735	??
9								-4.736	??
10									??

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	?	?	-0.2915	?	?	?	?	?	*
3		??	0.04887	-1.065	??	??	?	?	?
4			??	-0.2371	??	??	??	??	??
5				??	??	??	??	??	??
6					??	??	??	??	??
7						??	??	??	??
8							??	??	??
9								??	??
10									??

Table 25a. Geometric De-Eutrophication Aged Prediction Ratios (Surrogate / Surrogate)

Path 1

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	0.7205	0.0964	-0.633	0.09332	-0.9061	1.21	0.1761	0.301	
3		0.6122	-0.5682	-0.3054	-0.8433	0.6514	1.084	0.4205	
4			-0.1865	-0.4394	-1.129	0.4529	1.086	1.011	
5				-0.2891	-1.285	0.1246	0.9861	1.472	
6					-1.207	-0.07594	0.6542	1.594	
7						-0.08122	0.4316	1.312	
8							0.4241	1.08	
9								1.081	
10									

Path 4

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	0	-8.686e-7	-1.303e-6	-3.648e-5	-9.992e-4	-3.667e-3	1.233e-3	-8.303e-3	
3		-8.686e-7	-2.171e-6	-3.735e-5	-1.023e-3	-4.333e-3	-1.211e-3	-7.481e-3	
4			-2.171e-6	-3.822e-5	-1.036e-3	-4.684e-3	-2.931e-3	-8.904e-3	
5				-3.865e-5	-1.044e-3	-4.899e-3	-4.077e-3	-0.01052	
6					-1.05e-3	-5.045e-3	-4.88e-3	-0.01187	
7						-5.148e-3	-5.47e-3	-0.01294	
8							-5.921e-3	-0.01381	
9								-0.01451	
10									

Path 11

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	9.995e-4	-3.347e-3	3.383e-3	-4.79e-3	4.791e-3	-0.01178	0.01177	-5.699	
3		-2.68e-3	1.152e-3	-2.534e-3	1. 6e-3	-8.578e-3	3.934e-3	-5.699	
4			5.369e-4	-3.08e-3	1.695e-3	-9.376e-3	2.414e-3	-5.699	
5				-3.342e-3	7.515e-4	-8.98e-3	6.253e-5	-5.699	
6					1.225e-4	-9.541e-3	-8.56e-4	-5.699	
7						-0.01	-2.191e-3	-5.699	
8							-3.281e-3	-5.699	
9								-5.699	
10									

Path 13

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	0.7205	0.3112	-0.2936	-1.23	-0.1371	1.687	-0.4771	0.1761	
3		0.827	-0.07973	-1.42	-0.8784	1.597	0.8425	-0.1284	
4			0.3963	-1.364	-1.298	0.9941	1.776	0.3555	
5				-1.07	-1.445	0.5691	1.307	1.329	
6					-1.381	0.3735	0.786	1.569	
7						0.3657	0.5465	1.087	
8							0.5444	0.8378	
9								0.8491	
10									

Table 25b. Geometric De-Eutrophication Aged Prediction Ratios (Surrogate / Gold)

Path 1

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	0.7206	0.09667	-0.6311	0.1051	-0.5703	1.863	1.079	1.301	0.9542
3		0.6125	-0.5662	-0.2937	-0.5075	1.305	1.987	1.42	1.161
4			-0.1845	-0.4277	-0.7934	1.106	1.989	2.011	1.366
5				-0.2773	-0.9492	0.7778	1.889	2.472	1.85
6					-0.8715	0.5773	1.557	2.594	2.445
7						0.572	1.335	2.312	2.899
8							1.327	2.08	2.784
9								2.081	2.577
10									2.602

Path 4

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	1.303e-6	8.686e-7	0	-3.388e-5	-9.965e-4	-3.665e-3	1.235e-3	-8.3e-3	-4.77
3		8.686e-7	-4.343e-7	-3.518e-5	-1.021e-3	-4.331e-3	-1.209e-3	-7.477e-3	-4.77
4			-8.686e-7	-3.605e-5	-1.034e-3	-4.682e-3	-2.929e-3	-8.901e-3	-4.77
5				-3.648e-5	-1.042e-3	-4.897e-3	-4.075e-3	-0.01052	-4.77
6					-1.047e-3	-5.042e-3	-4.877e-3	-0.01186	-4.77
7						-5.146e-3	-5.468e-3	-0.01294	-4.796
8							-5.919e-3	-0.0138	-4.796
9								-0.01451	-4.796
10									-4.796

Path 11

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	1.0e-3	-3.346e-3	3.383e-3	-4.789e-3	4.792e-3	-0.01177	0.01178	-4.745	6.692
3		-2.68e-3	1.153e-3	-2.533e-3	1.6e-3	-8.577e-3	3.936e-3	-4.721	3.647
4			5.378e-4	-3.079e-3	1.696e-3	-9.3748e-3	2.415e-3	-4.721	2.571
5				-3.341e-3	7.524e-4	-8.9787e-3	6.427e-5	-4.721	1.996
6					1.233e-4	-9.5399e-3	-8.547e-4	-4.745	1.631
7						-0.01	-2.198e-3	-4.745	1.373
8							-3.28e-3	-4.745	1.179
9								-4.745	1.026
10									0.9012

Path 13

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	0.7206	0.3114	-0.2933	-1.218	0.1987	2.023	0.1761	1.176	1.255
3		0.8272	-0.0794	-1.409	-0.5426	1.933	1.496	0.8716	1.375
4			0.3967	-1.352	-0.9618	1.33	2.429	1.355	1.19
5				-1.058	-1.109	0.9049	1.961	2.329	1.492
6					-1.045	0.7093	1.439	2.569	2.011
7						0.7014	1.2	2.087	2.697
8							1.198	1.838	2.517
9								1.849	2.308
10									2.346

Table 26a. Basic Musa Aged Prediction Ratios (Surrogate / Surrogate)
* *indicates software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6388	-0.5424	-1.175	-1.082	-1.988	-0.7782	-0.6021	-0.301	*
3		-0.7176	-1.352	1.622	*	1.324	-0.3514	-0.4771	*
4			-0.1961	4.711	*	2.887	0.5561	-0.3654	*
5				7.397	*	4.226	4.2263	-0.1552	*
6					*	5.4718	1.926	0.03938	*
7						6.677	2.532	0.2271	*
8							3.119	0.4126	*
9								0.5983	*
10									*

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.301	-0.301	-0.301	-0.3011	-0.3021	-0.3057	-0.3045	-0.3128	*
3		-0.4771	-0.4771	-0.4772	-0.4782	-0.4818	-0.4806	-0.4889	*
4			-0.6021	-0.6021	-0.6031	-0.6068	-0.6055	-0.6138	*
5				-0.699	-0.7	-0.7037	-0.7024	-0.7107	*
6					-0.7792	-0.7829	-0.7816	-0.7899	*
7						-0.8498	-0.8486	-0.8569	*
8							-0.9066	-0.9149	*
9								-0.966	*
10									*

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.3011	-0.3044	-0.301	-0.3058	-0.301	-0.3128	-0.301	-6.038	*
3		-0.4805	-0.4805	-0.4819	-0.4771	-0.4889	-0.4771	-6.214	*
4			-0.6021	-0.6069	-0.6021	-0.6138	-0.6021	-6.339	*
5				-0.7038	-0.699	-0.7107	-0.699	-6.436	*
6					-0.7782	-0.7899	-0.7782	-6.515	*
7						-0.8569	-0.8451	-6.582	*
8							-0.9031	-6.64	*
9								-6.691	*
10									*

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6388	-0.3276	-0.6212	-1.851	-1.988	-0.301	-0.7782	-0.6021	*
3		-0.5033	-0.7973	-2.027	*	0.04021	-0.9542	-0.7782	*
4			-0.9223	-2.152	*	0.3555	-1.079	-0.2368	*
5				10.56	*	0.659	0.1241	0.4655	*
6					*	0.9636	1.097	1.054	*
7						1.275	1.924	1.584	*
8							2.667	2.082	*
9								2.562	*
10									*

Table 26b. Basic Musa Aged Prediction Ratios (Surrogate / Gold)
** indicates software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6388	-0.5424	-1.175	-1.082	-1.988	-0.7782	-0.6021	-0.301	*
3		-0.7176	-1.352	1.622	*	1.324	-0.3514	-0.4771	*
4			-0.196	4.711	*	2.887	0.5561	-0.3654	*
5				7.397	*	4.226	1.28	-0.1552	*
6					*	5.472	1.926	0.03938	*
7						6.677	2.532	0.2271	*
8							3.119	0.4126	*
9								0.5983	*
10									*

Path 4

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.301	-0.301	-0.301	-0.3011	-0.3021	-0.3057	-0.3045	-0.3128	*
3		-0.4771	-0.4771	-0.4772	-0.4782	-0.4818	-0.4806	-0.4889	*
4			-0.6021	-0.6021	-0.6031	-0.6068	-0.6055	-0.6138	*
5				-0.699	-0.7	-0.7037	-0.7024	-0.7107	*
6					-0.7792	-0.7829	-0.7816	-0.7899	*
7						-0.8498	-0.8486	-0.8569	*
8							-0.9066	-0.9149	*
9								-0.966	*
10									*

Path 11

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.3011	-0.3044	-0.301	-0.3058	-0.301	-0.3128	-0.301	-6.038	*
3		-0.4805	-0.4771	-0.4819	-0.4771	-0.4889	-0.4771	-6.214	*
4			-0.6021	-0.6069	-0.6021	-0.6138	-0.6021	-6.339	*
5				-0.7038	-0.699	-0.7107	-0.699	-6.436	*
6					-0.7782	-0.7899	-0.7782	-6.515	*
7						-0.8569	-0.8451	-6.582	*
8							-0.9031	-6.64	*
9								-6.691	*
10									*

Path 13

#	*MTTF Prediction Stage*								
Pts	2	3	4	5	6	7	8	9	10
2	-0.6388	-0.3276	-0.6212	-1.851	-1.988	-0.301	-0.7782	-0.6021	*
3		-0.5033	-0.7973	-2.027	*	0.04021	-0.9542	-0.7782	*
4			-0.9223	-2.152	*	0.3555	-1.0792	-0.2368	*
5				10.56	*	0.659	0.1241	0.4655	*
6					*	0.96363	1.097	1.054	*
7						1.275	1.924	1.584	*
8							2.667	2.082	*
9								2.562	*
10									*

Table 27a. Logarithmic Poisson Aged Prediction Ratios (Surrogate / Surrogate)
? indicates no solution for desired precision
** indicates software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.0778	?	?	?	?	0.306	?	?	?
3		-0.7125	?	-0.1256	0.0308	0.3108	0.1822	?	?
4			-0.5962	-0.09812	0.0519	0.3072	0.2424	0.03685	?
5				-1.668e-3	0.03011	0.2581	0.257	0.1707	?
6					0.07336	?	0.2241	0.2003	?
7						0.2184	?	0.1804	?
8							0.197	?	*
9								0.165	*
10									*

Path 4

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.301	-0.301	?	-0.3011	-0.3021	-0.3057	-0.3045	-0.3128	*
3		-0.4771	?	-0.4772	-0.4782	-0.4818	-0.4806	-0.4889	*
4			?	-0.6021	-0.6031	-0.6068	-0.6055	-0.6138	*
5				-0.699	-0.7	-0.7037	-0.7024	-0.7107	*
6					-0.7792	-0.7829	-0.7816	-0.7899	*
7						-0.8498	-0.8486	-0.8569	*
8							-0.9066	-0.9149	*
9								-0.966	*
10									*

Path 11

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.3011	-0.3044	-0.301	?	-0.301	-0.3128	-0.301	-5.425	?
3		-0.4805	-0.4771	?	-0.4771	-0.4889	-0.4771	-5.601	?
4			-0.6021	?	-0.6021	-0.6138	-0.6021	-5.726	?
5				?	-0.699	-0.7107	-0.699	-5.823	?
6					-0.7782	-0.7899	-0.7782	-5.902	?
7						-0.8569	-0.8451	-5.969	?
8							-0.9031	-6.027	?
9								-6.078	?
10									?

Path 13

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.0778	?	?	?	-0.94859	0.3112	?	?	?
3		-0.501	?	?	0.2033	0.4284	?	?	?
4			?	?	0.1142	0.3595	?	-0.2621	?
5				-0.6349	?	?	0.2453	0.2195	*
6					0.08095	0.2257	0.1908	0.2124	*
7						0.2212	?	0.1696	*
8							0.1538	?	*
9								0.1409	?
10									*

Table 27b. Logarithmic Poisson Aged Prediction Ratios (Surrogate / Gold)

? indicates no solution for desired precision

** indicates software predicted to be perfect*

Path 1

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.0776	?	?	?	?	0.9592	?	?	?
3	▨	-0.7122	?	-0.1139	0.3666	0.964	1.085	?	?
4	▨	▨	-0.5943	-0.0864	0.3877	0.96044	1.145	1.037	?
5	▨	▨	▨	0.01007	0.3659	0.9113	1.16	1.171	?
6	▨	▨	▨	▨	0.4091	?	1.127	1.2	?
7	▨	▨	▨	▨	▨	0.8716	?	1.18	?
8	▨	▨	▨	▨	▨	▨	1.1	?	*
9	▨	▨	▨	▨	▨	▨	▨	1.165	*
10	▨	▨	▨	▨	▨	▨	▨	▨	*

Path 4

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.301	-0.301	?	-0.3011	-0.3021	-0.3057	-0.3045	-0.3128	*
3	▨	-0.4771	?	-0.4772	-0.4782	-0.4818	-0.4806	-0.4889	*
4	▨	▨	?	-0.6021	-0.6031	-0.6068	-0.6055	-0.6138	*
5	▨	▨	▨	-0.699	-0.7	-0.70371	-0.7024	-0.7107	*
6	▨	▨	▨	▨	-0.7792	-0.7829	-0.7816	-0.7899	*
7	▨	▨	▨	▨	▨	-0.8498	-0.8486	-0.8569	*
8	▨	▨	▨	▨	▨	▨	-0.9066	-0.9149	*
9	▨	▨	▨	▨	▨	▨	▨	-0.966	*
10	▨	▨	▨	▨	▨	▨	▨	▨	*

Path 11

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.3011	-0.3044	-0.301	?	-0.301	-0.3128	-0.301	-4.425	?
3	▨	-0.4805	-0.4771	?	-0.4771	-0.4889	-0.4771	-4.601	?
4	▨	▨	-0.6021	?	-0.6021	-0.6138	-0.6021	-4.726	?
5	▨	▨	▨	?	-0.699	-0.7107	-0.699	-4.822	?
6	▨	▨	▨	▨	-0.7782	-0.7899	-0.7781	-4.902	?
7	▨	▨	▨	▨	▨	-0.8569	-0.8451	-4.969	?
8	▨	▨	▨	▨	▨	▨	-0.9031	-5.027	?
9	▨	▨	▨	▨	▨	▨	▨	-5.078	?
10	▨	▨	▨	▨	▨	▨	▨	▨	?

Path 13

#	*MTTF Prediction Stage*								
Pts	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	-0.07765	?	?	?	-0.6127	0.647	?	?	?
3	▨	-0.5008	?	?	0.5391	0.7642	?	?	?
4	▨	▨	?	?	0.45	0.6953	?	0.7379	?
5	▨	▨	▨	-0.6231	?	?	0.8985	1.22	*
6	▨	▨	▨	▨	0.4167	0.5615	0.844	1.212	*
7	▨	▨	▨	▨	▨	0.557	?	1.17	*
8	▨	▨	▨	▨	▨	▨	0.8071	?	*
9	▨	▨	▨	▨	▨	▨	▨	1.141	?
10	▨	▨	▨	▨	▨	▨	▨	▨	*

as the reliability basis in the comparison ratios. The "b" series tables instead use the gold oracle data as the reliability basis and are labeled "(Surrogate / Gold)".Recall that at the 10th Prediction Stage, unless the model reached an alternative conclusion, the "a" series table entries are shaded, since the fully instrumented test version is indistinguishable from the surrogate oracle. The idea here is to see how data produced using data windows along with data from the surrogate oracle compares with our previous conjectures: that results based on surrogate oracle data are consistent with those based on the gold oracle, and that less data can be as good, if not better, than more data in many cases.

When assessing the models' predictive abilities using the surrogate data as the reliability basis (i.e., the "a" series tables), these data generally appear to support the conjectures. We also noted the following:

- The Jelinski-Moranda model made fewer prediction attempts — but those made are quite good ones according to the ratios — using surrogate data. Interestingly, the choice of *w* appears to be less important than the path selection.
- The Geometric De-Eutrophication model made some overly optimistic predictions at the later stages and with the larger *w* values. The surrogate data on the whole were consistent in magnitude with the gold data. Notable exceptions were the extremely pessimistic predictions at the 9th Stage along the mixed order path, which nonetheless exhibited good agreement for all window sizes at each fixed stage.

- The primary effect of data aging on the surrogate data for the Basic Musa model was to degrade performance as *w* increased; data based on the smaller *w* values are slightly more pessimistic than we had seen using the gold oracle.
- The Logarithmic Poisson model's aged surrogate data are consistent with the aged gold data, though slightly more optimistic, with the smaller *w* values resulting in better performance than the larger values.

Assessing the models using the gold oracle as the reliability basis (i.e., the "b" series tables) reveals results which are consistent for all models. On paths 1 and 13, the predictions are generally very optimistic after the 7th Prediction Stage; performance along paths 4 and 11 was generally good. Once again, the range of values down a given column in any of these tables leads us to conclude that smaller window sizes result in performance consistent with that derived from larger window values. Again, for comparison purposes, Figures 20 and 21 graphically depicts the prediction ratios resulting for paths 1 and 4 based on the surrogate oracle for all four models, using both the cumulative approach and the aging criterion for $w = 2$ and $w = 5$. In this case, we have used the surrogate oracle as the reliability basis for both the cumulative and the aged data, since our previous experiments as reported in Section 7 revealed that these data are consistent with those using the gold oracle. Recall that for $w = 5$, the ratio data do not start until the 5th Stage.

For either window size, where differences exist between the aged and cumulative path 1 data sets, it is apparent that use of the surrogate data introduced optimistic bias into the predictions. Likewise, the plots for path 4 are nearly indistinguishable from those

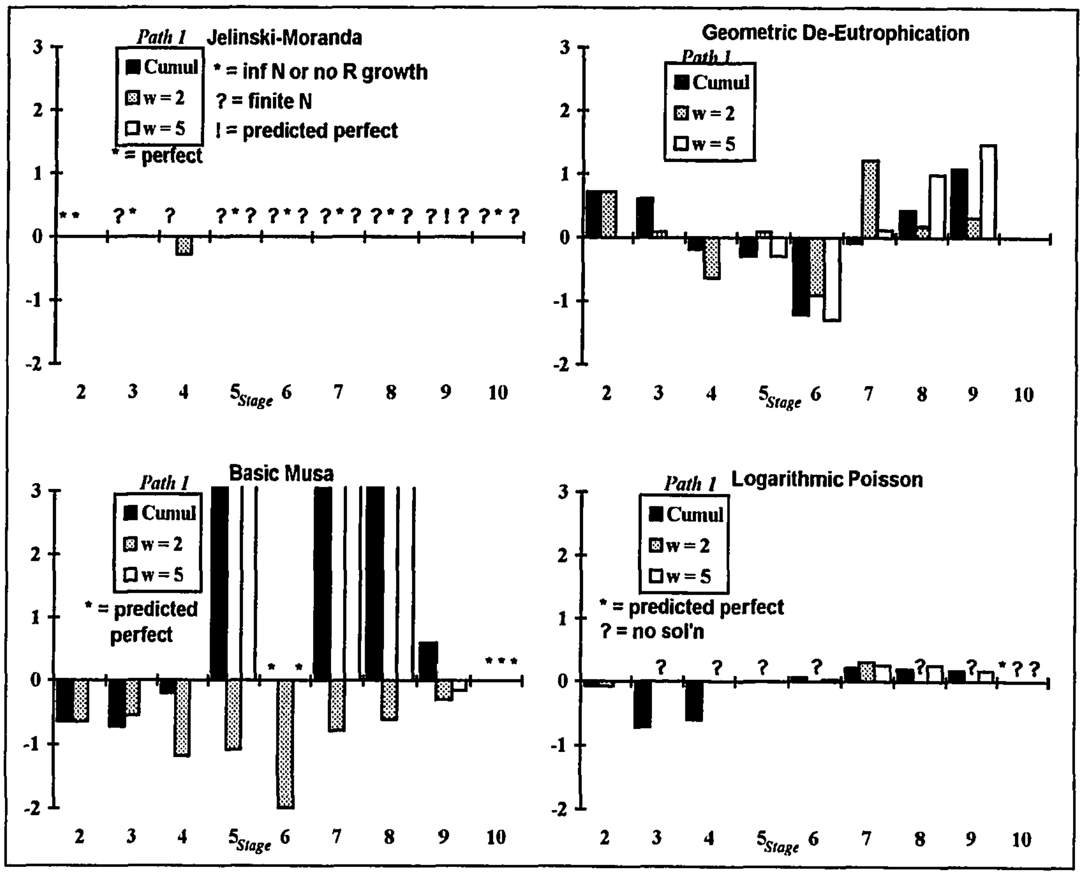

Figure 20. Cumulative Versus Aged Prediction Ratio Comparisons for Path 1 (Surrogate / Surrogate)

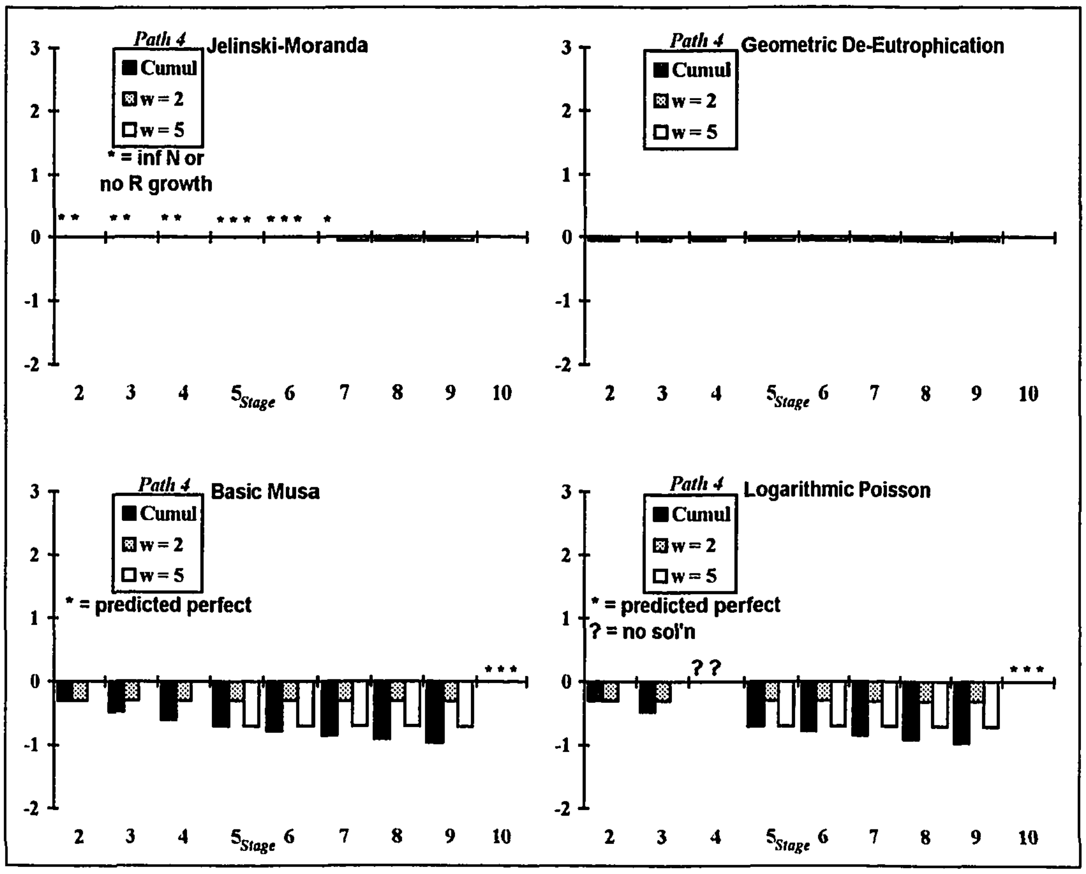

Figure 21. Cumulative Versus Aged Prediction Ratio Comparisons for Path 4 (Surrogate / Surrogate)

shown in Figures 18 and 19, when the gold oracle was used to both generate the inputs to the models and as the reliability basis. Thus, the observations made above based on using the gold oracle can probably be translated to this application of aging, provided one is cautious regarding the potential for predictive optimism once the surrogate data are introduced.

8.3 Conclusions

For the specimen software and the representative debugging paths, data aging — or ***path truncation*** as we call it — appears to be a viable avenue for controlling the computational component of debugging graph applications. Under path truncation, poor performance was observed to persist for models and paths which performed badly when all known data were used; although sometimes improvements (e.g., more and/or better predictions) were seen using the aged data. For models and paths which performed well in the presence of all known data, aging the data did not greatly change their performance.

A surprising result of this experiment is the observation that in many cases "less is better;" that is, a relatively small window size produces respectable, sometimes even better, predictions for a model observed to have done well using more data. Thus, if controlled fault recovery orders are intelligently simulated using the debugging graph model, eliminating some of the "older" data in the resulting MTTF sequences can help to keep the computational expense small. The practitioner can thus choose model(s) and path(s) which produce more accurate predictions whether a gold or surrogate oracle is used as the reliability basis.

The general utility of data aging is somewhat complicated, however, when a surrogate oracle is used. As we noted in 7.3, the surrogate oracle's inflated assessments of reliability during the later debugging stages results in predictive optimism in the later prediction stages when cumulative data are used. Use of data aging limits the model's focus to these inflated values and can inject extreme predictive optimism, as reflected in the prediction ratios for the "Surrogate / Gold" aged data presented in this chapter. This effect can be particularly problematic when trying to predict as we near Stage *n* as discussed in the previous chapter's conclusions.

Thus where applying data aging to software reliability prediction is concerned, our experiments endorse it as a useful technique when a gold oracle is used. Further we can state that it is a promising methodology for use with the surrogate oracle, provided we can find means to compensate for the optimistic bias in the surrogate's reliability measurements towards the end of the debugging path.

Chapter Nine

New Methodologies

In this chapter, we use what we have learned from our laboratory work to propose some methodologies for applying the techniques described in this thesis outside a laboratory setting.

9.1 Software Reliability Engineering Applications

An obvious goal of the debugging graph research is to produce means for obtaining better predictions from existing software reliability models. In a controlled laboratory setting, when a gold version program is available, the experiments we described in this thesis enable one to determine the "best" fault recovery order which can be used to establish a measurable upper bound on predictive models' accuracy. Further, we can select a model and a path to improve predictions. We looked in particular at one version of the LIC software and, after studying it in great detail, we would probably recommend using the Logarithmic Poisson model for making its predictions with an intuitive fault recovery path derived using replicated failure data.

In attempting to move laboratory methodologies into the domain of software development, we applied shortcuts — such as the surrogate oracle and path truncation. Typically, approximations degrade our ability to attain an optimal level of performance as compared to laboratory results; but occasionally, improvements are observed. For the LIC software, we found that using data derived using a surrogate oracle led us to conclusions that were consistent with those based on the gold oracle. However, our ability to analyze predictive performance in the laboratory revealed that the models became increasingly optimistic in the later stages as the program variants towards the end of the debugging path became increasingly indistinguishable from the surrogate. We also found that limiting our attention to the last four or five data points along a path produces predictions as good or better than using all the historical data when a gold oracle is available. The technique holds promise in the context of the surrogate oracle, provided we can devise means to compensate for the surrogate oracle's optimistic bias in the latter reliability measurement phases.

Some challenges remain, such as: validating the conclusions reached in this thesis using other LIC versions; investigating the methods' applicability to similar, then different, categories of software; determining whether the aged data window size can be set to a small constant such as four or five, or a percentage of the known failure data, such as one-half; devising methods to account for the surrogate oracle's measurement bias; and quantifying the computational cost versus predictive improvement benefits of debugging graph techniques in general. The ultimate goal, of course, is to refine the techniques into a unified methodology applicable to *any* category of software, consisting of a single model, a single type of fault recovery path produced from replicated data based on a surrogate oracle, and a small, fixed-sized data window.

In the subsequent paragraphs, we presume that this goal is attainable and discuss the implications of this contribution in terms of its use by the practioner.

9.2 An Approach to Using a Partial Debugging Graph

We begin by describing how one would go about making a single prediction assuming that *n* faults are known thus far. We then address subsequent iterations of the prediction process as new faults are found.

9.2.1 Initial Prediction

A computationally efficient way to make a more accurate software reliability prediction based on the premise in 9.1 is summarized in Figure 22. When *n* faults are known thus far, one would start by constructing the variants P_1 through P_n as a basis for determining a single, standardized, size-based fault recovery order (e.g., largest-to-smallest). The *n* known faults would be ranked according to relative sizes as determined by this collection of variants' R values. We assume a surrogate oracle would be used for

```
calculate $R(P)$
construct $P_1 \dots P_n$
calculate $R(P_1) \dots R(P_n)$
sort R values
construct program variants comprising standardized path
calculate R values for the path's program variants
input path's R values to model to make a prediction
```

Figure 22. Making a Single Reliability Prediction

measuring the variants' R values and to size-rank the faults, since in most cases a gold version program does not exist.

Although we intend to apply a data window w (with w ultimately much smaller than n) to reduce the computational complexity of the predictive procedure, to simplify this basic discussion of the methodology we will assume that initially n equals w. So the program variants corresponding to the standardized path would be constructed next. That is, suppose the size-ranked ordering of the known faults is represented by the sequence i_1 through i_n. The standardized path is represented by the sequence of variants P, P_{i1}, $P_{i1,i2}$, . . . , $P_{i1,i2,...,in}$.

The R values of this sequence of variants, as measured by the surrogate oracle, formulate the inputs to the predictive model. A single prediction can be made using all known data. Alternatively, an iterative prediction procedure might be followed as described in 6.1.3.1 if the practitioner would like to use the model's past performance along this path to assess any bias in the current prediction.

9.2.2 Subsequent Iterations

Figure 23 shows an approach to making subsequent predictions once the algorithm outlined above has been followed. When a new fault is found and corrected, several effects must be considered. The information pool of single-repair variants should be updated to reflect the newly discovered fault's size. To do this by measuring the reliability of the program variant with only the new repair installed, first the new repair also should

```
n = n+1
find / correct bug n
add repair n to the surrogate oracle
construct $P_n$
calculate $R(P_n)$
recalculate $R(P)$, $R(P_1)$ . . . $R(P_n)$ if desired
sort R values
choose "most recent" w+d R values
construct any new program variants for standardized path
calculate R values for any new program variants along the path
input "most recent" w+d R values to model to make a prediction
```

Figure 23. Making Subsequent Predictions

be added to the surrogate oracle. But changing the surrogate oracle implies that the sizes assessed for the previously known faults may change, although probably not by much.

Since we advocate applying a data window *w* in the predictive procedure, a reasonable alternative to recalculating all the single-repair reliability data at this point is to "discard" some historical data for the first *n-w* faults. Concerns over potential re-ordering of older faults (i.e., Are we really throwing away the "oldest" data along the standardized path?) may be addressed by re-measuring the corresponding single-repair variants on each iteration with the improved surrogate oracle, and re-ordering the faults prior to discarding any data. A more computationally efficient alternative is to track data for the "most recent" *w+d* faults along the standardized path, where *d* is a small "buffer factor" to account for any minor re-ordering. The issue of re-measuring the previous single-repair variants can be ignored altogether, or they may be re-measured at discrete intervals as time permits.

So when a new fault is found and corrected, its repair would be added to the surrogate oracle. The R value of the variant containing just the new repair would be measured by the improved surrogate oracle, and this data would be added to the information pool of fault size data. As we discussed above, the ordering of existing faults is unlikely to change very much if they are re-measured at this point by the improved oracle, so re-measuring the other single-repair variants is optional. All known faults would be re-ordered as needed to compose the standardized path, with some of the data "discarded" to maintain the fixed window size before the data are fed to the model to make a prediction.

Depending on where a newly discovered fault relatively ranks with respect to the older faults' sizes, the number of program variants one needs to construct and measure to present the standardized path's data to the predictive model will vary, as will the amount of additional computation required. That is, if a newly found fault is the smallest yet discovered, and a largest-to-smallest path construction criterion is being used, just one additional program variant's data needs to be calculated to represent the new standardized path.

9.3 Complexity Analysis

The algorithms sketched above contain several components contributing to the computational complexity of the proposed method; among them are the oracle runs and the reliability model implementation. Other factors to consider include the construction of the program variants and sorting the R values. Each algorithm aspect has time and space requirements that must be addressed in implementation. We will for this analysis assume that space is not a problem, since storage media are cheap and it is not difficult to back up incremental data on tape. The input size for the functions involved in the ***"big-O" notation*** calculations, then, is the number of observed failures, n.

9.3.1 Oracle Complexity

The computational complexity of the oracle, whether it is gold or surrogate, is driven by the tested software, as well as the number of test cases used, rather than n. The computational cost of the tested software will vary from case to case, so we cannot address it definitively here beyond stating that it is a real factor that needs to be considered in running the empirical reliability calculations. The number of test cases required in performing these calculations is constrained by the accuracy to which we desire each variant's reliability to be determined, and the size of the smallest fault we wish to be able to observe. This is true because we must exercise the software with enough input cases for the empirical reliability calculations to carry the desired decimal place accuracy, and to capture the occurrence(s) of (presumably) infrequently occurring failures.

There may exist some functional relationship between n and the number of input cases needed — if we know, for example, that the sizes of subsequent bugs are falling off by an order of magnitude — but we cannot state this in the general case. Thus we will represent the time required for one execution of the oracle — i.e., the calculation of R for one program variant — as a constant T_O, with the understanding that the precise value may be empirically observed and improved as needed.

9.3.2 Model Complexity

The complexity of the reliability model implementation is a function of the number of observed failures *n*. The fundamental implementations of the four reliability models we used are of linear time in *n*. But because we further explored an incremental prediction procedure, in which we start with two known times to failure and add an additional one at each predictive stage, the complexity of the algorithms discussed in Appendix D, as we implemented them, is actually $O(n^2)$. For the present analysis, we will estimate the computational complexity of executing a single predictive model with a single path's worth of data as $O(n)$, and as $O(n^2)$ if iterative predictions are considered.

9.3.3 Process Complexity

It is easy to automate the construction of program variants using a collection of faulty and repaired code fragments. We mention how to accomplish this with simple shell scripts in Appendix C; or, many source code revision control systems are available that provide canned procedures for doing so. In general, the piecing together of the required code to represent one variant would be of linear time in *n*. It is well known that $O(n \log n)$ comparisons are necessary and sufficient to sort a sequence of *n* elements [3, page 77], so this is the worst-case cost of sorting the R values to support path construction. Finally, in Figure 24 we have combined the two procedures described above into a single iterative algorithm, and augmented each significant step with a worst-case

```
calculate $R(P)$                                                  $T_O$
construct $P_1 \ldots P_n$                                        $n \cdot O(n)$
calculate $R(P_1) \ldots R(P_n)$                                  $n \cdot T_O$
loop:  sort R values                                              $O(n \log n)$
choose "most recent" w+d R values
construct any new program variants for path                       $n \cdot O(n)$
calculate R values for any new program variants                   $n \cdot T_O$
input path's "most recent" w R values to model & predict          $O(n^2)$
n = n+1
find & correct bug n
add repair n to the surrogate oracle                              $O(n)$
construct $P_n$                                                   $O(n)$
calculate $R(P_n)$                                                $T_O$
recalculate $R(P)$, $R(P_1) \ldots R(P_n)$ if desired             $n \cdot T_O$
go to loop
```

Figure 24. Algorithmic Complexity Analysis

complexity estimate. One can easily conclude that the cost of this kind of predictive procedure is dominated by $O(n^2)$ time.

9.4 Wall-Clock Analysis

While complexity analysis is a useful tool for comparing the relative performance of algorithms in an objective way, a more pressing concern to the practitioner is how long in terms of wall-clock time it actually takes to run a piece of software or to perform a particular procedure. For the proposed methodology, some start-up time would be required to set up the data collection environment; we estimate no more than a few days' effort would be needed if the configuration and scripting techniques described in this thesis are applied.

For each run of the LIC software oracle used to determine an R value, we used one million test cases, and each one-million-run execution of the oracle required approximately four hours of wall-clock time to complete. This figure obviously varies per subject software and host environment. Optimizations may need to be addressed to improve the oracle's execution time for some other piece of software. It should be noted that the oracle runs can take place as a background process, overnight, and/or be distributed across network nodes to minimize the impact of this part of the methodology on the practitioner's other tasks.

Additionally, the ***reification*** of the software reliability models is a critical component of the problem, since real-world difficulties must be faced in the parametric approximation schemes required by the algorithms. We found in early versions of our modeling software, for example, that straightforward iterations often decayed into endless loops or aborts in the face of real numbers versus textbook examples. Such implementation challenges could often be attributed to limits in representational accuracy on the host system, the ***consistent comparison problem*** [7] and over- and underflow. So although the predictive runs of the models required a negligible amount of time when compared to the oracle, we caution the implementor of alternative reliablity models to provide "escape hatches" in the iterative loops typically used to fit the models' parameters — such as fixing the maximum number of iterations, or testing for "no further change" in the approximated values — to limit the contribution of these loops to a constant factor in the worst case.

We will now use the LIC software as an example to calculate the real-world time to execute the proposed methodology; for simplification, path truncation will not be applied. To make an initial prediction for the LIC software using the algorithm sketched in Figure 22 as a guide would require approximately $4 \cdot [\,(n+1) + (n-1)] = 8n$ hours. This is how long it would take to run the oracle for each of the n single-repair variants as well as the original unrepaired program, plus the time required to run the oracle for each of the variants along the standardized path (noting that the R values for the first two variants along this path have already been calculated in the previous step). We have ignored the time needed to sort the R values and to run the predictive model as they are negligible compared to the oracle's execution time.

Each subsequent iteration of the procedure as depicted in Figure 23 requires roughly half the time of the initial run. Assuming that the existing single-repair variants are not re-measured once a new fault is found, just one additional run of the oracle is required to measure the new fault's size, which contributes 4 hours. In the worst case, generating the data representing the standardized path would require (n-1) additional oracle runs, if the new fault must be inserted "first" in the sequence of size-ranked faults. Thus, each subsequent iteration of the algorithm requires $4n$ hours.

Again, it should be noted that the times provided here assume performing the oracle runs sequentially on a single processor; distributing the work across a network can reduce the time to just a few hours depending on the number of processors available.

9.5 Fault Interaction Concerns

One point of concern is how serious might be latent fault interaction effects. Would assessing the faults' sizes at each of two or more graph levels, and using those data to construct multiple realizations of the standardized path result in significantly different predictions from the model? Recall that we used two different size orders for the LIC faults when we discovered discrepancies between the apparent fault sizes at two different levels of the debugging graph (see 4.1.1).

We mentioned in Chapter 6 that predictive performances along paths constructed using a particular sizing criterion (e.g., largest-to-smallest) appeared similar despite differences in the levels used to perform static relative size ranking. The performance along a largest-to-smallest path constructed with respect to level 1 of the LIC debugging graph, for example, was analogous to that along the level-9-based largest-to-smallest path. Thus it is our hypothesis that by controlling the fault recovery order by some consistent means, we are mitigating fault interaction effects; whereas, if one simply uses the fault recovery data in whichever order it is naturally encountered, then the models' performance is stressed and degraded not only by randomness in the recovery process, but also by latent interaction effects.

It may be instructive to measure the program variants each containing only one known fault as a redundancy check on the level-1-based fault size ordering we proposed in the practitioner's approach above. If the number of known faults is n, this will always require constructing and testing $n+1$ additional program variants on each debugging iteration, since the new repair must be added to each of the previously existing program variants at debugging graph level n-1. It should be recognized, however, that the computational expense of this procedure each time a new fault is repaired may not be worthwhile, unless computing resources are plentiful, debugging time maximal, and we can thereby derive some information useful to improve the prediction process.

An alternative approach we proposed in 5.1.3 is dynamic relative size ranking, which assesses the sizes of the faults at progressively higher levels in the debugging graph. This presumably would account better for localized interaction effects given the repairs installed so far in an iterative fault recovery scheme. Although we were unsuccessful in constructing a "greedy" path using this method (see 5.1), such an approach may be useful with other software specimens.

We note that the primary computational cost difference between using static and dynamic relative size ranking is the number of program variants which must be constructed and run through the oracle in order to dynamically re-assess the relative fault sizes. With a little additional thought, it should be obvious that the dynamic approach requires

$$1 + n + (n\text{-}1) + (n\text{-}2) + \ldots + (n\text{-}\,(n\text{-}2)\,) + 1 = [\, n \cdot (n+1) \,/\, 2 \,] + 1$$

program variants to measure the relative fault sizes, where n is the number of known faults. That is, after determining the empirical reliability of the completely unrepaired variant, the first addition to the path requires inspecting n program variants, one containing each of the n known repairs. The second addition to the path requires examining variants for each of the n-1 remaining known repairs, paired with the fix corresponding to the fault previously chosen, and so on through to the final addition to the path. Thus this approach is more costly than using static relative size ranking, and may not successfully yield a recovery path in every case.

Chapter Ten

Conclusion

In this chapter, we summarize the results reported in this thesis and our contribution to state-of-the-art software reliability prediction technology. We also propose some future investigations using the debugging graph database.

10.1 Summary

In this thesis, we described the use of a data model called the debugging graph to investigate the potential for fault recovery order to affect the predictive accuracy of existing software reliability models. Our laboratory experiments led us to conclude that if one can choose a predictive model, control the fault recovery order and use the average of large samples for interfailure times, then one can derive more accurate predictions from existing algorithms.

We validated the use of the surrogate oracle and path truncation (i.e., data aging) techniques to make the data collection component of our investigations more feasible and manageable. In doing so, we acknowledged some limitations of using the surrogate oracle that need further investigation, but still provided a tentative avenue for moving our methodologies out of the laboratory and into the domain of software reliability engineering applications. Such a transition will enable further investigations involving new software specimens.

We showed evidence of the fault interaction phenomenon as it occurred naturally in data derived from our subject software. We advocated further investigations of this effect using the debugging graph as the basis for a data collection environment, in the hope of increasing our understanding of the phenomenon and supporting further improvements in the software reliability prediction process.

10.2 Contribution to Current Practice

The approach to predictive modeling we described differs in several significant ways from current practice. Engineers presently compare models' performance based on the ***ambient data*** — that is, single observations of time to failure are fed to the predictive models simply in whichever order the faults have been recovered during debugging. Using the averages of many failure trials and a standardized, size-based fault recovery order mitigates the variance and noise inherent in this ad-hoc approach, thereby helping us to intelligently choose a model and a path for improved predictive accuracy. The methodology proposed in this thesis in fact is independent of how faults are located; any combination of debugging techniques could be used and produce the same predictive results.

10.3 Future Directions

It is our hope that future studies at ODU and other research facilities will exploit the voluminous data now available for the LIC specimen to study fault interactions as another possible avenue for improvement to the software reliability prediction process. We suspect that the detection of discrepancies in the apparent failure rates attributable to known faults at multiple debugging graph levels is a "first-cut" indication of interaction effects. We also believe that by delving deeper into the collected data — to study, for example, how particular inputs produce oscillating patterns of successful and unsuccessful outcomes along a given debugging path through the graph — may enable progress in some of the following areas:

- determining which faults (repairs) are interactive;
- determining whether interaction types exist other than those directly attributable to logical code dependencies;
- establishing how to distinguish other types of interaction if they exist; and
- demonstrating how best to account for interaction "fuzziness" in the failure data.

The surrogate oracle is clearly useful in assessing relative bug sizes. Its primary limitation in the context of the prediction methodologies we propose is its assessment of perfect reliability at the end of the debugging path, regardless of how unreliable the program actually remains. We suggested some possible means for correcting the later reliability figures to account for optimistic bias prior to making predictions, but clearly further investigation of the surrogate oracle tool is a future research path.

We advanced some conjectures about the utility of data aging. Our experiments with the LIC software showed that often less data produces better predictions than using the cumulative failure history. Further studies should be conducted to see if this conjecture is justified and not just a coincidence or an artifact of the experimental data set. The associated issue is to determine if a small, fixed-sized window or some percentage of the known failure history recommends itself.

When software supports the random generation of input test cases and a robust gold or surrogate oracle can be constructed, as with LIC, the debugging graph data collection environment provides unique opportunities to study fail sets for anomalous behaviors. A control program similar to **LICCtrl** is easy to implement and facilitates the collection of failure data for further fail set analysis and validation of our methodologies with alternative specimen programs. We advocate validating the conclusions and conjectures advanced in this thesis with other LIC specimens as well as other bodies of software to incrementally advance towards the unified prediction methodology discussed in the previous chapter.

References

[1] Russell J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability and Safety," ACM Computing Surveys, vol. 22, no. 1, March 1990, pp. 35-68.

[2] Abdalla A. Abdel-Ghaly, P. Y. Chan and Bev Littlewood, "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 9, September 1986, pp. 950–967.

[3] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Reading, Massachusetts: Addison-Wesley Publishing Company, 1974.

[4] Farokh Bastani, "Foreward: Software Reliability," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, November 1993, pp. 1013–1014.

[5] Sarah Brocklehurst, P.Y. Chan, Bev Littlewood and John Snell, "Recalibrating Software Reliability Models," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, April 1990, pp. 458–469.

[6] G. Becker and L. Camarinopoulos, "A Bayesian Estimation Method for the Failure Rate of a Possibly Correct Program," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, November 1990, pp. 1307–1310.

[7] Susan S. Brilliant, John C. Knight and Nancy G. Leveson, "The Consistent Comparison Problem in N-Version Software," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, November 1989, pp. 1481-1485.

[8] Sarah Brocklehurst and Bev Littlewood, "New Ways to Get Accurate Reliability Measures," *IEEE Software*, July 1992, pp. 34–42.

[9] Fred Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, April 1987, pp. 10–20.

[10] Christopher Cowles, "Measuring Software Reliability Models," Master's Degree Project Report (unpublished), Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1991.

[11] Janet R. Dunham, "Experiments in Software Reliability: Life-Critical Applications," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, January 1986, pp. 110–123.

[12] Janet R. Dunham and John L. Pierce, "An Experiment in Software Reliability," NASA Contractor Report 172553, Software Research and Development Center for Digital Systems Research, Research Triangle Park, North Carolina, 1985.

[13] Narasimhaiah Gorla, Alan C. Benander and Barbara A. Benander, "Debugging Effort Estimation Using Software Metrics," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, February 1990, pp. 223–231.

[14] A. L. Goel and F.B. Bastani (eds.), *IEEE Transactions on Software Engineering (Special Issue on Software Reliability, Part 1)*, vol. SE-11, no. 12, December 1985.

[15] A. L. Goel, “Software Reliability Models: Assumptions, Limitations, and Applicability,” *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, December 1985.

[16] Herbert Hecht, “Measurement, Estimation and Prediction of Software Reliability,” under Contract NAS1-14392, NASA Langley Research Center, Hampton, Virginia, 1976.

[17] Z. Jelinski and P. Moranda, “Software Reliability Research,” in Statistical Computer Performance Evaluation, Walter Freiberger (ed.). New York: Academic Press, 1972, pp. 465–483.

[18] B. Littlewood and P.A. Keiller, “Adaptive Software Reliability Modelling,” *Proceedings of the 14th International Conference on Fault-Tolerant Computing*, 1984, pp. 108–113.

[19] Bev Littlewood, “How to Measure Software Reliability and How Not To,” *IEEE Transactions on Software Engineering*, vol. R-28, no. 2, June 1979, pp. 103–110.

[20] Douglas R. Miller, “Exponential Order Statistic Models of Software Reliability Growth,” CR-3909, NASA Langley Research Center, Hampton, Virginia, July 1985.

[21] Douglas R. Miller, “Making Statistical Inferences About Sfotware Reliability,” CR-4197, NASA Langley Research Center, Hampton, Virginia, December 1988.

[22] John D. Musa, Anthony Iannino and Kazuhira Okumoto, Software Reliability: Measurement, Prediction, Application. New York: McGraw-Hill Book Company, 1987.

[23] P. B. Moranda, "Prediction of Software Reliability During Debugging," *Proceedings of the Annual Reliability and Maintainability Symposium,* 1975, pp. 327–332.

[24] J. D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *7th IEEE International Conference on Software Engineering*, 1984, pp. 230–238.

[25] Phyllis M. Nagel and James A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling," CR-165836, NASA Langley Research Center, Hampton, Virginia, February 1982.

[26] Eldred Nelson, "Estimating Software Reliability From Test Data," *Microelectronics and Reliability*, vol. 17, no. 1, January 1978, pp. 67–74.

[27] P. M. Nagel, F. W. Scholz and J. A. Skrivan, "Software Reliability: Additional Investigations Into Modeling with Replicated Experiments," NASA Contractor Report 172378, NASA Langley Research Center, Hampton, Virginia, June 1984.

[28] Normal F. Schneidewind, "Software Reliability Model with Optimal Selection of Failure Data," *IEEE Transactions on Software Engineering*, vol. 19, no. 11, November 1993, pp. 1095–1104.

[29] R. S. Swarz, "Methodology for Software Reliability Prediction," WP-29170, The MITRE Corporation, Bedford, Massachusetts, 1990.

[30] Wing N. Toy, "Fault-Tolerant Computing," in Advances in Computers, Marshall Yovits (ed.)., vol. 26. New York: Academic Press, 1987, pp. 335–391.

[31] Richard L. White and Christine F. Harbison, "The Error Graph: Research in Software Reliability," Master's Degree Project Report (unpublished), Department of Computer Science, Old Dominion University, Norfolk, Virginia, 1987.

[32] Lee J. White, "Software Testing and Verification," in Advances in Computers, Marshall Yovits (ed.). vol. 26. New York: Academic Press, 1987, pp. 335–391.

[33] Larry Wilson and Wenhui Shen, "Software Reliability Perspectives," TR-87-035, Old Dominion University, Norfolk, Virginia, 1987.

[34] Larry Wilson and Wenhui Shen, "Simulation Studies of Software Models," TR-89-10, Old Dominion University, Norfolk, Virginia, 1987.

[35] Huang Xizi, "The Limit Conditions of Some Time Between Failure Models of Software Reliability," *Microelectronics and Reliability*, vol. 30, no. 3, May 1990, pp. 481–485.

Appendix A

Porting the LIC Test Environment

This appendix describes the work involved in porting the LIC test enviroment from NASA to the ODU Sun network, and eventually rewriting the basic control program. It is included for background information and instructive value to future data collection efforts.

A.1 Background

An initial problem faced in collecting the experimental data was porting the LIC test environment from NASA's AIRLAB to the ODU Computer Science Department Sun network. The body of code included: the *control program*, the main FORTRAN routine that executed test versions of the LIC software as subroutines and collected failure statistics; the *gold version* of LIC, which is believed to have perfect reliability; and several *partially debugged versions* of LIC, which are independently developed LIC software implementations with debugging histories. There were compelling motivations for us to carry through with this work despite the time and risk inherent in any porting task.

By porting the software to local Suns, we could exercise much greater control over scheduling data collection runs, disk space, etc.. Another attractive feature of the port was re-hosting the software onto the much faster Sun platform; this hopefully would enable us to accelerate the data collection and analysis work, which had proven to be cumbersome during some preliminary studies. Potential for increased data throughput likewise existed in that locally, it would be possible to spawn multiple collection and analysis processes and distribute them to idle machines on the network.

Perhaps the overriding concern in porting the software, however, lay in the realization that the AIRLAB LIC test environment had been modified a number of times over the past ten years to suit sometimes divergent goals of various research projects. The resulting software thus featured an inflexible, monolithic design with many undocumented features, as well as extensions that had tied it very closely to the VMS operating system, the AIRLAB environment, or particular researchers' specific data collection needs. We felt that in the process of porting the code, we would have the opportunity to wipe the slate clean and devise a much simpler, more portable UNIX-based test environment under SunOS whose behavior we understood completely. Furthermore, useful system documentation could be developed from the ground up to serve our own, as well as future, research needs.

A.2 Re-Hosting the Original Test Environment

Our initial concerns were three-fold: to physically relocate the software; to understand the software's current functionalities; and to identify suitable substitutes for VMS-specific features of the LIC test environment that would be needed even in a simplified test scenario in the new environment.

A.2.1 Recompiling the Software

Copies of all pertinent LIC software were moved via the file transfer protocol (ftp) from AIRLAB to ODU. Our next tactic was to see whether it were possible on the Sun host to recreate the same test environment that existed at NASA. Thus we sought to recompile the code into an executable module using the Sun FORTRAN 77 compiler, **f77**.

The gold version of LIC as well as the partially debugged versions were written in "standard" FORTRAN and thus could be recompiled easily using **f77**. The main routine, which we call the *control program (LICCtrl)*, however, would not yield so easily. Part of this difficulty lay in the fact that a number of "canned" library routines were used in the control program that were unavailable at ODU. Another problem was that some VMS FORTRAN extensions and system calls were integral to the operation of the code.

A.2.2 Library Dependencies

The control program relied on a number of FORTRAN subroutines from the **IMSL Statistical Library** to generate uniformly distributed pseudorandom numbers in the range [0,1]. The underlying uniform generator used in IMSL is the multiplicative congruential method, which has the form:

$$x_i = c * x_{i-1} \bmod(2^{32}- 1)$$

IMSL offers the choice of various values for c which maximize the period for the generator cycle and result in a close approximation of a true uniform distribution.

The particular IMSL routines accessed from the control program were:

• **RNOPT**	selects one of 3 possible *c* values;
• **RNSET**	initializes the seed used in the generator;
• **RNGET**	retrieves the current seed value; and
• **RNUN**	retrieves a block of random values in sequence

Unfortunately, the IMSL package was not available on ODU's Sun network, so a suitable alternative approach had to be identified.

The **FORTRAN Library Routines** on the Sun offer a function called **rand** that returns real values in the range [0,1]. This is done via a non-linear feedback random number generator which, in the form used in the library, claims to produce a uniformly distributed sequence of random values having a period greater than 2^{69}. Calling **rand** with any argument greater than 1 supplies a new seed for the random number generator along with the first random value; this functionality is nearly the same as **RNSET**. Calling **rand** with an argument of 0 returns the next random value in the sequence, which could be used to mimic the loading of an array of values by **RNUN**.

The purpose of using **RNGET** was to ensure that a run could be resumed at some intermediate place in a particular random number sequence. This could be emulated through **rand** only by knowing the initial seed of that random sequence and the position of the desired element in the resulting sequence. One could then reinitialize **rand** with the appropriate seed and "skip over" the intermediate values to get back to the same place in the sequence. The invocations of the IMSL routines were replaced with equivalent **rand** calls before continuing the porting effort further.

A.2.2 VMS Dependencies

A.2.2.1 Syntax Problems

Some VMS-specific problems were merely syntactic extensions. For example, in the following expression:

```
include 'param.for/list'
```

attaching the "/list" extension is a VMS-ism disallowed by standard FORTRAN 77. This could be filtered out by using a VMS-tolerant switch, "-xl," on the **f77** compile line.

Other VMS-isms could be tolerated by the compiler with the VMS switch on, yet caused runtime errors. One such feature is underlined in the following file **open** statement:

```
open( lout2, organization = 'relative', access = 'direct', status = 'new',recl = reslen )
```

The **organization** option is not available as part of standard FORTRAN 77. So even though the f77 compiler would tolerate such statements when the VMS switch was used, the first attempt to open a relative file from an executable caused the process to abort.

The other atypical aspect to each **open** statement used in the LIC control program was that the file to which the logical unit number ("lout2" in the above example) should attach was not explicitly named. The appropriate file names were provided under VMS as environment variables that were established prior to invoking the program. Although the latter behavior could be mimicked under SunOS using **setenv** and **ioinit** calls to the **FORTRAN Library Routines**, there did not appear to be any easy way to eliminate the use of relative file organization without substantially rewriting certain portions of the code.

A.2.2.2 System Calls

The final VMS dependencies in the LIC test environment involved system calls to routines named **ESTABLISH** and **UNWIND**. In the existing LIC data collection scenario, one or more gold version LIC programs and one or more partially debugged versions of LIC were iteratively run as subroutines of the control program. Any of these subroutines, some of which were specifically intended to be "buggy," could result in an abnormal termination of the control program. This was handled under VMS by having the control program call **ESTABLISH** to assert a general purpose, user-defined exception handling routine prior to running any of the test subroutines.

Once **ESTABLISH** had been called, any raised exception resulted in the invocation of the user-defined handler in front of the native VMS error handler(s) that normally would be triggered by the exception(s). In the LIC test environment, if any subroutine failed, the handler it defined would set a status variable for the control program to examine later indicating that the subroutine had aborted. The handler would then disable any pending native exception handler invocations and use **UNWIND** to remove frames from the stack. This had the effect of poising the controller to talley that subroutine execution as an abort case and to move on to the the next gold or partially debugged subroutine call upon returning from the handler.

No direct counterparts of **ESTABLISH** and **UNWIND** exist under SunOS, although similar behaviors can be emulated. Using multiple invocations of the system library function **signal**, it is possible to enumerate an exception handler for individual signal (exception) types. Naming the same handler for all signal types would result in the same net effect as the VMS **ESTABLISH** call. A **signal** interface is provided in the Sun's **FORTRAN Library Routines**. To effect the same results as the handler written for the LIC controller on VMS would involve using two system library functions, **setjmp** and **longjmp**. These functions respectively retain a given program context and allow returning to that context from an arbitrary location elsewhere in the program. Thus, **setjmp** could be called at a reasonable point before invoking a subroutine. In case of subroutine failure, the user-defined handler could, as its last action, perform a **longjmp** back to the saved location. Unfortunately, neither interface is provided in the **Sun's FORTRAN Library Routines**.

A.3 A New Experiment Scenario

A.3.1 Goals

After identifying the essential characteristics required in the LIC data collection environment and the areas of the NASA code that were more challenging to port, we changed our focus to creating new control program instrumentation. We had three primary goals in mind for the new software:

- a clean, simple data collection program that could be run with human interaction or as an unattended process;
- support for partitioning the data collection and analysis work into smaller tasks that could be distributed to various processors on the network, thereby reducing the calendar time required to complete the study; and
- economy of statistics in initial runs, coupled with run repeatability at a later time to collect more detailed data if needed.

The original test and gold programs were written to run as FORTRAN subroutines which would read their input values from a common block and write their calculated output values to another common block. To avoid modifying the actual test software in any way, we decided to implement the control program again in FORTRAN 77, although it was not our language of preference. We hoped this would help to avoid any potential data alignment or language incompatibility problems that might have resulted had we implemented the control program in C, for example, and "faked" the manipulation of the common blocks prior to and after calling the subroutines. Certainly an easy conversion to another programming language from FORTRAN could be made in the future.

We formulated a simplified view of the desired control program process as follows:

```
read runtime parameters;
for( desired number of cases ) {
    generate next set of input values;
    load common block;
```

```
        call gold subroutine;
        load common block;
        call test subroutine;
        compare gold and test results;
        talley;
}
output summary statistics;
exit(0);
```

A.3.2 Initial LICCtrl Implementation

The most technically interesting aspect to the control program was allowing for and detecting disagreements between the gold and test version programs. Especially critical was the fact that previous studies had detected actual aborts of the test and/or gold subroutines, which we wished to intercept and talley. As explained above, under the system calls **ESTABLISH** and **UNWIND** had been used under VMS for this purpose by enabling the control program to return to a saved context after unexpected failures of the subroutines. Since the similar pair of C library system calls named **setjmp** and **longjmp** were not directly available in Sun's FORTRAN library, we initially tried to implement the control program under SunOS by using the available system calls **fork**, **signal** and **wait** to detect such failures.

A.3.2.1 Implementation Description

An invocation of **fork** creates a new process, called the *child*, which is more or less an exact copy of the creating process, called the *parent*. A successful call to **fork** returns the value zero to the child process, and the (positive, non-zero) process identifier of the child to the parent. After forking, the two processes continue to run concurrently from that common point of execution. In this way, a block of code was included in the control program around each critical subroutine call — in this case, the gold and the test subroutines — which specified certain actions to be taken by the child process (when **fork** returned 0) and others to be taken by the parent process (when **fork** returned a positive value).

The basic behavior of the child portion of the code was to call either the gold or the test subroutine and, if successful, return the calculated output values to the parent process. On the other hand, if the subroutine terminated abnormally, via prior invocations of the system routine **signal** we arranged that a simple signal handler would cause a non-zero return status from the child to the parent process. By iteratively calling the system routine **wait**, the parent portion of the code could poll the system for the termination of the child process. The return status could then be examined to determine if the subroutine had failed — that is, the exception handler had been invoked. In this case, the parent could talley that iteration as an abort for either the gold or the test version of the program as appropriate. Otherwise, the parent could obtain the output values from the invoked subroutine for comparison against the corresponding test or gold version outputs.

A.3.2.2 Implementation Limitations

An unfortunate limitation of the **fork** mechanism is that, although the state of the parent process is known to the child at the time of its birth, any changes subsequently made to state variables by the child are unknown to the parent. Thus, data that the test or gold subroutines loaded into common blocks as calculated output values were inaccessible to the parent process. A normal mode of communication between parent and child is to establish a ***pipe***, which is an i/o mechanism for creating a private, interprocess communication (ipc) channel into which the child can dump data for subsequent processing by the parent, or vice versa. However, the pipe mechanism was not available in Sun's FORTRAN library, so a pipe was not an immediate option for the child to communicate its calcuated output values to the parent. Thus we resorted to using a simple data file into which the child wrote its results prior to exiting and from which the parent read the child's results after the child's death.

The overall logic of this approach can be summarized as follows:

```
read runtime parameters;
enable signal handler;
for( desired number of cases ) {
        generate next set of input values;
        load common block;
        fork a child process;
        if( child_process ) {
                call gold subroutine;
                output results to file;
                exit(0);
        }
        else if( parent_process ) {
                wait for child process to terminate;
                if( ! termination_normal )
                        talley as gold abort;
                else
                        read gold results from file;
        }
        else {
                error — fork failed;
```

```
            exit(-1);
        }
        load common block;
        fork a child process;
        if( child_process ) {
            call test subroutine
            output results to file;
            exit(0);
        }
        else if( parent_process ) {
            wait for child process to terminate;
            if( ! termination_normal )
                talley as test abort;
            else
                read test results from file;
        }
        else {
            error — fork failed;
            exit(-1);
        }
        compare gold and test results;
        talley;
    }
    output summary statistics;
    exit(0);
```

A.3.2.3 Observations

We successfully implemented a new control program using this approach. Unfortunately, it was very slow, requiring nearly 30 hours (wall-clock time) to collect comparisons of 100,000 cases of a single test and gold version subroutine. The projected time for 1 million cases was therefore over 12.5 days. We felt these run times were unacceptable for two reasons. First of all, they were slower than the run times of the control program at NASA on much slower hardware. Secondly, it seemed unlikely that we could expect nothing to go wrong with the host network during such a long period of time, implying that collecting 1 million cases would require numerous restarts of the control program and substantial data management effort.

A.3.3 Modified Experiment Procedures

To ameliorate this situation, the software was moved onto a processor that had the fastest available CPU on ODU's Sun network and a large amount of local memory — an ideal host for the computation-intensive LIC process. Two simple optimizing strategies were exploited on this host.

First of all, the small file that **LICCtrl** was using to mimic the piping mechanism between a parent and child process was moved to the **/tmp** directory to keep it in faster RAM, thereby avoiding costly disk accesses. Secondly, publicly accessible disk space on the host enabled running the software and recording statistical data locally on the host rather than in a temporarily mounted directory across the network file service. In this way, the run time of a 100,000 iteration process on the machine with local disk was reduced to about eight hours under ideal conditions (i.e., no other processes competing for CPU time), and to approximately 20 hours on machines with non-local disk

We also tried generating the gold program outputs in advance and reading them into the **LICCtrl** program in lieu of calculating them in real time. This offered virtually no run time improvement, implying that the time to do a disk access to obtain the previously generated gold program outputs was roughly equal to the time required to run the gold subroutine, output its results to RAM, and read them back into the control program. We concluded that the cumbersome interprocess communication and the context switching between the parent and child processes were consuming most of the control program's execution time.

We performed some baseline runs, and found that the new statistical data compared favorably with those collected in the previous experiments. No **LICCtrl** aborts were observed, and runs were found to be repeatable; that is, the same partially debugged LIC version produced the same failure cases each time it was run against the gold version with the same input stream. Thus, we felt reasonably confident that our instrumentation, although slow, was working properly and we could confidently continue with optimization.

A.3.4 Code Optimization Areas

While benchmarking the code, we noted that the new control program instrumentation detected no aborts for either the gold or the test subroutines. This observation led us to more thoroughly investigate floating point exceptions and signal handling under SunOS. We also felt that the speed improvements that we were gaining from manipulating the run time environment were relatively small and limiting us to a small number of specially configured host machines. For that reason, we realized that some code changes would be necessary to reduce the software's execution time to within reasonable bounds.

A.3.4.1 Floating Point Exception Handling

Although the newly calculated reliablity figures were in line with previously observed data, the absence of unrecoverable errors (aborts) in the tested software seemed counter-intuitive. That is, disagreements were being detected, but none were due to outright failures of test program variants that had been intentionally seeded with known bugs, some of which involved substantial numerical errors. Through some additional research, we found that Sun makes a distinction between the occurrence of a floating point exception (such as divide by zero, overflow, underflow, etc.) and the physical generation of the floating point signal SIGFPE.

Sun's basic philosophy is that most users are not interested in most signals and, in particular, the floating point exception signal will only be generated if it has been explicitly enabled by invoking **ieee_handler**. The simple signal handler we wrote for the new **LICCtrl** software had defined a behavior on detecting SIGFPE, but, because we naively failed to enable that signal using ieee_handler, such problems were never being detected. In this circumstance, the test subroutine was clearly running to completion and simply "getting the wrong answer." No aborts (i.e., SIGFPEs) were being detected because the operating system was ignoring floating point exceptions. Fortunately, interfaces to the IEEE mechanisms were available in Sun's FORTRAN library to remedy the problem through invoking the **ieee_handler** function.

A.3.4.2 Ipc Bottleneck

Our second concern was the required execution times for the control program. The potential to talley 100,000 cases in eight hours by the "improved" control program was more attractive than the 30 hours required of the "initial" version, yet the projected time for 1 million cases was still untenable. Several documented bugs in the test software were known to require at least 1 million cases to get an adequate reliability estimate. Only one computer was available on the host network with locally writeable disk. Even running continuously, data could be collected using that single platform at the rate of only three nodes per day (24 hours / 1 day * 1 node / 8 hours). Given a collection of ten known bugs, the debugging graph we wished to investigate contained 1024 nodes (see 3.2.2.2). This would require nearly a year of collection time — 341.3 days (1 day / 3 nodes * 1024 nodes) — for just a minimum of 100,000 cases per node. Adding additional, slower machines would improve this figure somewhat, but at the increased effort to monitor and restart jobs that died due to network failure.

Clearly we needed substantial improvement in the control program's execution times. In particular, we needed to focus on eliminating the clumsy ipc required by the forking mechanism we were using to detect subroutine failures. This led us to investigate means of accessing some of the other SunOS system functions for coordinating signal detection and exception recovery from FORTRAN. Such interfaces are available by using the PRAGMA directive to the f77 compiler.

A.3.4.3 The PRAGMA Directive

PRAGMA allows a FORTRAN module to interface to library functions written in other programming languages, such as C or PASCAL. We initially used PRAGMA to interface to the pipe mechanism in the C library in an attempt to avoid the awkward file-based ipc between the parent and child control programs. This proved to be a blind alley. When a pipe is instantiated, the operating system returns two file pointers — *unit numbers* in FORTRAN jargon — to the invoking program. One of these pointers references the "read" end of the pipe, while the other references the "write" end of the pipe, but both reference the same i/o unit. However, when FORTRAN is provided with two distinct unit numbers, in this case associated with an unnamed, private channel, it creates two physical files called "fort.x" and "fort.y," where *x* and *y* are the unit numbers. Now suppose the child process writes to the pipe using the appropriate unit number, say y. The effect is to write to a file called "fort.y." Then when the parent attempts to read the child's output using the unit number *x*, it is actually accessing a completely different file called "fort.x," which is empty. Thus, the pipe mechanism could not be used from FORTRAN to suit the purposes of **LICCtrl**.

We were successful, however, in designing a way to eliminate reliance on the sluggish fork mechanism and the file-based pipe by using PRAGMAs to access the **setjmp** and **longjmp** C library routines. The **ieee_handler** can be used to nominate an exception handler routine that intercepts raised floating point exceptions and returns to a known program context by using a combination of setjmp and longjmp. The effect of **setjmp** is to save a current program context into a buffer. A normal call to **setjmp** returns zero to the invoking program; non-zero is returned otherwise. Thus, a call to a critical section of code can be surrounded by an if block involving "normal" actions to be taken when **setjmp** returns zero, and "recovery" actions to take when **setjmp** returns non-zero. If something abormal occurs in the critical section of code, the exception handler routine is invoked. It is intentionally encoded to do a **longjmp** back to the context saved by **setjmp** and to activate the recovery actions. Because a single process can be used, the main routine and subroutines all share the same data space; interprocess communication is no longer an issue.

A.3.5 Optimized LICCtrl Implementation

In the case of the control program, a signal handler was written that logged to a journal file the type of floating point exception raised, and returned control to a known program context. This signal handler was nominated using **ieee_handler**. The **setjmp** function was used to surround the calls to the gold and test subroutines. The ***setjmp-returns-zero*** block of the statement handled the case that the test or gold subroutine was called and no exceptions were raised. The calculated output values were simply read from the designated common block in which the subroutine stored them. The ***setjmp-returns-non-zero*** block of the statement handled subroutine failures by talleying them as aborts. This portion of code was activated only if the gold or the test subroutine failed, implying the exception handler had been invoked. As its last action, the handler was designed to call **longjmp** to restore the program context that had been previously saved by **setjmp** just prior to the test or subroutine call. This caused the return to the control program to look as though that same call to **setjmp** had returned a non-zero value, thereby activating the recovery actions. The overall implementation logic used in the optimized LICCtrl software can be summarized as follows:

```
read runtime parameters;
use ieee_handler to nominate "handler" as the SIGFPE handling routine;
for( desired number of cases ) {
        generate next set of input values;
        load common block;
        if( ! setjmp ) call gold subroutine;
        else talley as gold abort;
        load common block;
        if( ! setjmp ) call test subroutine;
        else talley as test abort;
        compare gold and test results;
        talley;
}
output summary statistics;
exit(0);

handler {
        log signal raised;
        longjmp;
}
```

In this way, we reduced the **LICCtrl** execution time on a machine with no local disk to approximately four hours for 1 million cases. By comparison, when previously using the **fork** mechanism, this same sized run would have required 80 hours (8 hours / 100,000 cases * 10) on a machine with local disk, or 50 times as long (20 hours / 100,000 cases * 10 = 200 hours / 1 million cases * 1 million cases / 4 hours) on a machine with no local disk — an order of magnitude improvement. At this point, we felt the **LICCtrl** executable was running fast enough to start massive data collection.

Appendix B

Description of the LICCtrl Program Interface

This appendix describes the functionality of the control program, **LICCtrl**, used in the experiments reported in the thesis.

B.1 LICCtrl Capabilities

LICCtrl, the control program used in our experiments, was designed to serve several purposes. While its primary function is to run a gold version of LIC and a partially debugged version of LIC in tandem to calculate a reliability figure on the latter, a number of additional capabilities seemed desirable. In particular, when disk space is not a problem, it is possible to use **LICCtrl** to produce "canned" results to be reused in subsequent runs of the program. For example, the user may wish to can the gold outputs to avoid having to execute the gold version over and over again in tandem with different partially debugged programs using the same input data sets. Or, the input data sets may be saved to a file to avoid having to recalculate them in later runs, or to allow examining them for statistical analysis.

Among the functionalities of **LICCtrl** are the following:

- generate *n* sets of randomized input values for (x,y) coordinates, the logical connector matrix and the preliminary unlocking matrix diagonal according to value distributions defined in the LIC problem specification;
- generate *n* compressed output values from the gold version of LIC;
- generate *n* compressed output values from a partially debugged version of LIC;

- produce summary and/or detailed statistics from running the gold version and partially debugged version in tandem;

- run in *silent, unattended* mode or *verbose, interactive* mode.

B.2 LICCtrl Limitations

B.2.1 Compilation Requirements

A limitation of the **LICCtrl** program is that it is necessary to create a new executable of the program for each distinct partially debugged LIC implementation that one wishes to run in tandem with the gold version. This is done by putting a FORTRAN subroutine ***wrapper*** called "TESTPGM" around the partially debugged LIC implementation and one called "GOLDPGM" around the gold implementation. The corresponding source files are then included on the **f77** compile/link line with the other **LICCtrl** modules. This process is repeated for each different partially debugged version subroutine, presumably using the same gold version subroutine in each case.

This limitation is an artifact of two original requirements of the LIC program; first, that it be encoded in FORTRAN; and secondly, that its input data be loaded into a COMMON block prior to a run and its outputs be available from a COMMON block after the run. In order to protect the integrity of the LIC module, it is minimally necessary to supply some kind of additional code — the wrapper — around the gold or partially debugged version to load data into and read data from the COMMON. This purpose is served in FORTRAN by running the LIC versions as SUBROUTINEs, while the PROGRAM routine we call **LICCtrl** does, among other things, the common block accesses prior to and after each gold or partially debugged subroutine run.

It is therefore necessary to explicitly link two subroutines with known, declared names into the **LICCtrl** program so that they may be executed during the runs. To simplify matters, we decided to fix the names of the two subroutines since we would have to apply the subroutine wrapper around the code to be tested anyway. This avoids potentially having to edit the subroutine CALL statements in each **LICCtrl** version, which would have to be done if arbitrary subroutine wrapper names were used. Creation of the various **LICCtrl** instantiations and subsequent executions can be managed quite easily by makefiles and shell scripts.

B.2.2 Input Data

In its current configuration, an execution of the **LICCtrl** program that generates input data sets for immediate or later use invokes a native UNIX random number generator, **rand**. The function **rand** will produce the identical sequence of values on distinct **LICCtrl** invocations, provided the same initial seed is used. This is desirable, since it supports repeatability in the testing process. Additional LIC inputs, called *runtime parameters*, were provided in the problem specification and are passed to the executable via an input file.

The default behavior of **LICCtrl** is to use the same initial random seed (1) anytime that **rand** is needed. However, **LICCtrl** does allow the specification of an alternate seed via a command line argument ("-s" discussed below). Likewise, the user of **LICCtrl** supplies the name of the runtime parameter file as part of the **LICCtrl** command sequence. The flexibility of this interface, while convenient, can cause problems if the analyst is not careful.

When comparing data collected during separate **LICCtrl** program invocations, the analyst must be assured that the same input data were used. This ultimately traces back to the same initial seed to **rand** and the same runtime parameters. One way of assuring this is by canning the input data sets and then using the command line arguments "-i" and "-p" (discussed below) to make sure the same data are used in all the collections.

As an additional safety measure, **LICCtrl** records, as part of the statistical output of a tandem run, the seed that was used in the original random sequence that gave rise to the input data sets. The name of the runtime parameter file is also logged. Canned gold outputs, partially debugged outputs and input data sets are likewise labelled with the information used in their generation. Although **LICCtrl** attempts to ensure integrity of the runs by verifying that the use of seeds and runtime parameters is consistent, the analyst should still carefully verify that only outputs and statistics from **LICCtrl** invocations using the same input data are compared.

B.3 LICCtrl Runtime Options

Following UNIX conventions, eleven different command line arguments may be used to select combinations of the various **LICCtrl** functionalities. These single character commands preceded by a dash ('-') are known as ***switches***. One or more switches are supplied on the command line after the executable name to direct the **LICCtrl** processing activity. The meaning of a switch may depend on which other switches have been simultaneously specified. General descriptions of their uses are enumerated in the following list:

- **Run Mode: -d**

Specifying the "-d" switch (for "debug") causes **LICCtrl** to run in verbose, interactive mode. Silent, unattended mode is the default behavior. In interactive mode, certain error conditions that might otherwise cause **LICCtrl** to abort may be overriden. For example, if the attempt to create a new output file for recording run statistics reveals that the file already exists, interactive mode will offer the opportunity to overwrite the existing file or supply a new file name; whereas in silent mode, the execution will fail at this point with an appropriate error message and status. Interactive mode also echoes output to the screen at various checkpoints throughout the execution, which is not done when **LICCtrl** runs unattended.

- **Gold Output File: -g <filename>**

Used to name a file from which compressed outputs from the gold version of LIC may be read or a file into which such outputs should be written.

- **Input Data File: -i <filename>**

Used to name a file from which sets of randomized input data for LIC may be read or a file into which generated input data sets should be written.

- **Run Label: -l <string>**

Used to supply a user-defined title to label output file contents. Note that if the <string> contains blanks, it should be enclosed in quotation marks.

- **Run Size: -n <cases>**

Used to specify the number of runs to perform or the number of data sets to generate. Note that the value of <cases> should be a positive, non-zero integer.

- **Run Output File: -o <filename>**

Used to name the file into which output statistics from tandem **LICCtrl** runs should be written.

- **Runtime Parameters File: -p <filename>**

Use of this switch is mandatory, unless **LICCtrl** is only being used to "can" input data sets. The file referenced by the argument contains the LIC runtime parameters as described in the problem specification.

• **Resume Run: -r <case>**

Supply an integer case number that specifies the number of cases in the input data set sequence that should be skipped during this execution of the control program. That is, if "-r 100" is specified, then the first 100 input data sets are skipped over; the 101st input data set is the input for the first case executed.

• **Random Number Generator Seed: -s <seed>**

Supply an integer seed whose value is greater than 1 and less than MAXINT. The seed is used to initialize the UNIX random number generator **rand** that is used by the input data set generation routine. If no seed is supplied on the command line, a seed of "1" is used by default. It should be noted that the same seed (i.e., same input data sets) must be used across all tandem runs if the analyst wishes to conduct fail set studies.

• **Partially Debugged Output File: -t <filename>**

Used to name a file containing compressed outputs from the partially debugged (or "test") version of LIC or a file into which such outputs should be written.

• **Detailed Statistics: -v <increment>**

Specifying the "-v" switch (for "verbose") causes detailed statistics to be calculated and included in the output during tandem runs. The integer increment supplied determines how often intermediate summary results are posted. Useful runs typically involve tens of thousands of cases. The executing control program may die for any variety of reasons, including network failure, accidental process termination by an outsider, etc. Thus, information about specific bit-wise disagreements between gold and partially debugged output data, which case numbers aborted, and so on, is recorded for each disagreement or abort during the execution. Posting intermediate summary results at specific intervals allows partial results from a run that terminates prematurely to be retained. These can then be collated with results from a resumed run (see "-r" above) that completes the data for the remainder of cases after the last data posting.

B.4 Command Line Examples

A basic rule of thumb for using the **LICCtrl** command line arguments is the following:

The presence of the "-o" switch implies that a tandem run of gold and partially debugged LIC versions will ultimately take place, subject to successful combinations of other command line arguments. Use of the "-g" and/or "-i" options in the absence of the "-o" switch implies the creation of canned gold output and/or canned input data sets, respectively.

The following examples interpret the command line arguments for several different invocations of **LICCtrl**.

- **LICCtrl -n 1000 -i lic.dat -l "Standard 1000 Run"**

Create a file called **lic.dat**, and write into it the first 1000 generated input data sets. Label the output file contents as "Standard 1000 Run." If a file named **lic.dat** already exists, the program invocation fails.

- **LICCtrl -n 1000 -i lic.dat -d -s 12345**

Nearly the same as the previous example, except that if **lic.dat** already exists, the user may optionally overwrite it or supply a new file name. Also, the value 12345 is used to seed the random number generator in lieu of the default seed value.

- **LICCtrl -i lic.dat -g gold.out -p runparam.dat**

If a file called **lic.dat** does not exist, or the file **runparam.dat** does not exist, or the file **gold.out** already exists, the program invocation fails. Otherwise, run the gold version of LIC on each input data set contained in **lic.dat**. Write the compressed output from each gold run into the file **gold.out**.

- **LICCtrl -i lic.dat -g gold.out -p runparam.dat -d**

Same as the previous example, except that if there are problems with the existence of any of the files, the user will be offered the opportunity to overwrite existing files or provide alternate file names as appropriate.

- **LICCtrl -n 1000 -i lic.dat -o summary.dat -p runparam.dat**

If a file called **summary.dat** already exists, or **runparam.dat** does not exist, the program invocation fails. Otherwise, **summary.dat** is created, and one of the following interpretations is implied for the remaining switches:

If a file called **lic.dat** already exists, it is presumed to contain previously generated input data sets. Run the gold and partially debugged programs in tandem, using the **lic.dat** file contents as input. Do 1000 runs, or continue running until the contents of **lic.dat** are exhaused, whichever occurs first. Write summary statistics to file **summary.dat**.

If a file called **lic.dat** does not exist, create one. Then make 1000 tandem runs, saving the generated input data sets to the file **lic.dat**. Write summary statistics to file **summary.dat**.

- **LICCtrl -n 1000 -i lic.dat -o summary.dat -p runparam.dat -v**

Same as above, only along with the summary statistics, include detailed statistics in the output file **summary.dat**.

- **LICCtrl -i lic.dat -g gold.out -t test.out -o summary.dat -p runparam.dat**

If a file called **summary.dat** already exists, the program invocation fails; otherwise, the file is created. Several interpretations are possible from this point:

If both **gold.out** and **test.out** exist and were generated from the same input data, no further LIC executions are needed. Just calculate and write summary statistics to file **summary.dat**. The files **lic.dat** and **runparam.dat** are effectively ignored in this case.

If one of the files, say **gold.out**, exists, and **lic.dat** contains the same data that generated **gold.out**, then the partially debugged version is run using **lic.dat** as input. Its results are compared to those in **gold.out** and summarized in **summary.dat**.

If one of the files, say **test.out**, exists, and **lic.dat** was not used to generate **test.out**, then **lic.dat** is ignored. Provided a suitable runtime parameter file is available — either the one specified on the command line or the one named in **test.out** — the random input data sets used to generate **test.out** are recreated in real time as inputs for the gold version. Results from the gold and partially debugged versions are then are compared and summarized in **summary.dat**.

Appendix C

Test Environment Configuration

This appendix explains some conventions we adopted with respect to the automation aspects of the study. We also describe instrumentation we developed to assist with the generation and management of the partially debugged program variants and the results of running the control software. This includes shell scripts for contructing and navigating through the test environment, as well as the gold and faulty versions of LIC (**bbgold.for** and **Probla.for**, respectively) used in the experiments.

C.1 Terminology

The rows, or *levels*, of the debugging graph can be numbered from 0 to *n*, where *n* is the number of *bugs* in the software for which repairs, or *fixes*, are known.

C.2 Bug Identification

To construct a debugging graph for experimental purposes, we needed to identify a suite of known bugs and fixes for the LIC solution named **Probla**. The debugged version of **Probla**, which contained annotations of the twelve fixes identified during an earlier study, was examined along with the original unrepaired Probla source code. Analysis of the identified bugs revealed that two of them, those labeled bug 1 and bug 7, were side-effects of the original control program test environment rather than flaws in the LIC programming solution studied during past reliability experiments.

The control program was designed to run the gold and partially debugged programs as subroutines. Input values to run these subroutines were passed to them from the control program through common blocks. In the original control program instrumentation, a common block was incorrectly used, resulting in a failure rate of over fifty percent. In fact, during a 100,000 case run of the original, unrepaired software with an early version of LICCtrl, we found that 57,017 cases failed, resulting in a reliability figure of only 42.9830%. This bug was designated as bug 1. The fix used for bug 1 was an imperfectly done repair. We found that installing its corresponding fix in the original **Prob1a** source code caused the reliability of the software to degrade to 42.6390% for a 100,000 case run.

A second bug designated as bug 7 was actually the correction of bug 1's incorrect fix. Installing fix 7 in addition to fix 1 improved the reliability of the software to 42.9930% for a 100,000 case run. Since bugs 1 and 7 were actually artifacts of the original test environment, we decided that they were uninteresting for the purposes of our study. Thus, we considered the program variant at level 0 of this debugging graph to be Prob1a with both fixes 1 and 7 installed. We then re-numbered the remaining bugs identified in the earlier study as shown in Table C-1. These are the bug numbers that we will use in the remainder of the discussion and data collections for **Prob1a**.

C.3 Naming Conventions

An orderly scheme was needed to keep track of the numerous program variants and the data generated during the many runs of the control program.

Table C-1. Revised Bug Numbers for Prob1a

Original Number	*Revised Number*
1	*
2	1
3	2
4	3
5	4
6	5
7	*
8	6
9	7
10	8
11	9
12	10

** denotes fix pre-installed at level 0*

C.3.1 Program Variants

The naming conventions we adopted for the partially debugged program variants were directed by the following rules: use the name "**fix**" or "**bug**," followed by an enumeration of the fixes or bugs installed in the source code in ascending integer order, separated by underscore characters ('_'). Of course, all module names ended in the "**.for**" suffix.

We chose two possible name formulations for the partially debugged software modules in order to have the option of keeping the names of the program variants short but still meaningful. For example, suppose there is a pool of ten known bug fixes. Then for a program variant in which all fixes *except* those for bug 2 and bug 5 were installed, we might want to use the name

bug2_5.for

instead of the longer name

fix1_3_4_6_7_8_9_10.for

although, under our naming conventions, the two modules would contain the same source code. In practice, we adhered to the "fix" naming convention since future studies may try to expand the pool of known bug fixes.

The gold version LIC module used in the study was called **bbgold.for**, the name it was given during a previous research effort. The failure data for each program variant — that is, the case numbers that showed disagreement between the gold and the tested software versions — were stored in files called **LICmonitor.tmp**, while the summary statistics detailing run times and the calculated reliability figures were stored in files called **test.dat**. A small number of runtime parameters needed to initialize data used in LIC calculations were stored in a file called **runparam.dat**. These values were taken from the previous studies and were held constant for all cases.

C.3.2 Directory Organization

The initial stage of our study was targeted at generating a small but significant debugging graph in its entirety. For Prob1a, with the ten known bugs we felt were significant to study, this meant a minimum of 1024 (2^{10}) program variants would have to be created and run against the gold version program. Pertinent information about failure behavior also would have to be retained for later analysis in such a way that they could be traced to the appropriate program variants and cases run. We decided initially to use a directory-naming convention to keep track of the data for corresponding program variants.

A root directory called **FIX** was created for representing level 0 of the debugging graph — that is, the software with none of the ten (revised) fixes installed. We created and moved to this directory the following code:

- **fix0.for**: source code equivalent to the original Prob1a module (with the original fixes 1 and 7 installed);
- **bbgold.for**: the gold version LIC module;
- other FORTRAN code needed to create the executable control program;
- **fortLC**: a shell script used to compile all the FORTRAN code except fix0.for;
- **loadLC**: a shell script used to compile fix0.for and link it with the other FORTRAN object modules (see below);
- **LICCtrl**: the executable control program;

After running the data collection for the control program for this program variant, the two files **LICmonitor.tmp** and **test.dat** as described above were present in this directory.

For subsequent levels of the graph, we created subdirectories for representing the various graph nodes. For example, in the ten-bug case, the **FIX** directory had ten subdirectories:

FIX1, FIX2, FIX3, FIX4, FIX5, FIX6, FIX7, FIX8, FIX9, FIX10

Taken together, directories **FIX/FIX1**, **FIX/FIX2**, . . ., **FIX/FIX10** represented level 1 of the debugging graph. Each one contained a partially debugged program variant with exactly one fix installed — fix 1 in the case of **FIX/FIX1**, fix 2 in the case of **FIX/FIX2**, and so on. After running the collection effort for these variants, the corresponding data in **LICmonitor.tmp** and **test.dat** were stored in the appropriate corresponding subdirectories.

Subsequent subdirectories for representing levels 2 through 10 did not have to be so extensively subdivided. That is, with N known bugs, a directory name ending in **FIXn**, $1 \leq n \leq N$, needed only to contain subdirectories named **FIXm**, where $n+1 \leq m \leq N$; that is, subdirectories for the fixes having numbers higher than n. In this manner, data about the combinations of fixes that would have otherwise been represented in subdirectories named **FIXp**, where $1 \leq p < m$, would already have been represented in previously created subdirectories at the same level.

For example, look at Figure C-1, a small example directory structure for $n = 4$ known bugs. The directory representation is the traditional top-down, tree. As one moves from top to bottom down a column representing one level in the directory structure, it is easy to see that subdirectories towards the top of the level column have already captured the representation for some of the fix combinations that might otherwise have (redundantly) appeared as subdirectories towards the bottom of the column at the same level. For example, there is no need to create a directory **FIX/FIX2/FIX1**, since **FIX/FIX1/FIX2** has already been included to contain data about the variant having both fixes 1 and 2 present.

So that the directory structure appeared complete for subsequent navigation requirements of various tools, however, we used the **ln** system command to create **symbolic links** within levels from subdirectories that "actually were" there to subdirectories "could have been" there, but would have held redundant data. This implies, for example, that FIX/FIX2 had a symbolic subdirectory called FIX1, which was linked to the physical occurrence of directory FIX/FIX1/FIX2. Figure C-2 is the augmentation of Figure C-1 with these symbolic subdirectories included. A symbolic directory is represented as an italic named joined to a parent directory with a dashed line. In each case, the "imaginary" directory is actually just a pointer to the directory with the cumulative fix numbers listed in ascending order.

C.4 Shell Scripts

C.4.1 Source Code Creation

From the original Prob1a source code and an annotated version identifying appropriate fixes for all the known bugs, we extracted interchangeable "buggy" and "fixed" segments of the code. These source code partitions were respectively called "**bugn**" and "**fixn**" for all bugs n, $1 \leq n \leq 10$. An "a" or "b" was also added to the names of some partitions, since in a few cases the bug fix appeared at two non-contiguous locations in the source code. These "bug" and "fix" segments could be concatenated in a specific order to instantiate a legitimate version of a partially debugged source code version of Prob1a. (Note: For the two bugs from the original experiment that we discounted, the corresponding "fix" partitions were "force installed" in the program variants and were given "**.orig**" as an extension to their partition name.)

A shell script called **builder** was developed to automate the creation of source code for the LIC solution Prob1a with various combinations of fixes installed. Command line arguments to **builder** enabled us to provide a unique name for the program variant being created using the naming conventions discussed above, followed by an enumeration of the fix numbers that were to be installed — that is, integers in the range of 1 to 10 in ascending order for Prob1a. For any integer listed on the command line, **builder** was

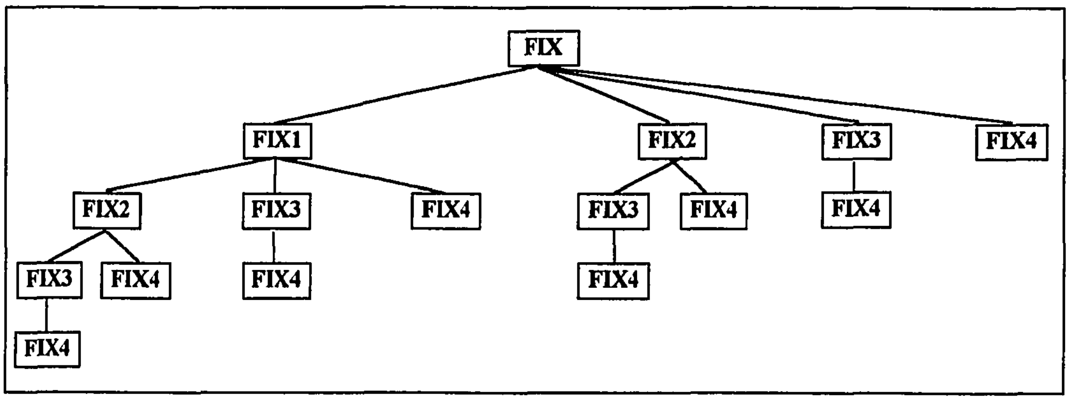

Figure C-1. Directory Structure for n=4 Known Bugs

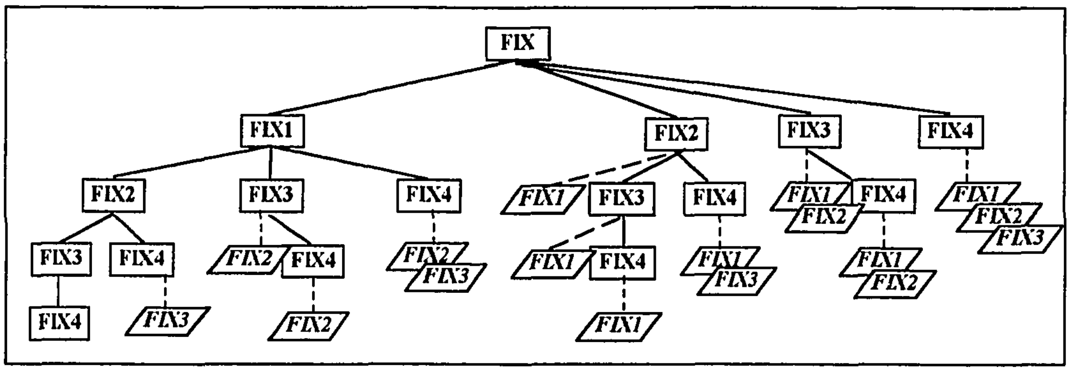

Figure C-2. Augmented Directory Structure for n=4 Known Bugs designed to install the corresponding "fixed" lines of code; otherwise the corresponding "buggy" lines of code would be used during the concatenation process. Thus, the command

```
builder fix0.for
```

would result in the creation of the a module called **fix0.for**, with no (revised) fixes installed, thereby equivalent to the level 0 program variant, while

```
builder fix2_7.for 2 7
```

resulted in the creation of a module called **fix2_7.for** with (revised) bugs 2 and 7 repaired, but bugs 1, 3 through 6 and 8 through 10 still present in the code. Each module could then be stored in its appropriate FIX subdirectory for subsequent data collection purposes.

C.4.2 Executable Generation

A shell script called **buildLoad** was also used. It accepts as its command line argument a FORTRAN module name which it inserts into a template of a second "canned" script called **loadLC** (for "*load L*ICCtrl"). This second script contains the **f77** command line arguments to compile the test version subroutine and link it together with the appropriate modules of the control software and the gold version to create an executable **LICCtrl** program. Thus it is possible to run **loadLC**, specifying the name of each partially debugged program variant module to be tested, and store the generated **loadLC** script in the directory in which the corresponding source code for the partially debugged program variant is stored. The **loadLC** scripts can then be run in those directories to create each executable **LICCtrl** program.

C.4.3 Level-Wise Graph Construction

To relieve the manual effort involved in creating the executable programs, a shell script called **buildLvl** was encoded. Given a level number and a few other command line arguments for navigation purposes, this script automatically creates the appropriate FIX subdirectories, the partially debugged program variants, the **loadLC** script and the executable **LICCtrl** program for each node of the level. It installs in each subdirectory virtually everything that is needed to collect data for that program variant. The **buildLvl** script was eventually run to create every level of the debugging graph in the FIX directory structure.

C.5 Data Collection

Due to the possibility of network failure and other unforseen problems, we still felt it was necessary to exert a certain amount of manual control over the actual data collection effort. Operating from left to right across a level of the directory structure, we used the operating system command "**at**" to schedule a number of data collection jobs to take place back-to-back on each of four or five machines on the Sun network. We tried to keep jobs running 24 hours a day, but some down time was necessary to accommodate system backups and partial data analysis

Although we had measured the **LICCtrl** execution at approximately four hours of wall-clock time, we allowed each job five hours to complete, thereby accounting for competition with other processes on the system. We ran the executables with a "nice level" of +19 to avoid hindering other system users. As jobs completed, we ran a shell script called **cleanLIC** to delete unnecessary files in its subdirectory and to compress the generated output to conserve disk space. If any jobs were found to have terminated prematurely, they were restarted at a later time, and the subsequent outputs concatenated to the earlier results.

As data collection was completed for an entire level, shell scripts called **cleanLvl** and **trimLvl** were run to ensure that all unnecessary files in each subdirectory had been deleted and all output compressed. This process was repeated until all subdirectories for each of the ten graph levels were processed. The overall collection effort required approximately two months.

Appendix D

Validation of Software Reliability Model Implementations

We performed a "sanity check" on the C programming language implementations of the four software reliability models included in our study: Jelinski-Moranda, Geometric De-Eutrophication, Basic Musa and Logarithmic Poisson. This was accomplished by using "canned" data, whose expected outputs were known or could be manually calculated, to input to the models. Each algorithm's behavior, given sanity data as inputs, was compared against expected outputs to validate its correct performance. We discuss the basic parameter estimation and validation procedures for each of the four models below, and provide their C programming language implementations.

D.1 Jelinski-Moranda

D.1.1 Formulae

The model estimates the total number of errors in a program by determining a value of N for which the following two functions are equal:

$$\Sigma_{i=1,n}[1 / (N - (i-1))] \quad \text{and} \quad n / (N - (\Sigma_{i=1,n}[(i-1) \cdot X_i]) / T)$$

In these equations:

N represents the total number of errors in the program;
n represents the number of failures observed;
X_i represents the time at which the i^{th} failure was observed;
T is the sum of all X_i's.

Table D-1. Sanity Data for Jelinski-Moranda Model
rounded to nearest 1/1000000

i	*R*	*F*	*MTTF*
0	0.000000	1.000000	1.
1	0.200000	0.800000	1.25
2	0.400000	0.600000	1.666667
3	0.600000	0.400000	2.5
4	0.800000	0.200000	5.
5	1.00000	0.00000	Infinite

A proportionality constant, ϕ, is then estimated by using the estimator for N in the following formula:

$$\phi = n / ((N \cdot T) - \Sigma_{i = 1,n}[(i-1) \cdot X_i])$$

Thus, after the i^{th} error has been found, the residual number of errors in the software is estimated to be (N - n), while the failure rate F is (N - i) $\cdot$ ϕ. R and MTTF are estimated using the relationships R = 1 - F and MTTF = 1 / F.

D.1.2 Sanity Check

We generated input values for the model using $\phi = 0.2$ and N = 5. Applying these values iteratively as i ranged from 0 to 5 in the failure figure formula, and using known relationships of F = (N - i) $\cdot$ ϕ to R and MTTF, we obtained the data shown in Table D-1.

Using these data as inputs, our Jelinski-Moranda program correctly estimated the total number of errors and ϕ value for the predictive stages 1 through 4; for example, at stage 1, the 0^{th} and 1^{st} sets of values were used to interpolate the parameter estimates. It also correctly predicted the next MTTF in all 4 cases, including an extremely large (essentially infinite) MTTF at stage 4, which is consistent with the expected outputs.

At the 5^{th} predictive stage, the model correctly detected it could produce no solution. At this point, Littlewood's test was applied, which states that finite N and non-zero ϕ exist if and only if the following is true:

$$\Sigma_{i = 1,n}[(i-1) \cdot X_i] > (\Sigma_{i = 1,n}[X_i]) / n$$

where n is the number of failures observed thus far. The test failed, indicating that either an infinite number of bugs remained—which we knew to be false—or ϕ was equal to zero—which is reasonable, since all 5 canned failures had been accounted for. Thus we believe our Jelinski-Moranda implementation is a valid one.

D.2 Geometric De-Eutrophication

D.2.1 Formulae

D represents the initial detection rate. It holds until the first error is found, at which time the rate becomes $k \cdot D$, where $0 < k < 1$. In general, the detection rate is $k^i \cdot D$ after the i^{th} error has been found, with the detection rates forming a converging geometric series.

The model estimates a value for the proportionality constant k for which the following two functions are equal:

$$(n + 1) / 2 \quad \text{and} \quad (\Sigma_{i = 1,n}[\, i \cdot k^i \cdot X_i \,]) / (\, \Sigma_{i = 1,n}[\, k^i \cdot X_i \,]\,)$$

In these equations:

n represents the number of failures observed;
X_i represents the time at which the i^{th} failure was observed.

D can then be estimated by using k's estimate in the following formula:

$$D = n / (\, \Sigma_{i = 1,n}[\, k^{i-1} \cdot X_i \,]\,)$$

F is easily estimated using the formula $D \cdot k^n$, while R and MTTF can be estimated using the relationships $R = 1 - F$ and $MTTF = 1 / F$.

D.2.2 Sanity Check

We generated input values for the model using $k = 0.2$ and $D = 1$. Applying these values iteratively as i ranged from 0 to 5 in the failure figure formula, and using known relationships of $F = D \cdot k^i$ to R and MTTF, we obtained the data shown in Table D-2.

Table D-2. Sanity Data for Geometric De-Eutrophication Model
rounded to nearest 1/1000000

i	*R*	*F*	*MTTF*
0	0.000000	1.000000	1.
1	0.900000	0.100000	10.
2	0.990000	0.010000	100.
3	0.999000	0.001000	1000.
4	0.999900	0.000100	10000.
5	0.999990	0.000010	100000.

Using the calculated sequence of reliability figures as inputs, our Geometric De-Eutrophication program correctly estimated the initial detection rate and proportionality constant for all 5 predictive stages; for example, at stage 1, the 0th and 1st sets of values were used to interpolate the parameter estimates. It also correctly predicted the next MTTF in all observed cases, although at the 5th stage some representational error was beginning to manifest itself in the 1/1000000 decimal place. Thus we believe our Geometric De-Eutrophication implementation is a valid one.

D.3 Basic Musa

D.3.1 Formulae

This model is the continuous counterpart to Jelinski-Moranda. A value b_1, which represents the ratio of initial failure intensity over the total number of bugs in the program, is estimated for which the following holds:

$$m_e / b_1 - m_e \cdot t_e / (\exp(b_1 \cdot t_e) - 1) - \Sigma_{i=1,me}[t_i] = 0$$

In this equation:

m_e represents the number of bugs removed;
t_i represents the time the ith bug was removed;
t_e represents the time at which testing ended.

An estimator for b_0, the total number of bugs in the software, is obtained by using b_1's estimator in the following formula:

$$b_0 = m_e / (1 - \exp(-b_1 \cdot t_e))$$

The number of bugs removed by time t is then given by the function:

$$u(t) = b_0 \cdot (1 - \exp(-b_1 \cdot t))$$

The failure rate is estimated using the function:

$$\lambda(t) = b_0 \cdot b_1 \cdot \exp(-b_1 \cdot t)$$

For a given failure rate F, R and MTTF can be estimated using the relationships $R = 1 - F$ and $MTTF = 1 / F$.

D.3.2 Sanity Check

We generated input data for the model using failure time data provided in [22, Table 12.1, page 305]. Using the known relationships among R, F and MTTF, where in this case MTTFs are approximated using observed failure times, we obtained the empirical data shown in Table D-3. According to the authors' example [22, page 324], the model's parameters should be approximated as b_0 = 142 failures (rounded to the nearest integer) and b_1 = 0.0000348/CPU sec at the final predictive stage when Musa's data are provided as inputs. We considered how well our implementation matched these known b_0 and b_1 values to validate our model. We ignored for the moment how predicted MTTFs compared with empirical ones.

Our program matched the authors' parameters quite well at the 136th predictive stage, using the time of the last failure (88682) as the time testing ended. That is, when all input data were considered, the interpolated parameters were $b_0 \approx 142.881$ and $b_1 \approx$ 0000342. If instead we used the true end of test time (91208), as did the authors, our parameter estimates matched theirs identically, with $b_0 \approx 141.933$ and $b_1 \approx .0000348$. Thus we believe our Basic Musa implementation is a valid one.

D.4 Logarithmic Poisson

D.4.1 Formulae

This model is the continuous counterpart of Geometric De-Eutrophication. A value b_1, which represents the product of the initial failure intensity and an intensity decay parameter, is estimated for which the following holds:

$$1 / b_1 \cdot (\Sigma_{i=1,m_e}[1 / (1 + b_1 \cdot t_i)]) - m_e \cdot t_e / ((1 + b_1 \cdot t_e) \cdot \ln(1 + b_1 \cdot t_e)) = 0$$

In this equation:

m_e represents the number of bugs removed;
t_i represents the time the ith bug was removed;
t_e represents the time at which testing ended.

An estimator for b_0, the inverse of the intensity decay parameter, is obtained by using b_1's estimator in the following formula:

$$b_0 = m_e / \ln(1 + b_1 \cdot t_e)$$

The number of bugs removed by time t is then given by the function:

$$u(t) = b_0 \cdot \ln(1 + b_1 \cdot t)$$

The failure rate is estimated using the function:

$$\lambda(t) = b_0 \cdot b_1 / (1 + b_1 \cdot t)$$

For a given failure rate F, R and MTTF can be estimated using the relationships R = 1 - F and MTTF = 1 / F.

D.4.2 Sanity Check

We generated input data for the model using failure time data provided in [22] (see Table 12.1, page 305). Using the known relationships among R, F and MTTF, where in this case MTTFs are approximated using observed failure times, we obtained the empirical data shown in Table D-3. According to the authors' example (see page 326), the model's parameters should be estimated as $b_0 = 42.3$ failures and $b_1 = 0.000262$/CPU sec at the final predictive stage when Musa's data are provided as inputs. We considered how well our implementation matched these known b_0 and b_1 values to validate our model. We ignored for the moment how predicted MTTFs compared with empirical ones.

Our program matched the authors' parameters quite well at the 136th predictive stage, using the time of the last failure (88682) as the time testing ended. That is, when all input data were considered, the interpolated parameters were $b_0 \approx 43.1$ and $b_1 \approx 000253$. If instead we used the true end of test time (91208), as did the authors, our parameter estimates matched theirs identically, with $b_0 \approx 42.3$ and $b_1 \approx .000262$. Thus we believe our Basic Musa implementation is a valid one.

Table D-3. Sanity Data Derived from Musa's Failure Time Data
rounded to nearest 1/1000000

i	*R*	*F*	*MTTF*
1	.666667	.333333	3
2	.969697	.030303	33
3	.993151	.006849	146
4	.995595	.004405	227
5	.997076	.002924	342
6	.997151	.002849	351
7	.997167	.002833	353
8	.997748	.002252	444
9	.998201	.001799	556
10	.998249	.001751	571
11	.998590	.001410	709
12	.998682	.001318	759
13	.998804	.001196	836
14	.998837	.001163	860
15	.998967	.001033	968
16	.999053	.000947	1056
17	.999421	.000579	1726
18	.999458	.000542	1846
19	.999466	.000534	1872
20	.999497	.000503	1986
21	.999567	.000433	2311
22	.999577	.000423	2366
23	.999617	.000383	2608
24	.999626	.000374	2676
25	.999677	.000323	3098
26	.999695	.000305	3278
27	.999696	.000304	3288
28	.999774	.000226	4434
29	.999801	.000199	5034
30	.999802	.000198	5049
31	.999803	.000197	5085
32	.999803	.000197	5089
33	.999803	.000197	5089
34	.999804	.000196	5097
69	.999937	.000063	15806
70	.999938	.000062	16185
71	.999938	.000062	16229
72	.999939	.000061	16358
73	.999942	.000058	17168
74	.999943	.000057	17458
75	.999944	.000056	17758
76	.999945	.000055	18287
77	.999946	.000054	18568
78	.999947	.000053	18728
79	.999949	.000051	19556
80	.999951	.000049	20567
81	.999952	.000048	21012
82	.999953	.000047	21308
83	.999957	.000043	23063
84	.999959	.000041	24127
85	.999961	.000039	25910
86	.999963	.000037	26770
87	.999964	.000036	27753
88	.999965	.000035	28460
89	.999965	.000035	28493
90	.999966	.000034	29361
91	.999967	.000033	30085
92	.999969	.000031	32408
93	.999972	.000028	35338
94	.999973	.000027	36799
95	.999973	.000027	37642
96	.999973	.000027	37654
97	.999973	.000027	37915
98	.999975	.000025	39715
99	.999975	.000025	40580
100	.999976	.000024	42015
101	.999976	.000024	42045
102	.999976	.000024	42188

Table D-3 (concluded). Sanity Data Derived from Musa's Failure Time Data
rounded to nearest 1/1000000

35	.999812	.000188	5324
36	.999814	.000186	5389
37	.999820	.000180	5565
38	.999822	.000178	5623
39	.999836	.000164	6080
40	.999843	.000157	6380
41	.999846	.000154	6477
42	.999852	.000148	6740
43	.999861	.000139	7192
44	.999866	.000134	7447
45	.999869	.000131	7644
46	.999872	.000128	7837
47	.999872	.000128	7843
48	.999874	.000126	7922
49	.999886	.000114	8738
50	.999901	.000099	10089
51	.999902	.000098	10237
52	.999903	.000097	10258
53	.999905	.000095	10491
54	.999906	.000094	10625
55	.999909	.000091	10982
56	.999911	.000089	11175
57	.999912	.000088	11411
58	.999912	.000088	11442
59	.999915	.000085	11811
60	.999920	.000080	12559
61	.999920	.000080	12559
62	.999922	.000078	12791
63	.999924	.000076	13121
64	.999926	.000074	13486
65	.999932	.000068	14708
66	.999934	.000066	15251
67	.999934	.000066	15261
68	.999935	.000065	15277

103	.999976	.000024	42296
104	.999976	.000024	42296
105	.999978	.000022	45406
106	.999979	.000021	46653
107	.999979	.000021	47596
108	.999979	.000021	48296
109	.999980	.000020	49171
110	.999980	.000020	49416
111	.999980	.000020	50145
112	.999981	.000019	52042
113	.999981	.000019	52489
114	.999981	.000019	52875
115	.999981	.000019	53321
116	.999981	.000019	53443
117	.999982	.000018	54433
118	.999982	.000018	55381
119	.999982	.000018	56463
120	.999982	.000018	56485
121	.999982	.000018	56560
122	.999982	.000018	57042
123	.999984	.000016	62551
124	.999984	.000016	62651
125	.999984	.000016	62661
126	.999984	.000016	63732
127	.999984	.000016	64103
128	.999984	.000016	64893
129	.999986	.000014	71043
130	.999987	.000013	74364
131	.999987	.000013	75409
132	.999987	.000013	76057
133	.999988	.000012	81542
134	.999988	.000012	82702
135	.999988	.000012	84566
136	.999989	.000011	88682

Glossary

Acronyms

F	Failure Rate
MTTF	Mean Time to Failure
NASA	National Aeronautics and Space Administration
ODU	Old Dominion University
R	Reliability

Terms

ambient data: Information collected and used in its natural state, with no pre-processing.

bathtub curve: The functional plot of failure rate versus time that describes the expected life of hardware components.

bias: Consistent deviation between prediction and reality.

"big O" notation: The asymptotic order of magnitude of the time complexity of an algorithm as size increases; for example, an algorithm that processes inputs of size n in time cn^2 for some constant c is said to have time complexity $O(n^2)$, read "order n^2."

bug: A colloquial term for fault.

compensation: A kind of fault interaction originally reported by Dunham in which certain failures occur when either of two faults is in a program, but not when boths faults are simultaneously present.

conditions met matrix: Part of the LIC output; a column matrix containing a zero for any LIC condition which has not been met, and a one for any condition which has been met.

consistent comparison problem: An observed effect of finite-precision arithmetic in which two sets of different computations, which should produce the same output, arrive at very different values due to slight deviations in intermediate calculations attributable to the order of comparisons and the particular arithmetic algorithms used by the hardware.

control program: The software implementation of an algorithm for performing an empirical reliability estimation for a test program against a gold version.

counter-intuititve path: A debugging session which recovers faults in a smallest-to-largest size order.

data aging: The selection of a subset of the failure data based on the assumption that older data may not be as representative of the current and future failure process as more recent data.

debugging graph: A pictoral representation of the *n*! orders in which a collection of *n* faults can be removed from a program.

debugging session: The recovery of acollection of *n* known faults from a program; a path in the debugging graph from P to $P_{1...n}$ that follows edges through exactly one node at each of the levels 0 through *n* and represents the removal of all known bugs from the program.

debugging state: The combination of faults and repairs known to be present in a program at a particular point in time during the debugging process.

delta: A relative change in reliability produced by installing a repair in the program.

delta graph: A debugging graph constructed using a partially debugged variant instead of an oracle, and whose edges are labeled with delta values.

dynamic relative size ranking: Constructing a non-decreasing or non-increasing sequence of failure rates associated with program variants at progressively higher level in the debugging graph for the purpose of arranging the program's faults in a smallest-to-largest or largest-to-smallest order.

error graph: Former terminology for debugging graph.

f77: The name of the standard FORTRAN compiler on the ODU Sun network.

fail set: The collection of input cases for which failures are observed and attributed to some fault.

failure: A departure of the external results of a program's execution from its requirements on a particular run.

failure rate: An expression for the probability that a software product will exhibit a failure during a given time period in its specified environment.

fault: Defective, missing or extra code that is the cause of one or more failures for a program.

fault recovery: The identification of faults and the implementation of suitable code repairs to remove them from the program.

full compensation: A kind of fault interaction originally reported by Dunham in which neither of two faults' associated failures are manifested when the faults are simultaneously present in a program.

gold oracle: An error detector constructed using a gold version of the software being tested.

gold version: An independently developed version of the software being tested, whose reliability is believed to be perfect.

intuititve path: A debugging session which recovers faults in a largest-to-smallest size order.

Launch Interceptor Condition: A simulation of part of a radar tracking system that generates a launch interceptor signal based on input tracking coordinates; the subject software of the experiments documented in this thesis.

level: One row in the debugging graph consisting of variants having same-sized subsets of known repairs (e.g., P_1 through P_n).

LICCtrl: The name of the control program used in the experiments reported in this thesis.

mean time to failure: The expected value of time between failures.

negative interaction: The insertion of a correct repair which makes no difference, or degrades, the program's performance during certain debugging states.

noise: Large variability in the difference between prediction and reality.

oracle: An error detector constructed by running the tested software in tandem with a highly reliability, independently developed solution to the same problem whose outputs are used as the performance baseline.

partial compensation: A kind of fault interaction originally reported by Dunham in which only some of two faults' associated failures still can be observed when the faults are simultaneously present in a program.

path truncation: A kind of data aging applied to debugging graph data in which only the last *w* MTTF values representing a given debugging session are used in the predictive process.

pipe: An i/o mechanism for creating a private, interprocess communication channel for data sharing.

positive interaction: The removal of a correct repair which has no effect on, or degrades, the program's performance during certain debugging states.

recalibration: A graphical technique for adjusting a model's predictions based on past performance.

reification: In software design, the conversion of an abstract conceptualization of functionalities into the mechanisms necessary to implement them given a set of constraining requirements, such as host machine architecture and programming language features.

repair: Code whose installation removes a fault from the program.

repetitive run modeling: An approach to replicated debugging devised by Nagel and Skrivan to study error rates.

run: A single execution instance of a program involving the transformation of an input case to an output (or abnormal termination).

software reliability: The probability of a software product operating for a given period of time in a particular environment without exhibiting any failures.

stage: An iteration of the debugging process; at stage *i*, the first *i*-1 failures have already been observed and corrected.

static relative size ranking: Constructing a non-decreasing or non-increasing sequence of failure rates associated with program variants at a fixed level in the debugging graph for the purpose of arranging the program's faults in a smallest-to-largest or largest-to-smallest order.

surrogate oracle: An error detector constructed using the software being tested with all known repairs installed.

switch: An input argument, signaled by a dash ('-') and commonly supplied after the program name on the command line invocation, in the UNIX environment.

time complexity: The computation time needed by an algorithm expressed as a function of problem size.

u-plot: A graph of previously predicted failure rates arranged so that they appear to be a random sample from the uniform probability distribution which is used to recalibrate prediction systems.

UNIX: A highly portable, rich and productive programming environment written in the C programming language at Bell Laboratories in the later 1960's that has grown to world-wide use; UNIX is not an acronym, but a weak pun on the name of the operating system (MULTICS) on which its originators worked prior to UNIX.

VMS: The name of the VAX operating system.

variant: Any version of the original program with some subset of the known repairs installed.

wrapper: Additional code added to a module — usually both before and after its original text — to ease its integration into a larger system.

AUTOBIOGRAPHICAL STATEMENT

Mary Ann Hoppa was born in Pittsburgh, Pennsylvania on October 24, 1959. She graduated Magna Cum Laude, with both a B.S. in Applied Mathematics and a B.A. in French, from Auburn University, Alabama in June 1981. In January 1986, Ms. Hoppa completed an M.S. in Computer Science at George Mason University, Virginia, where she concentrated on complexity analysis and public key encryption algorithms. She began her doctoral studies at Old Dominion University, Virginia in January 1988 and was later inducted into Phi Kappa Phi.

Ms. Hoppa has 15 years experience as a professional software engineer for various contractors to the U.S. government, specializing in defense applications. She is currently a task engineer with The MITRE Corporation, where she supports the development of object-oriented solutions to tactical message handling problems for the U.S. Air Force. Her publications include the following: "Some Effects of Fault Recovery Order on Software Reliability Models," co-authored with L. W. Wilson, which appeared in *Proceedings of the Fifth International Symposium on Software Reliability Engineering (ISSRE94)*; "An Empirical Assessment of Interface Changes for an Object-Oriented, 'Not-So-Rapid' Prototype," which appeared in *Proceedings of the 11th International Technology of Object-Oriented Languages & Systems (TOOLS11) Conference*; and "The Application of Object-Oriented Techniques to Processing United States Message Text Formats," which appeared in *Proceedings of the 3rd International Technology of Object-Oriented Languages & Systems (TOOLS3) Conference*.