Hardware and Software Reliability (323-08)

Application and Improvement of Software Reliability Models

Submitted By: Dolores Wallace
And
Charles Coleman, Manager, SATC

October 12, 2001

Technical POC: Dr. Linda Rosenberg

Phone #: 301-286-0087
Fax #: 301-286-1701
Email: Linda.Rosenberg@gsfc.nasa.gov
Mail Code: 304

Administrative POC: Dennis Brennan

Phone #: 301-286-6582
Fax #: 301-286-1667
Email: Dennis.Brennan@gsfc.nasa.gov
Mail Code: 300

ABSTRACT

This report presents the results of Task 323-08, Hardware and Software Reliability. Although hardware and software differ, they share a sufficient number of similarities that the mathematics used in hardware reliability modeling have been applied to software reliability modeling. This task examines those models and describes how they may be practical for application to projects at Goddard Space Flight Center. The task also resulted in improvements to one model to allow for fault correction.

EXECUTIVE SUMMARY

NASA acquires and uses many systems in which software is a major component, and many of these systems are critical to the success of NASA's mission. These systems must execute successfully for a specified time under specified conditions, that is, they must be reliable. The capability to provide accurate measurement of the reliability of the software in these systems before NASA accepts them is an essential part of ensuring that NASA software will meet its mission requirements.

The purposes of Task 323-08, Hardware and Software Reliability, are to examine reliability engineering in general and its impact on software reliability measurement, to develop improvements to existing software reliability modeling, and to identify the potential usefulness of this technique as one data point in measuring reliability of software at the Goddard Space Flight Center.

The first part of this project identified the mathematics and statistical distributions used in reliability modeling and found that essentially all have been applied to software reliability modeling. The study identified major differences between hardware and software and indicated that the software reliability models do not specifically accommodate those differences. The study resulted in several recommendations for model modification.

The second part of this project explored the use of these software reliability models at Goddard Space Flight Center (GSFC) and their improvement. A case study to determine usefulness of this technique at GSFC used project failure data and characterized and manipulated it for use with a software reliability tool. The actual process of software reliability modeling includes the preparation of the data, selection of the appropriate model, and analysis and interpretation of results of the models. A key criterion to practicality is the amount of effort required for each step.

The Naval Space and Warfare Center (NSWC), Dahlgren, Virginia, has sponsored the development of a software tool, Software Modeling and Estimation of Reliability Functions for Software (SMERFS), under the direction of Dr. William B. Farr. This public domain tool exercises several software reliability models and served as an instrument for assessing usability of software reliability modeling at GSFC

One difference between hardware and software is the correction process. By the time hardware is in operation and reliability studies occur, generally design faults have been removed. With software, faults are often remaining during system test and operation. The hardware reliability models do not account for correction during the time of reliability measurement. For this research task, Dr. Norman Schneidewind of the Naval Postgraduate School developed adjustments to the Schneidewind model to allow for fault correction.

This report describes the results of these studies.

TABLE OF CONTENTS

1. Introduction

NASA is increasingly dependent upon systems in which software is a major component. These systems are critical to the success of NASA's mission and must execute successfully for a specified time under specified conditions, that is, they must be reliable. The capability to accurately measure the reliability of the software in these systems is an essential part of ensuring that NASA systems will meet mission requirements.

The Software Assurance Technology Center (SATC) at the NASA Goddard Space Flight Center (GSFC) performed Task 323-08, Hardware and Software Reliability to examine reliability engineering, its impact on software reliability measurement and the practicality of using it to provide one data point for measuring the reliability of software at GSFC. Reliability engineering executes various mathematical functions on past failure data to predict future behavior of a component or system, that is, to measure the increase in its reliability, usually referred to as reliability growth. This project explored the improvement of software reliability engineering models to accommodate fault correction.

The first part of this project identified the mathematics and statistical distributions used in reliability modeling and found that essentially all have been applied to software reliability modeling. The study identified major differences between hardware and software and indicated that the software reliability models do not specifically accommodate those differences. While the complete findings were reported previously, Section 2 of this report contains a brief summary to provide appropriate context for this part of the project.

The second part of this project explored issues for using these software reliability models at GSFC and developed improvements to the Schneidewind model. We examined the process of software reliability modeling identified by the American Institute of Aeronautics and Astronautics in its *Recommended Practice for Software Reliability* [AIAA]. The purpose was to identify the difficulties of using software reliability modeling and some steps of the process that may be made easier with the aid of a software reliability modeling tool.

The mathematical and statistical functions used in software reliability engineering employ several steps. The equations for the models themselves have parameters that are estimated from techniques like least squares or maximum likelihood estimation. Then the models, usually equations in some exponential form, must be executed. But verifying the selected model for the particular data set may require iteration and study of the model functions. From these results predictions can be made, and confidence intervals for the predictions can be computed. All of these computations are time-consuming and error-prone when computed manually.

We searched for a software tool to assist us with software reliability modeling on GSFC project data to understand how practical use of this measurement technique can be. One tool that reduces the difficulty of software reliability modeling is the Software Modeling and Estimation of Reliability Functions for Software (SMERFS), developed under the direction of Dr. William B. Farr of the Naval Surface Warfare Center, Dahlgren, Virginia. It performs curve-fitting, model selection and execution, and statistical analysis for several software reliability models.

The latest version, SMERFS^3, like its predecessors, contains the mathematics for many of the software reliability models. Except for user features, features of SMERFS concerning the models are likely to be similar to any other software reliability modeling tools. Because both the tool and guidance from Dr. Farr were available to us, we selected this tool to served as an instrument for assessing the practicality of using software reliability modeling at GSFC.

Next we found two GSFC projects with data in a defect tacking system. While failure data are available in this tracking system, other information may be needed to characterize the project, select models and organize data correctly. We describe this experience with SMERFS^3 and these data to show how software reliability modeling works. We identify intellectual aspects that the tool cannot perform. We also demonstrate the type of effort needed by the project staff to use software reliability modeling as a successful technique for software reliability measurement. We show pitfalls that may entrap those who do not analyze their project characteristics and data before exercising SMERFS^3 on failure data.

One difference between hardware and software is the correction process. By the time hardware is in operation and reliability data are collected, generally design faults have been removed. The hardware reliability models do not account for correction during the time of reliability measurement. With software, faults exist during system test and operation such that reliability growth occurs as these are corrected Dr. Norman Schneidewind of the Naval Postgraduate School developed adjustments to his model to allow for fault correction.

Section 2 of this report provides a brief synopsis from the first report of this study of the basic information about software reliability modeling. Section 3 describes in detail the software reliability modeling process and options for applying the process at GSFC. Section 4 describes the research results of modifying the Schneidewind model to accommodate differences between hardware and software, with Appendix B providing complete information. Section 5 provides the conclusions about potential use of software reliability modeling at GSFC. Appendix A describes a case study using GSFC project data with SMERFS^3.

2. Overview of Hardware and Software Reliability

Hardware and software reliability engineering have many concepts with unique terminology and many mathematical and statistical expressions. Basically, the approach is to apply mathematics and statistics to model past failure data to predict future behavior of a component or system. Major statistical distributions used in hardware reliability modeling include the exponential, gamma, Weibull, binomial, Poisson, normal, lognormal, Bayes, and Markov distributions. To use these distributions, data collected from failures of systems need to be fitted with techniques like maximum likelihood or least squares estimates. The appropriateness of the models selected need to be verified by using statistical methods like Chi-squared or goodness-of-fit. Because mechanical and electrical systems tend to deteriorate over time, these reliability distributions depend on time as the variable, usually calendar time.

To provide context for the rest of this report, this section provides definitions, descriptions of a few software reliability models, and assumptions and requirements for using these models. It also provides a discussion about differences between hardware and software and their impact on modeling for software reliability.

2.1 Definitions

Unless otherwise indicated, these definitions are taken from the Department of Defense MIL-HDBK-338B on electronic reliability [Mil338].

Failure: The event, or inoperable state, in which any item or part of any item does not, or would not, perform as previously specified.

Failure: (1) The inability of a system or system component to perform a required function within specific limits. A failure may be produced when a fault is encountered and a loss of the expected service to the user results. (2) The termination of the ability of functional unit to perform its required function. (3) A departure of program operation from program requirements [AIAA] .

Failure intensity function: the instantaneous rate of change of the expected number of failures with respect to time [Lyu].

Failure rate: The total number of failures within an item population, divided by the number of life units expended by that population, during a particular measurement period under stated conditions.

Failure rate: (1) The ratio of the number of failures of a given category or severity to a given period of time; for example failures per second of execution time, failure per month. Synonymous with failure intensity. (2) The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs [AIAA].

Mean Time Between Failure: A basic measure of reliability for repairable items. The mean number of life units during which all parts of the item perform within their specified limits, during a particular measurement under stated conditions.

Mean Time Between Failure: The expected or observed time between consecutive failures in a system or component [I982].

Mean Time to Failure: A basic measure of reliability for non-repairable items. The total number of life units of an item population divided by the number of failures within that population, during a particular measurement under stated conditions.

Reliability: 1. The duration or probability of failure-free performance under stated conditions. 2. The probability that an item can perform its intended function for a specified interval under stated conditions.

Reliability: The ability of a system or component to perform its required functions under stated conditions for a specified period of time [I610].

Reliability: see Software Reliability [I982].

Reliability growth: The improvement in reliability that results when design, material, or part deficiencies are revealed by testing and eliminated through corrective action.

Software reliability: (1) The probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered [AIAA] [I982]. (2) The ability of a program to perform a required function under stated conditions for a stated period of time [AIAA].

Software reliability model: A mathematical expression that specifies the general form of the software failure process as a function of factors such as fault introduction, fault removal and the operational environment [AIAA].

Time: a fundamental element used in developing the concept of reliability and is used in many of the measures of reliability. Determining the applicable interval of time for a specific measurement is a prerequisite to accurate measurement. Usually the interval of interest is calendar time, but may be broken down into other intervals (or calendar time may be reconstructed from other intervals).

Wearout: The process that results in an increase of the failure rate or probability of failure as the number of life units increases.

2.2 Software reliability models

Software reliability engineering produces a model of a software system based on its failure data to provide a measurement for software reliability. The mathematical and statistical functions used in software reliability modeling employ several computational steps. The equations for the models themselves have parameters that are estimated using techniques like least squares fit or maximum likelihood estimation. Then the models, usually equations in some exponential form, must be executed. Verifying that the selected model is valid for the particular data set may require iteration and study of the model functions. From these results predictions about the number of remaining faults or the time to next failure can be made, and confidence intervals for the predictions can be computed.

A few algorithms of some popular models are shown in Tables 1 and 2 [Pham]. Table 1 identifies software failure rate models used to study the program failure rate per failure at the failure intervals. Table 2 describes some NHPP models. Software reliability models rely on two types of data, either the number of failures per time period or the time between failures. Most software reliability models are well known and have been used in the 1980s and 1990s. Exponential distributions are used almost exclusively for reliability in the prediction of electronic equipment, that is, the probability distribution function pdf of X: $f(x) = \lambda e^{-\lambda t}$. This distribution

can be chosen as a failure distribution if and only if the assumption of a constant hazard rate can be justified, that is, the hazard rate = λ [Mann]. Models using an exponential distribution include the Musa Basic, Jelinski-Moranda De-eutrophication, Non-homogeneous Poisson process (NHPP), Goel-Okumoto, Schneidewind, and hyperexponential models. For these, memory is not important, that is, failures in the past have no impact on future failure rates.

The Weibull model is a general form of the gamma distribution, lognormal, exponential, or normal, depending on the value of β. Variations include the S-shaped reliability growth model and the Rayleigh model. Musa uses the logarithmic model and assumes errors contribute differently to the error rate. Shooman uses the binomial distribution and Littlewood-Verrall uses Bayesian statistics. For Bayesian models, memory is important, that is, what has failed before has an impact on current and future failure rates [Lyu].

Table 1. Software Reliability Failure Rate Models

Model Name	**$f(t_i)$**	**$R(t_I) = 1 - F(t_i)$**	**$\lambda(t_i)$**
Jelinski-Moranda	$\phi[N-(i-1)]\exp(-\phi(N-(i-1))t_i)$	$\exp(-\phi(N-i+1)t_i)$	$\phi[N-(i-1)]$
Schick-Wolverton	$\phi[N-(i-1)]t_i\exp(-(\phi[N-(i-1)]t_i^2)/2)$	$\exp(-(\phi[N-(i-1)]t_i^2)/2)$	$\phi[N-(i-1)]t_i$
J-M Geometric	$Dk^{i-1}\exp(-Dk^{i-1}t_i)$	$\exp(-Dk^{i-1}t_i)$	$\exp(-Dk^{i-1})$
Goel-Okumoto	$\phi[N-p(i-1)]\exp(-\phi[N-p(i-1)]t_i)$	$\exp(-\phi[N-p(i-1)]t_i)$	$\phi[N-p(i-1)]$
Littlewood-Verrall	$\alpha[\xi(i)/(t_i+\xi(i)]^{\alpha}[1/(t_i+\xi(i))]$	$\int_{t_i}^{\infty}\{\alpha[\xi(i)/(s+\xi(i)]^{\alpha}$ $[1/(s+\xi(i))]\}ds$	$\alpha/(t_i+\xi(i))$
Shooman	NA	NA	$\omega - [\text{from } 0 \text{ to } t_i\int\beta_3(s)\,ds]\beta_1\beta_2$
Weibull	$(\beta(t-\gamma)^{\beta-1}/\theta^{\beta})\exp(-((t-\gamma)/\theta)^{\beta})$	$\exp(-((t-\gamma)/\theta)^{\beta})$	$(\beta(t-\gamma)^{\beta-1}/\theta^{\beta})$

Where
α = a parameter to be estimated;
ϕ = a proportional constant, the contribution any one fault makes to the overall program;
N = the number of initial faults in the program;
t_i = the time between the (i–1) th and the i th failures;
D = initial program failure rate;
k = parameter of geometric function (0<k<1);
p = the probability of removing a failure when it occurs;
$\xi(i) = \beta_0 + \beta_1 i$ or $\beta_0 + \beta_1 i^2$;
ω = the inherent fault density of the program;
β_1 = the proportion of unique instructions processed;
β_2 = a bulk constant;
$\beta_3(t)$ = the faults corrected per instruction per unit time.

Table 2. Software Reliability NHPP Models

Model name	Model type	MVF(m(t))
Goel-Okumoto (G-O)	Concave	$m(t) = a(1-\exp(-bt))$ $a(t) = a$ $b(t) = b$
Schneidewind	Concave	$m(t) = (a/b)(1-\exp(-bt))$ $a(t) = a$ $b(t) = b$
Duane Growth	Concave or S-shaped	$m(t) = at^b$ $a(t) = a$ $b(t) = b$
Delayed S-shaped (Yamada)	S-shaped	$m(t) = a(1-(1+bt)\exp(-bt))$ $a(t) = a$ $b(t) = b^2t/(1+bt)$
Inflection S-shaped (Ohba)	Concave	$m(t) = a(1-\exp(-bt))/(1+\beta\exp(-bt))$ $a(t) = a$ $b(t) = b/(1+\beta\exp(-bt))$
Musa Basic	Concave	$m(t) = \beta_0(1-\exp(-\beta_1 t))$
Musa Log	Concave	$m(t) = a(1-\exp(-ct/nT))$ $a = k/\Sigma_{i=1}^{k}(1-\exp(-ct_i/nT))$ $c = (1/knT)\Sigma_{i=1}^{k}t_i+(a/knT)\Sigma_{i=1}^{k}t_i \exp(-ct_i/nT)$

where
MVF = the mean value function;
m(t) = expected number of errors detected by time t ("mean value function");
a(t) = error content function, i.e., total number of errors in the software including the introduced errors at time t;
b(t) = error detection rate per error at time t;
t_i = the observed time between the (i–1) th and the i th failure;
a = number of failures in the program;
c = the testing compression factor;
T = mean time to failure at the beginning of the test;
n = total number of failures possible during the maintained life of the program.

2.3 Requirements for using the models

Certain assumptions should be true for the models to produce valid results; these are provided for several models in Table 3 [Lyu]. Musa and several other models share the following group of assumptions, referred to as basic assumptions in Table 3:

1. Software is operated in a similar manner as that in which reliability predictions are to be made.
2. Every fault has same chance of being encountered within a severity class as any other fault in that class.
3. The failures when faults are detected are independent.

Table 3. Assumptions for software reliability models

Model name	Assumptions
Musa Basic Exponential Applied after integration	1. Basic assumptions 2. Cumulative number failures by time t, M(t), follows Poisson process with mean value function $\mu(t) = \beta_0 [1-\exp^{-(\beta 1 t)}]$ where β_0, $\beta_1 > 0$. $\mu(t)$ such that the expected number of failure occurrences for any time period is proportional to the expected number undetected faults at that time. Finite failure model. 3. Time between failure expressed in cpu time 4. Errors contribute equally to error rate 5. Finite number of inherent errors 6. Error rate declines uniformly for every error corrected => constant error rate over time 7. Mean value function: expected number failure occurrences at anytime proportional to number undetected faults at that time 8. Execution times between failures are piecewise exponentially distributed (hazard rate for a single fault is constant) 9. Quantities of resources number fault ids, correction personnel, computer times) available are constant over segment for which the software is observed 10. Resource expenditures for the kth resource can be approximated (complicated) 11. Fault correction personnel utilization established by the limitation of fault queue length for any fault correction person. Fault queue is determined by assuming that fault correction is a Poisson process and that servers are randomly assigned in time. 12. Fault – identification personnel can be fully utilized and computer utilization is constant
Musa Log Applied during unit to system test	1. Basic assumptions 2. Infinite number of inherent errors 3. Logarithmic: expected number failures over time is a log function 4. NHPP. With intensity function that decreases exp. As failures occur; earlier discovered errors have greater impact on reducing failure intensity function 5. Because of 3, failure intensity decreases exponentially with expected number failures experienced 6. Cumulative number failures by time t, M(t), follows a Poisson process. 7. Every fault has same chance of being encountered within a severity class as any other fault in that class.[Neu] 8. The failures when faults detected are independent.[Neu] 9. Failure rate not constant; errors contribute differently to error rate
Schneidewind Exponential Applied During Integration, System test Or operation	1. Current fault rate better predictor of future behavior – changing failure rate 2. 3 forms of model: all n, from s to n; use the two groups differently 3. Basic assumptions 4. Cumulative number failures by time t, M(t), follows a Poisson process with mean value function $\mu(t)$ such that the expected number of fault occurrences for any time t to $t + \Delta t$ proportional to expected # of undetected faults at time t. Bounded nondecreasing function of time. Finite failure model 5. Failure intensity function $\lambda(t)$: exponentially decreasing function of time; $\lambda(t) = \alpha \exp^{-\beta t}$ Large β implies small failure rate; small β implies large. α is initial failure rate at time t = 0 6. Number of faults f_1, f_2,... f_n detected in each interval $[(t_0 , t_1), (t_1,t_2), \ldots (t_{(n-1)}), t_n]$ independent for any finite collection of times $t_1,t_2,\ldots<t_n$ 7. Fault correction rate is proportional to the number of faults to be corrected. 8. Intervals over which software observed are of same length so $t_i = l*i$, for i= 1,..n. When l (length of interval) =1 (the number), then $t_i = i$.

The models require project data for their execution. Several use time between failure or the actual times of failure. Others use the number of faults in each testing interval. Other types of data may be used. Table 4 summarizes the data requirements [Lyu].

Table 4. Data Requirements for Software Reliability Models

#	Data	Model
1.	Elapsed time between failures $x_1, x_2,... x_n$ or actual times of failure, $t_1, t_2,.. t_n$ where $x_i = t_i - t_{(i-1)}$, i =1 ,…n, t_0 =0	Musa basic for execution time component; Musa-Okum log.
2.	Fault counts in each testing interval, $f_1, f_2,... f_n$	Schneidewind
3.	The available resources for both identification and correction personnel and the number of computer shifts such as P_I, P_F, and P_C. The utilization factor for each resource, that is, ρ_I (=1), ρ_I, and ρ_C. The execution time coefficient of resource expenditure for each resource, that is, θ_I, θ_f, (usually 0), and θc. The failure coefficient of resource expenditure for each resource, that is μ_I, μ_F, and μ_C. The maximum fault queue length Q for a fault correction personnel. The probability P that the fault queue length is no larger than Q	Musa basic calendar time component

2.4 Some Hardware and Software Differences Impacting Reliability Models

Several differences between hardware and software may have an impact on the validity of hardware reliability models applied to software.

The source of failure in software is a design fault, while for hardware the cause has usually been physical deterioration, a manufacturing defect, or poor quality of materials. The hardware component is removed and another component with the same specifications replaces the defective part. Design errors have usually been removed before manufacturing. While sometimes a hardware component may need to be ordered, often replacements are kept on hand. There may be some down time if the part is not readily available but the part itself does not require a corrective process adding to the downtime.

For software, a correction is made directly in the defective component, or if in a copy of the software component, the corrected copy then replaces the defective component in all versions of the software. For software, the repair process is not a simple matter of replacing a part, and the corrective process of software may itself introduce new faults.

Software correction time adds incorrectly to the time between faults because they imply the same assumption in the hardware reliability models that the time is counted for a continuously tested or executing system. For hardware, calendar time is close to execution time. For software, the time between failures is in calendar time and is often not close to execution time. There are other problems concerning the corrective process of software. Sometimes, when strict configuration control is not in place, the wrong version of the software component may be changed. This replacement may contain previous problems or may lack features of the current version. The corrected software may not have been adequately regression-tested and may contain

new faults. If not properly tested for integration with the system, it may contain additional faults. Reliability models have not generally considered these factors.

Hardware reliability may change during certain periods such as at initial use or at end of a useful life. Software reliability will vary during the periods of development and test because faults are introduced, discovered and then removed. The faults that lead to the failures have many causes such as incorrect logic, incorrect statements, misunderstood or incorrect requirements, incorrect input data descriptions or other mistakes in the development of the software. It is assumed that the failure rate for software will decrease during testing and be essentially stable during operation (after an initial shake-out period). Software faults correspond to hardware design errors. In many instances both hardware and software design errors are systematic because they cause the system to fail every time certain inputs or environmental conditions are encountered. As the traditional causes of hardware failure become less important and as the complexity of integrated chips grows, hardware design errors are becoming a significant factor in hardware failure. Several reliability experts now claim that the fields of hardware reliability and software reliability are converging [Fried]. It is possible that accommodations in software reliability models for unique software features may ultimately benefit hardware if indeed hardware design errors grow.

3. Applying Software Reliability Modeling at GSFC

Although hardware and software are different, software reliability models based on hardware reliability models have been developed and used in industry. Some issues that may affect their usage at GSFC include:

- The modeling process
- Data collection requirements
- Availability of software tools to support these models
- Difficulties of using the models and interpreting their results and
- Options for using software reliability modeling at GSFC.

3.1 The Modeling Process

The current software reliability modeling process is based on features of hardware. While some models compensate for some differences, overall they do not. Schneidewind, for example, discounts the earlier part of the software failure history because the software gets corrected and therefore the earlier number of failures does not affect the current failure rate. Some models exercise the system to identify the mean time to next failure and assume that the system has been tested after debugging with no failures observed. The models assume that the system is exercised in circumstances very similar to the final operating environment. Usually software reliability growth models are used during debugging to predict when the mean time to failure is large enough to release the software [Hamlet].

The AIAA's Recommended Practice for Software Reliability defines a formal procedure of eleven steps for software reliability estimation [AIAA]. The first three pertain to establishing

requirements for the system and are outside this discussion. Another, defining failure, is accomplished by NASA guidelines on defining non-conformances [GPG] and by the project's requirements on the data they will collect for their defect tracking system. The software reliability analyst may help to establish those requirements. The remaining steps are essential to software reliability modeling and some may require interaction between project members and the reliability analyst. They include:

- Characterizing the operational environment
- Selecting tests
- Selecting the models
- Collecting data
- Estimating parameters
- Validating the model, and
- Performing analysis.

Characterizing the operational environment. The purpose of reliability measurement is to assign some reliability measure to a system, such as predicting the number of faults remaining, the number of failures expected in a given time, how much time is needed to find a specified number of faults, or the probability of operating without failure in a specified time. Systems are made of components and reliability measures may be determined for them rather than the entire system or for the entire system rather than the components. Hence, the analyst needs to know the system configuration either to allocate system reliability to component reliabilities or to combine component reliabilities to establish system reliability. The system may evolve with new code or components and these may affect usage of the reliability estimations. Finally, the system operational profile may show how different modes are utilized and may need to have separate reliability tracking for them.

Test approach. The analyst needs to know the testing approach because it may influence how the failure data are used; this issue is discussed with data collection.

Model selection. Model selection is complex. Fortunately, software tools ease this complexity but it is worthwhile to list here the criteria for model selection:

- predictive validity
- ease of parameter measurement (e.g., the amount of data should be ~5 times as much as the number of parameters)
- quality of assumptions
- capability
- applicability
- simplicity
- insensitivity to noise (calendar instead of execution time) [AIAA].

Predictive validity addresses the forecast quality of each model and is usually determined by a goodness-of-fit test. Most of the models have parameters that need to be determined before the equations of the model can be solved. Fortunately, the tool used in this study performs these determinations.

On the other hand, matching a project to the assumptions of a model may not be easy. The assumptions need to be as close to the actual project testing and operational environment as

possible. These include features such as test inputs randomly encountering faults, effects of all failures being independent, test space covering the use space, all failures observed when they occur, or faults removed on discovery are not counted again. For example, assumptions of the Schneidewind and Musa models shown in Table 3 are extensive and differ. The development and test staffs need to interact with the analyst to determine which assumptions hold for a specific project. The project staff may identify as many true or false assumptions as possible. After that, it becomes a guessing game. The modeling tools do not perform this function.

Capability refers to the ability of a model to estimate other reliability measurements such as the mean-time-to-failure (MTTF) or the confidence intervals for estimated parameters. The tool used in this study provides these measurements for each model.

Each model may accommodate different development and operational environments and therefore should consider features such as evolving software, failure severity classification, incomplete failure data, multiple installations of the same software, and project environments departing from the model assumptions. A model's applicability is determined by how well it accommodates a project's special features.

Simplicity refers to three time-consuming and expensive parts of reliability measurement: the data collection process, the modeling concepts, and implementation. Data collection and tools to implement the models are discussed in Sections 3.2 and 3.3 of this report. The simpler the modeling concepts, the easier to understand the assumptions, estimate the parameters, and interpret the results.

A model's results should not be biased by noise. For software reliability, the usual noise is the time component. For hardware, calendar time may provide a continuous variable, especially for wear-out, or may provide a continuous execution time. For software, failure data usually are provided in calendar time, rather than execution time, but calendar time does not necessarily provide a continuous variable. Adjustments need to be made to get as close as possible to execution time. Calendar time can be especially difficult to adjust if the data collection system only records the date of the failure but not the number of hours (or days) spent testing. The analyst can assign a test interval of a day but then must know whether testing occurs on weekends and holidays. Or, the interval could be a week but possibly no testing was performed during a period of one or more weeks. Correct modeling relative to the time interval is essential for model validity.

Data collection. Data collection of any type, in any environment, in any domain, is almost always resisted and difficult to achieve. For that reason, the analyst must clearly state the objectives for the data and request only the data essential to successful use of the models. Because the right kind of data is crucial to the success of software reliability modeling, Section 3.2 describes features of data collected in a current data collection system at GSFC and the transformation of that data for use with software reliability models.

The analyst must remember that if data requests are too intrusive on the project, costs and schedules suffer and project cooperation may go rapidly to zero. The project and the analyst will benefit from discussions about the project's development and test processes and about the system

description. The analyst must keep the data collectors motivated, get access to the data quickly and review it promptly.

Parameter estimation. Three common methods for parameter estimation include the method of moments, least squares, and maximum likelihood estimation. The tool used in this study performs maximum likelihood estimation, the most commonly used approach.

Model validation. All of the software models in the SMERFS^3 tool have been used in industry, often the industry for which the model's designer first exercised the model. When the domain changes, then, one needs to examine the model's assumptions very carefully to assure that the model is valid in this new domain and even for a new project within a domain. Should we expect the models to produce similar results for a specific project? One answer comes from Littlewood, "Different software reliability models can produce very different answers when called upon to predict future reliability in a reliability growth context. Users need to know which, if any, of the competing predictions are trustworthy. Some techniques are presented which form the basis of a partial solution to this problem. In addition, it is shown that this approach can point the way towards more accurate prediction via models which learn from past behaviour" [Little].

Dr. Farr 's chapter in [Lyu] cautions us to be careful about a model's assumptions, for some are unforgiving, for example, Schneidewind's model assumes the time intervals over which the failures are observed are equal. Others may be violated, such as the distributional one about the number of failures per unit time, and still credibly fit the data. The Brocklehurst-Littlewood chapter in [Lyu] provides a partial solution to this problem. It is difficult to characterize programs that fit specific models because programs differ so widely in the problem being solved, development practices, architecture, degree of fault tolerance and other features. The models themselves make fairly crude assumptions about what may be a complex failure process. A way around this situation may be to evaluate each model's predictive accuracy upon each data set that is analyzed, that is, compare a prediction with an actual observation of a program.

This potential solution to model validity may seem complex, but the discussion of SMERFS^3 in Section 3.3 suggests that this approach is manageable. SMERFS^3 exercises several models, which are either time-between-failure models or failure count models. The time to run the program is trivial. The real problem is that of collecting and preparing the data at frequent intervals.

Analysis. Once all the previous steps have been performed, the analyst executes the tool for the selected model(s). The difficult part is that the analyst must study the results. One reason is to determine if the timeline should be changed. Another is to decide what values to enter to enable predictions. The analyst may have selected several models and may need to determine which is the better fit for the project.

Software reliability modeling is only one of many methods to aid in measuring software reliability. Others may include inspections, testing, change control boards, metrics gathered from these and other methods. In a paper addressing software reliability modeling for *Shuttle* software, Schneidewind suggests that perhaps most important for understanding software reliability measurements are experience and judgment [Schneid].

3.2 Collection of Data

Many projects at GSFC already collect nonconformance data during test and operation. To place failures in the context of nonconformances for this study, failures are nonconformances that caused the system or component not to perform a required function within specified limits, or the termination of the ability of a functional unit to perform its required function, or caused program operation to depart from program requirements.

At GSFC, some projects use a Distributed Defect Tracking System (DDTS) to collect nonconformance data. The data are used primarily for managing and tracking non-conformances. For reliability prediction and estimation, it is important for the analyst to review the data when it is collected and to keep information about the system and all the activities in developing, testing, and debugging and correcting it. The analyst may not have direct access to the DDTS.

Several projects have their own implementation of the DDTS, each possibly with slight variations in the data collected. Each is a distributed system with many people with different project responsibilities having access rights. Data are entered and housed in the resident database for each project's system. In some cases, data from that database may be offloaded elsewhere for other purposes. The data manipulation discussion deals with using the offloaded, rather than the direct database, version of the data. It is likely that an analyst who may not be directly aligned with the project will work with the offloaded data.

Issues regarding the data are:
- data content and validity from the DDTS, and
- sorting of DDTS data by dates, test activities, and components, and
- transformation to data needed by software reliability modeling.

Data content. The software reliability modeling data requirements appear to be simple: usually either the number of faults in a time interval or the time between the fault occurrences. The problem is the large amount of information that must be known to reduce data to that simplicity, unless the DDTS collects data in exactly the right form. Most likely, for a specific project, data from all testing activities for all components are entered into the system. Therefore specific information that the analyst must be able to extract consists of the activity that found the failure, the date of the failure, severity of the failure. Other dates such as date the correction began and date the correction was completed are important for the proposed improvements to software reliability modeling. The analyst must be able to define equal time intervals, e.g., 1 day. When the timeline is long and occurs over holidays or weekends, then it is important to know if testing occurred during those periods. When failures are not recorded for an interval, the analyst needs to know if testing occurred during that time. Was fault correction occurring? The size or number of test intervals must accommodate the fact of any interval with more than one tester.

Because the defect tracking systems appear to be used across the entire project, the analyst should understand the testing approach concerning integration of components and how they are identified within the tracking system. Data must be sorted by components as they are tested. The analyst needs to discuss with the project staff exactly what data are stored in the DDTS to be able to get valid data for the modeling. If the project does not collect appropriate data, then either

modeling cannot be used for that project or the requirements for data collection need to be changed. It would be best for the analyst to look at a project's DDTS frequently to validate that the data have been entered correctly and in a timely manner.

Some initial considerations for modeling purposes include:

- the data should be collected from integration test through operation
- extensive changes should not be routinely made, that is, software cannot be changing so fast that data gathered one day is radically different from another day
- the data should provide an indicator of the type of testing, e.g., component testing, integration testing, or system testing, including names of components.

To emphasize again, the analyst must have information about the validity of the data, including but not limited to its history, other events occurring when the data were collected, how accurate the date submitted is relative to the date actually found and especially the dates of testing.

Some of the models may be used at unit or integration test, but most are used during system testing and operation. Failure data in the form of time between failure or the failure count within equal time intervals is the input to software reliability modeling. Each model uses data that meet assumptions about failure rate and intensity and fit a curve implied by the model's mathematics.

Data manipulation. When the analyst has direct access to data in a project's DDTS many of the difficulties may be eliminated because of the ease of sorting and transforming that may be provided by database capabilities. Unless the data are provided to an analyst in a database or spreadsheet, the offloaded data consist of one text file for every database record. Some effort must be expended to transfer the data into a spreadsheet or other medium such that an analyst can filter and rearrange the data to satisfy the input requirements of any software reliability modeling tool.

The effort consists of developing a script to reduce the test files of data to only those fields of interest. Then the data need to be put into a database or spreadsheet. The data may be further reduced by some of the project parameters, for example, faults at severity 1-3 may be corrected at a higher priority than those at severity 4 and 5. The models would use only those at the higher priority of correction. Data may be needed only for certain components and must be sorted out. The data for an analysis must come from the same testing activity and are extracted from the date of discovery and the number of faults found on the same date. The specific tool used in this study requires a text file of either the number of faults found in every time interval or the number of time intervals between faults. This transformation may require considerable manipulation of the DDTS data.

3.3 Software Tool Availability

Software support is a valuable resource because the reliability models require several complicated steps engaging mathematical or statistical algorithms. Several software tools have been built to implement several of the models. Recent WEB searches identify basically the same tools from a survey conducted in the early 1990s and those listed in the AIAA recommended practices [Stark] [AIAA]. A compact disc (CD) containing several tools is provided with the *Software Reliability Handbook* [Lyu]. While some tools implement a single model, a couple of

tools exercise several of the models. Both CASRE and SMERFS are included on the CD in Lyu and both employ several of the models.

CASRE uses the code of the algorithms of the models from SMERFS. SMERFS has been modernized and is now called SMERFS^3 to indicate the latest version. Table 5 shows the models contained in SMERFS^3[1]; these are known to be used in industry. While this version has not been officially released, it is easier to use than earlier versions because of the user interface. One method for selecting and using new technology in industry is to have an expert in the technology provide guidance [ZELK98]. Because expert help was available from Dr. Farr, SMERFS^3 became the baseline tool supporting this study.

Table 5. Software reliability models in SMERFS^3

Interval Data Models
Brooks and Motley's Binomial Model
Brooks and Motley's Poisson Model
Generalized Poisson Model
Non-homogeneous Poisson Model
Schneidewind's Model
Yamada's S-Shaped Model
Time Between Failure Models
Geometric Model
Jelinski / Moranda Model
Littlewood and Verrall Linear Model
Littlewood and Verrall Quadratic Model
Musa's Basic Model
Musa's Logarithmic Model
Non-homogeneous Poisson Model

Once the analyst has understood the project's characteristics relative to assumptions of the models but is unsure which model is best, the analyst may reasonably permit SMERFS^3 to select the appropriate models. SMERFS^3 provides other services that free the analyst to worry only about the goodness of the failure data and the interpretation of the results. The tool computes the parameters needed for the various models. It performs the maximum likelihood function to determine model validity. It provides confidence intervals for the parameters for the models having them. It allows the user to select a specific model or to have SMERFS^3 select those that are appropriate. All interval data models use the same input and all time-between-failures use the same. The program provides some transformation assistance between the two types of data. To exercise several reliability models on a data file of over 300 time intervals took only seconds.

On the downside, SMERFS^3 has not been completed and is missing the ability to save output. Only the plots can be printed directly from SMERFS^3. Saving the output can be accomplished by saving each screen of output to and printing from another file, such as a WORD file. These deficiencies are more of a nuisance than they are serious.

SMERFS^3 is only a tool. It does not interpret results, that is, it will not replace the intellectual ability of the analyst. The analyst must rely on knowledge about the project and judgment to

[1] SMERFS^3 also computes hardware and system reliability models but our study addresses only software.

understand what the results mean relative to the system's reliability. Even though the tool computes all the mathematics, a fair amount of understanding of the underlying mathematics and statistics is needed to interpret the results. The tool does perform curve fitting on the input data and will refuse to exercise the data on inappropriate models. Of the models it does exercise, results may vary widely. The parametric values are provided, but the engineer needs to understand them to decide which of the output results are meaningful. After an engineer or analyst acquires experience, interpretation and judgment may become easier and quicker. Execution time is in seconds but the time to prepare the data may be substantial. The time to interpret results, alter the length of the data set, and repeat the process may be overwhelming.

An example of conducting software reliability modeling with SMERFS^3 is provided in Appendix A.

3.4 Options for applying software reliability modeling at GSFC

Many of the projects at GSFC are large and complex and will support important NASA missions. Project managers may find software reliability modeling to be a useful tool in estimating testing times and assessing system reliability. As with any technology new to them, they need to consider the best way to implement that technology. Three possible approaches include these:

- project staff apply the technology
- project and SATC staff work as partners in applying the technology, and
- the SATC provides a service, such as the existing service for complexity analysis.

Usage by project staff. Under this option, the project staff may need formal help in getting started. This could come from training provided by one of the experts in software reliability modeling. The difficulty here is that often an expert who is also the designer of a model teaches to that model. GSFC staff ideally would want an expert who can and will instruct in several of the models. GSFC staff would need to understand several models in order to select the most appropriate for their project. It would be useful to have detailed tutorial information about the models, especially in interpreting results. Finally, for the first usage it would be useful to have an expert monitor. The monitor could assist the staff in installing and using a software reliability tool and in preparing the data. However the greater benefit would come from relating project characteristics to model assumptions and in interpreting model results.

Partnering. In this arrangement project staff would have training as if they were using the technology by themselves. SATC could assist in the initial data preparation, provide guidance in the various places of using the modeling tools where intellectual intervention and interpretation are needed, for example, in iterating with different sample sizes and in making predictions, or provide guidance in interpreting the final SMERFS results. Once a project becomes familiar with software reliability modeling, it may not need as much SATC involvement in interpreting results.

SATC service. A project could request SATC to exercise software reliability modeling on project data. The amount of effort by SATC would depend on several items, e.g., the point in the project that SATC becomes involved, the format of the data when given to SATC, the period of time over the project's life when SATC is involved, and the amount of analysis required. It would be best to have SATC work with the project initially to understand how the various

models' assumptions relate to the projects. Access to the defect tracking database may reduce SATC's effort in converting data to tool input. In an arrangement between SATC and GSFC projects, GSFC project staff would not need to be expert in the models but would need to understand how to use the results. SATC can apply its knowledge of statistics to interpreting the results. SATC staff would need understanding of the models implemented in the tools they use, but after experience with two or more GSFC projects they may find that GSFC projects have similar characteristics and can use the same models.

Musa states that "practitioners have found it is necessary to strictly limit the number of models they work with. There is a substantial effort involved in becoming sufficiently familiar and proficient with a model to be able to insightfully relate that model with your software product and development process. You will be collecting and interpreting data for your environment and the effort in doing this for numerous models is impractical. … Two models seems to be a very good solution, and three appears to take you past the point of diminishing returns" [Musa]. The problem with restricting the use of software reliability modeling to only two models is that the analyst may not know a priori which models are appropriate to the data.

4. Modeling the Fault Correction Process

In general, software reliability models have focused on modeling and predicting failure occurrence and have not given equal priority to modeling the fault correction process. However, there is a need for fault correction prediction, because there are important applications that fault correction modeling and prediction support: predicting whether reliability goals have been achieved, developing stopping rules for testing, formulating test strategies, and rationally allocating test resources. Because these factors are related, we integrate them initially into the Schneidewind model.

Our modeling approach involves relating fault correction to failure prediction, with a time delay between failure detection and fault correction, represented by a random variable whose distribution parameters are estimated from observed data. Our original contribution is the quantification of the relationship between fault correction delay and reliability goals, which provides the software engineer with information for making informed decisions about meeting reliability goals, developing test strategies and allocating test resources. In addition, we contribute to the state of the practice by providing a comprehensive model with both failure detection and fault correction predictions.

It is imperative to include fault correction in reliability modeling and prediction because without it, predictions will understate the reliability of the software. It is important for the field of software reliability engineering to give more emphasis to this issue. Our research is an attempt to provide a framework for integrating failure prediction and fault correction. We have developed a number of equations for assessing the improvement in reliability resulting from fault correction. In addition, we have shown examples of how they apply to stopping rules for testing and prioritization of tests and test resources. We shown that the *number of remaining faults* is better than the *fault correction rate* for assessing reliability and prioritizing tests because the former

can be used with a reliability threshold to predict how much testing would be required to meet the reliability goal.

We found that the most important factor in fault correction modeling is the delay between failure detection and fault correction. We modeled this factor, using the concept of a fault correction queuing service with exponentially distributed delay – a highly statistically significant empirical result based on Shuttle data. We assumed that fault correction would commence with failure detection and that the delay would be equal to fault correction time. This assumption is valid for organizations that choose to keep their software updated with corrections in the current release.

An operational increment (OI) is a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements. We obtained good validation results for two of the three OIs evaluated. The third OI served as a lesson learned that we will apply to future work on the NASA Goddard software project. We will investigate whether fault corrections are postponed and, if so, whether we can model this delay. This will be a challenge because, whereas failure detection is a machine process, fault correction is part human process (deciding when to implement a correction and analyzing how to make the correction) and part machine process (verifying the correction on a computer). The human component is difficult to generalize for inclusion in the model. It may be necessary to use an empirical distribution on each project or set of projects.

We were unable to validate *maximum fault correction delay* because, as stated, many fault corrections in the Shuttle are postponed to future releases. However, our theoretical finding that the maximum delay is independent of its distribution is significant. This delay is based on the assumption that corrections are made in the current release. In a future effort, we will attempt to collect data with various delay distributions to confirm or reject this finding.

Appendix B includes a complete discussion of the challenges of model validation, given the difficulties in collecting complete and accurate failure and fault data. It includes the equations for this fault correction improvement. Finally, it includes a summary and conclusions about the applicability and validity of the model.

5. Conclusions

Software reliability modeling may provide one measurement to be used in combination with other metrics to assess a software system's reliability. Software reliability modeling is a class of mathematical and statistical algorithms based on the mathematics used in hardware reliability modeling. The variable time works well in general reliability modeling because the failures occurring in materials are usually caused by depreciation of the materials over time. Many other factors may cause software failure and one in particular is that software failures are due to faults in the design whereas design faults are generally worked out before materials are manufactured. Nevertheless, the software reliability models evolved from general reliability theory.

This project has shown that

- Most of the mathematics used in reliability engineering has been applied to software reliability engineering.
- Modifications to some software models may accommodate some differences between hardware and software.
- Major effort is needed to introduce software reliability modeling techniques into an organization.
- Use of the models requires careful data collection and transformation. The process may possibly be made simpler by developing macros for manipulating the data in databases and spreadsheets.
- Because no two models provide exactly the same answers, care must be taken to select the most appropriate model for a project and in not giving too much weight to the value of the results.
- Results of software models need to be carefully interpreted and weighed along with other metrics found during development of a software system.
- At least one public domain software tool, SMERFS^3, exists and removes some of the guesswork and complexity in using these models. The tool will exercise several models within seconds.

A recommendation from this study is to use the reliability measurements from modeling as only one of many data points for an assessment of a system's reliability. This measurement method may be introduced to NASA as a SATC service or may be used directly by the projects, perhaps partnering with SATC.

Another recommendation is to consider new methods of measurement, such as non-parametric models for software reliability. Finally, another recommendation is to consider defining a framework of all the metrics across people, process, and product that contribute to software reliability and a method to weigh and sum the metrics.

6. References

[AIAA] American Institute of Aeronautics and Astronautics, *Recommended Practice for Software Reliability*, ANSI/AIAA R-013-1992, February 1993.

[Farr] Farr, William B., "Software Reliability Modeling Survey", *Handbook of Software Reliability Engineering,* Michael R. (Ed), IEEE Computer Society Press, McGraw Hill, 1996.

[Fried] Friedman, Michael A. and Jeffrey M. Voas, *Software Assessment: Reliability, Safety, Testability,* John Wiley & Sons, Inc., 1995.

[GPG] Corrective and preventive Action, Goddard Procedures and Guidelines, GPG 1710.1E, November, 1999.

[Hamlet] Hamlet, Dick, "Are we testing for true reliability?" *IEEE Software,* Vol. 9, No. 4, July 1992, PP. 21-27.

[I610] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std.610.12-1990, The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY 10017, USA.

[I982] IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std.982.1-1988, The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY 10017, USA.

[I1061] IEEE Standard for a Software Quality Metrics Methodology, IEEE Std. 1061-1992, The Institute of Electrical and Electronics Engineers, 345 East 47th Street, New York, NY 10017, USA.

[Little] Littewood, Bev, Abdel Ghaly, A.A., and Chan, P.Y., "Tools for the Analysis of the Accuracy of Software Reliability Predictions," *Software System Design Methods*, Edited by J. K. Skwirzynski, NATO ASI Series, Vol. F22, Springer-Verlag, 1986, PP.299-333.

[Lyu] Lyu, Michael R., Editor, *Handbook of Software Reliability Engineering,* IEEE Computer Society Press, McGraw Hill, 1996.

[Mann] Mann, Nancy R., Ray E. Schafer, and Nozer D. Singpurwalla, *Methods for Statistical Analysis of Reliability and Life Data,* John Wiley & Sons, 1974.

[Mil338] Military Handbook Electronic Design Handbook, MIL-HDBK-338B, U.S. Department of Defense, 1 October 1998.

[Musa] Musa, John D., *Software Reliability Engineering*, McGraw Hill, 1999.

[Neu] Neufelder, Ann Marie, *Ensuring Software Reliability,* Marcel Dekker, Inc., 1993.

[Pham] Pham, Hoang, *Software Reliability,* Springer, 2000.

[Schneid] Schneidewind, Norman F,, "Reliability Modeling for Safety Critical Software," *IEEE Transactions on Reliability*, Vol. 46, No. 1, March 1997, PP. 88-98.

[Zelk] Zelkowitz, Marvin V., and Dolores R. Wallace, Validating the Benefit of New Software Technology, *Software Quality Professional,* American Society for Quality, December 1998.

Appendix A. Performing Software Reliability Modeling

To understand better the process of software reliability modeling described in Section 3 of this report, we applied it to two GSFC projects with data collected during integration of a component of each project. Fortunately software tools provide the curve-fitting and goodness-of-fit tests needed to model the failure data. They compute the algorithms for each model and plot the results. One tool that reduces the difficulty of software reliability modeling is the Software Modeling and Estimation of Reliability Functions for Software (SMERFS), developed under the direction of Dr. William B. Farr of the Naval Surface Warfare Center, Dahlgren, Virginia. It performs the curve-fitting, model selection and execution, and statistical analysis for several software reliability models. Except for user features, features of SMERFS concerning the models are likely to be similar to any other software reliability modeling tools. [2]

Dr. Farr generously allowed us to use a copy of the tool in its current but as yet unreleased version, called SMERFS^3. He also provided guidance to us in using the tool and in understanding the modeling process. This version of SMERFS^3 has a modern graphical interface that made it easy to use immediately on our Windows PCs.

We describe this experience with software reliability modeling to characterize the advantages and the constraints of using this measurement technique. While the tool reduces manual computations, questions remain about other tasks. For example, what other tedious tasks may not be reduced and how much knowledge of the mathematics would a user need to interpret the results? We show pitfalls that may entrap those who do not analyze their project characteristics and data before exercising SMERFS^3 on failure data. We describe the steps necessary before applying SMERFS^S and the process of using the tool. We show examples of input and output. Finally we discuss some lessons learned.

A.1 Initial Process Steps

Data from two projects was available to us from the defect tracking systems (DDTS) they employed. In one case we had no other information about the project and in the other we at least had a description of the variables in the defect tracking system and some information about testing schedules. We could not develop a true characterization of the operational environment or the various test activities and their relationships to the pieces of the software system. Given those constraints, we prepared the data for the modeling process.

Of the many data fields of both projects, only a few were significant. These were the dates the failures occurred, the activity or phase in which they were found, and their severity level. The severity level mattered because corrections of non-conformances at lower severity levels were deferred and may have reappeared in later testing. The activity or phase was the only information for sorting the failure data by software component or subsystem. The date provided the link to time-between-failure and to the failure count for a time interval. If we chose time

[2] This tool also provides hardware reliability modeling and system reliability modeling taking into account both hardware and software failures.

intervals of a day, we had to find calendars for the dates involved to eliminate weekends and holidays. While it was much simpler to use intervals of a week or a month, we still needed to map those time periods properly and to ensure we had enough data for these larger intervals. We chose months only for the second project. Regardless of whether the data was in a database or spreadsheet, manipulation and reduction to forms needed by the models took a significant amount of time. The process is error-prone and requires verification.

A.2 Exercising the Models

SMERFS^3 is very easy to execute because its pull-down menus are concise and leave little room for misunderstanding. The user is asked to specify whether the input is time-between-failure or interval data. For time-between-failure data, the menu in Figure A1 requests more information about the data format; for software models only the first two columns are needed. For interval models, failure counts for every interval, even those without failures, must be provided.

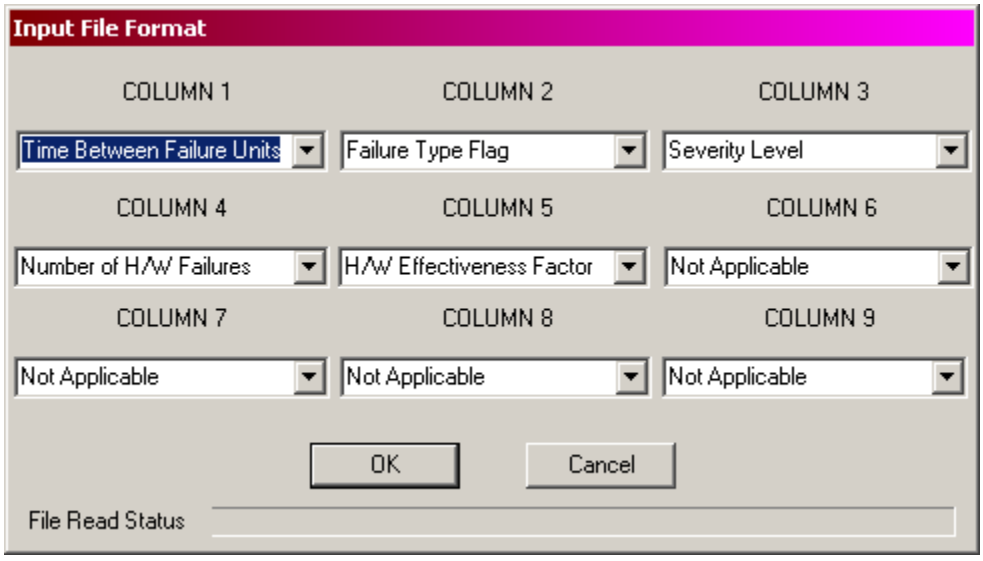


Figure A1. Format for TBF Data Input

The user may select the models to be executed but unless the user has already selected a model from previous use with the same type of data, it is better to let SMERFS^3 do the selections with accuracy analyses for the models. If the data are grossly inappropriate for a model, the tool will inform the user.

A.3 Sample Executions

We applied software reliability modeling to two projects that have data spanning several years. For Project 1 we chose integration testing for subsystem 1 spanning 26 months, or, 110 weeks. The data indicate that only 1 failure occurred at month 12 and at month 26 with no failures during the intervening months. Month 11 had 27 failures with a peak of 28 failures in month 2. WE chose interval data, but the intervals appeared too large for monthly data. We exercised the data at 69 weeks, and 110 weeks to see if predictions made at 69 weeks would indicate the full 130 faults found at 110 weeks. There were 129 faults at 69 weeks. Not all curves for the

acceptable models show because some produced same results, e.g., the two Brooks-Motley models, the Generalized Poisson models, and NHPP and Schneidewind Treatment 1. In Figure A2 observed faults are indicated by the squares.

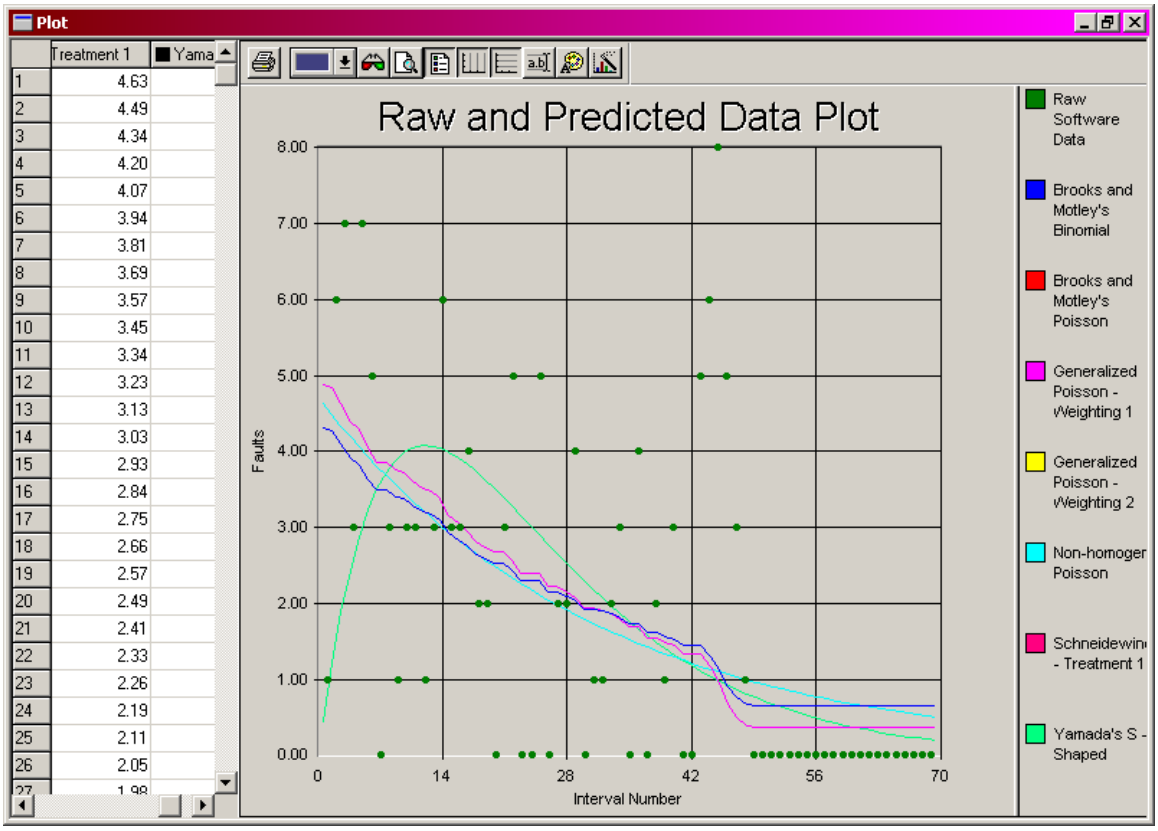


Figure A2. Results for Weekly Intervals, Integration Test, Subsystem 1.

The number of observed failures varied widely in the first weeks, as is anticipated. Then, the number of failures leveled off and reliability growth occurred. For each model, a chart such as in Figure A3 indicates the estimate of remaining failures. The users enters data so that the model produces a prediction for the number of faults remaining by using variations of each model's algorithms with its calculated parametric values. Of course if the estimated number of faults is less than 1., then a prediction cannot be made. The user may enter data to estimate how many test intervals are needed to find N faults or to make variations of that estimate. Some possible predictions are indicated in Figure A3. This model also produces confidence intervals. All of the models produce Chi-square values to indicate how good the curve fit is to the input data. The analyst uses all of this information to determine which model to use.

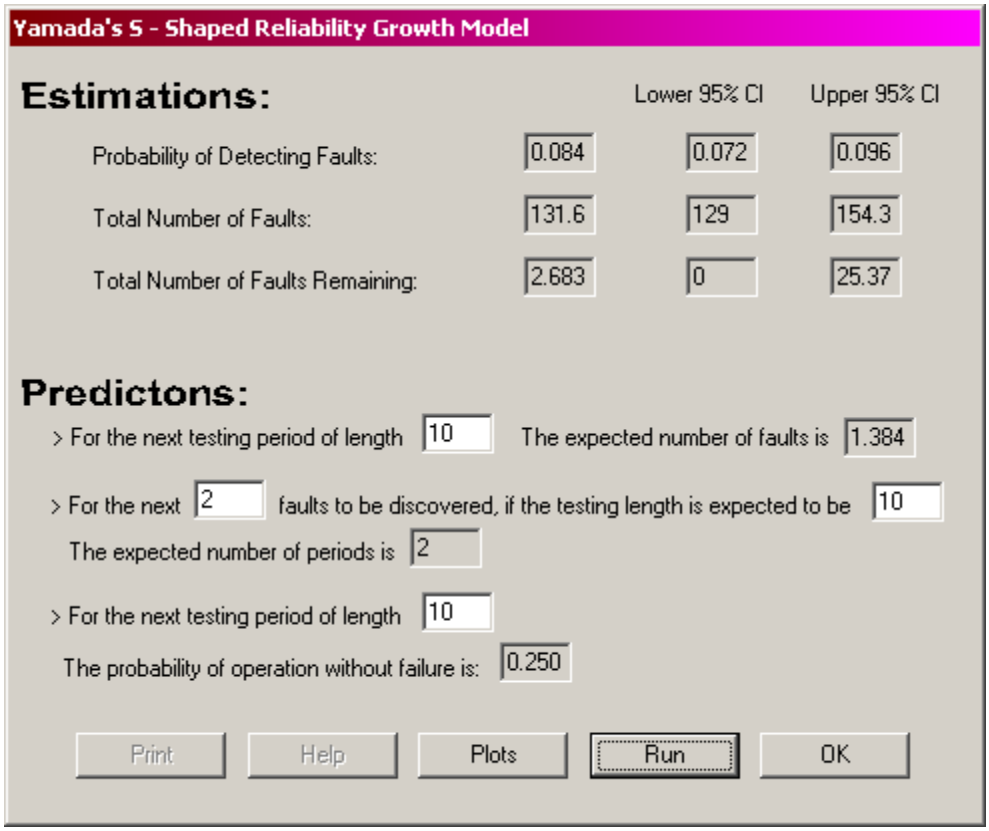


Figure A3. Sample of Output for a Specific Model

A major pitfall in software reliability modeling is selecting the model that best fits the data. Using Chi-square fits and confidence intervals helps as does comparing results of same data using one shorter and one longer set of observed values for predictions. Table A1 summarizes the output for the interval data of 69 weeks, for the models that produce confidence intervals. The model that produced the prediction closest to what we know to be true based on the 110 weeks of data is Yamada. When using software reliability modeling, of course, one is interested in predicting out into the future as far as possible, further than the time of observed failure data.

Table A1. Summary of 3 Models for Interval Data of 69 weeks

	LCI	Prob	UCI	L.CI	TNF	UCI	LCI	TNFR	UCI	Chi2	Pred
GP-2	.027	.035	.043	137.8	139.3	140.8	8.882	10.339	11.85	127	.362
NHPP	.022	.032	.042	129	144	171	0	15	42.3	58.6	4.212
Yama	.072	.084	.096	129	131.68	154.3	0	2.68	25.37	117.8	1.384

GP-2: Generalized Poisson
NHPP: Non-homogeneous Poisson
Yama: Yamada
Prob: probability of detecting faults
Chi2: chi-squared value
TNF: total number failures
TNFR: total number failures remaining
LCI: lower 95% confidence interval
UCI: Upper 95% confidence interval
Pred: number of faults predicted in next 10 Test periods

We also had access to a commercial tool, Loglet Lab[3], which is used for growth estimation for complex data, such as population growth or bacterial growth. Whereas the reliability models generally show exponential decay (negative exponential), Loglet Lab uses exponential growth (positive exponential). In Figure A4, Loglet Lab estimated out to 40 months and indicated zero growth, in this case meaning zero additional failures. By examining results of both tools, we gained confidence in the software reliability modeling results. Loglet Lab does not produce the refined estimation and predictions that SMERFS^3 does by computing the algorithms associated with each model for predictions.

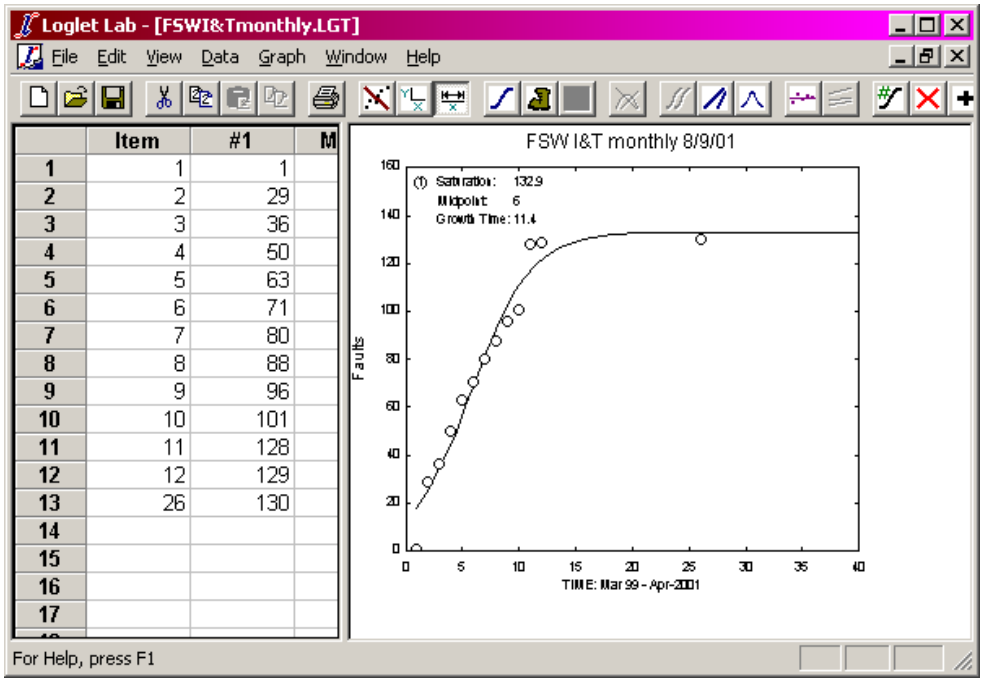


Figure A4. Loglet Lab Results With Monthly Integration Failure Data

For the time between failure models, results for each data set were more diverse. Figure A5 shows the Musa Basic Model for project 2, using data from integration test with daily intervals with weekends removed, while Figure A6 shows the plots of observed and estimated values of the time-between-failure models. Each model provides its parametric values which are then used in the equations for that model to provide predictions. For the time-between-failure models, the predictions are in different terms than those for failure count data. Examples are reliability, mean time to next failure, and intensity function.

[3] Loglet Lab Software, Program for Human Environment, The Rockefeller University, New York, NY.

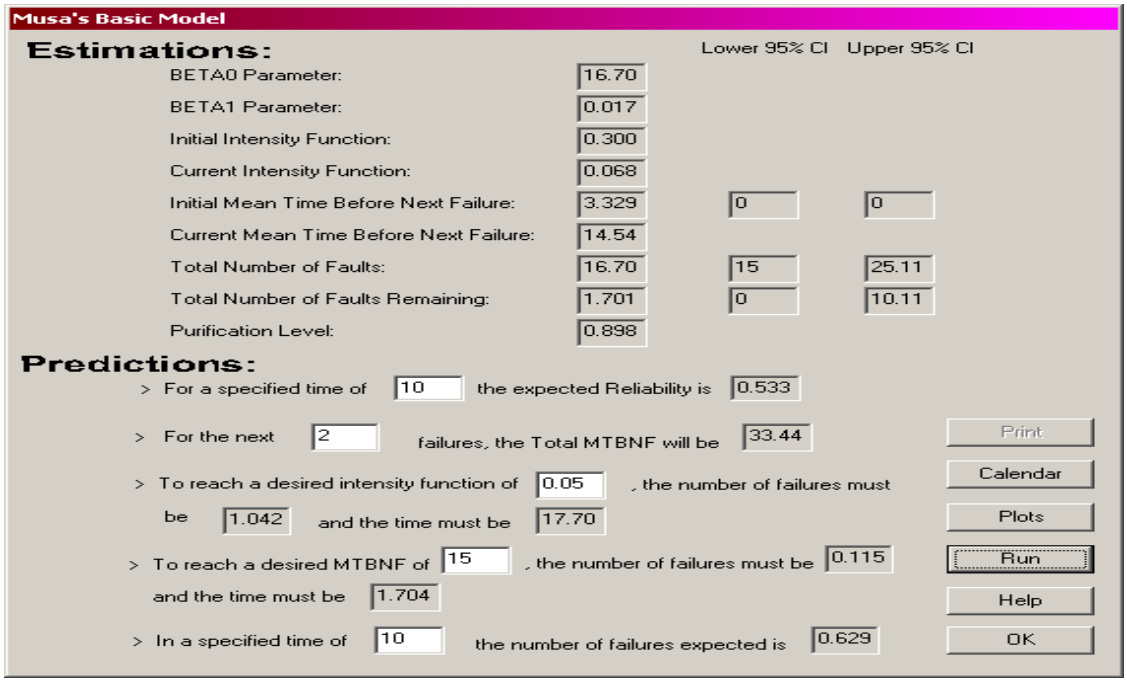


Figure A5. Sample Output for Time Between Failure

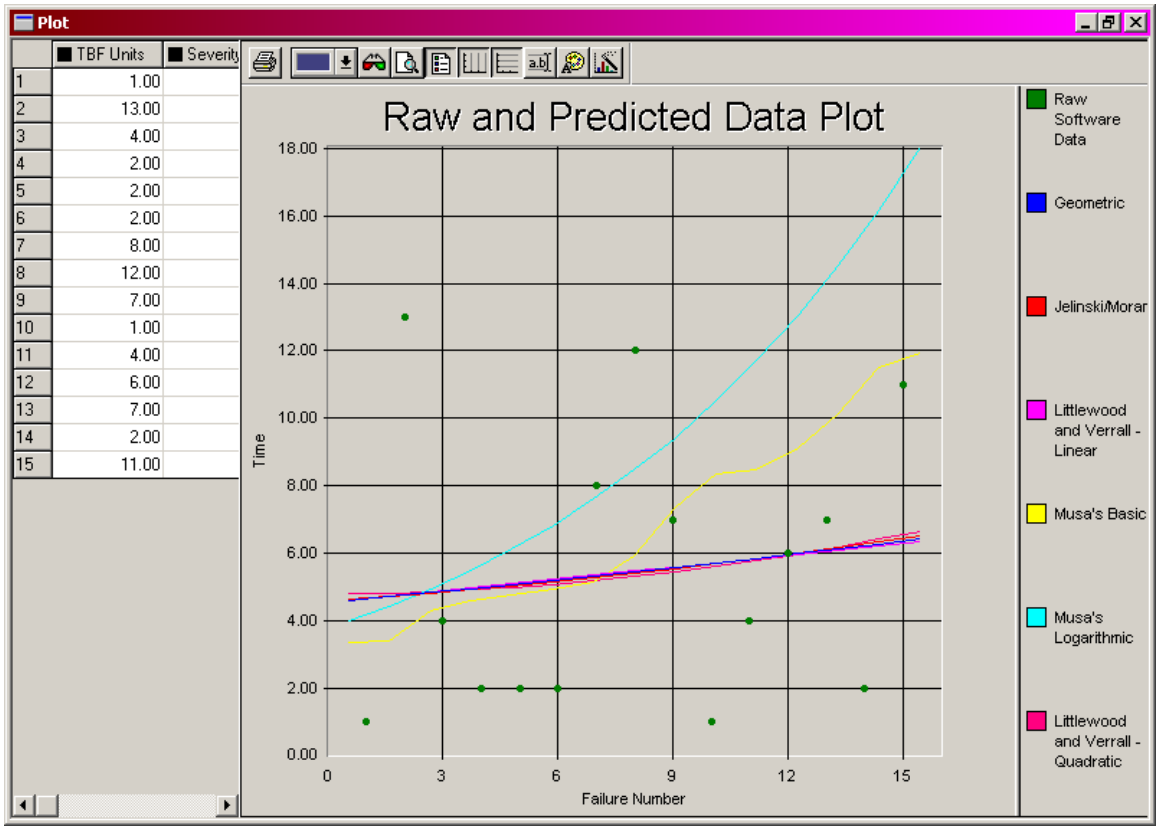


Figure A6. Observed, Estimated Values for Time-Between Failure Models

A.4 Lessons Learned

When we began the software reliability modeling process on the first set of data, we had no experience with either SMERFS^3 or data from any of the defect tracking systems.

To ease difficulty in learning how to use SMERFS^3, we recommend a brief tutorial with one set of input data of interval type and one of time between failure type, along with output and interpretation for each. Within a couple hours anyone could understand how this tool can be used for software reliability modeling. We believe that once an analyst has applied the SMERFS^3 tool, he will have little or no future difficulty utilizing it. While each run with SMERFS^3 is almost instantaneous, an analyst needs to have reports with the charts and plots for people without access to the tool. Saving the data to a file accessible by other people or printers took some manipulation. As SMERFS^3 matures, some of the problems with printing or saving output may disappear. Lessons are to prepare a simple tutorial to be used with SMERFS^3 and to save each screen output to a WORD file. A macro may ease the difficulty of manipulating screen outputs in a file.

Data issues are somewhat more involved and concern either data collection or data manipulation. Most GSFC projects span several years. The defect tracking systems provide data for monitoring the status of non-conformances. Records for test schedules and staffing levels do not appear in the database but without this information the time between failures may be incorrect. It may also be misleading when there are no failures for consecutive intervals because it is not clear if testing has occurred during those intervals. If not, then the intervals need to be adjusted. Similarly two people conducting tests instead of one person changes the interval size. The lesson is that test schedule information and staffing levels should be required information in the tracking system and should be made available to the analyst. Data input must be constructed so that the failures are from the same software; therefore failures should relate to the software. The "activity" field almost accomplishes this, but could be enhanced with another field. The lesson here is to require the software field in the tracking system. If some fault corrections are deferred indefinitely, then they should be labeled accordingly. Addressing these lessons should satisfy the data collection concerns.

The analyst needs to manipulate the data considerably to organize it by software type and test activity, then by data and finally by count. These are manual and highly error-prone tasks. Macros to ease the difficulty of manipulating the project data in a spreadsheet may mitigate these concerns.

The remaining step, interpretation of results, is probably the most difficult aspect of software reliability modeling. It also depends on how well the data were interpreted in the first place. Therefore, the first lesson is that the analyst must work carefully with project staff to understand the organization, test schedule, and operational environment of the software. SMERFS^3 produces statistical information to aid the analyst in selecting the best model for the data and in evaluating the estimates. A second lesson is that training in how to apply the results would remove the final hurdle to making this technology a practical instrument at GSFC for software reliability measurement. Finally, there are some supporting computations that could be handled by a spreadsheet.

Appendix B. Improvements to a Software Reliability Model[4]

There is a need for greater emphasis on fault correction modeling and prediction in software reliability models. This need stems from the fact that the fault correction process is vital to ensuring high quality software. If we only address failure prediction, reliability assessment will be incomplete because it would not reflect the reliability of the software resulting from fault correction. This view is shared by Xie, who laments that fault correction prediction is absent in most software reliability models [XIE92]. In addition to achieving greater accuracy in reliability prediction, there are by-product benefits associated with fault correction prediction as follows:

a. Predicting whether reliability goals have been achieved: If no predictions are made of the number of faults to be corrected, fault correction rate, and fault correction time, accurate prediction of reliability cannot be obtained.

b. Providing stopping rules for testing as follows: (1) The predicted number of remaining faults is less than or equal to a specified critical value and (2) The fault correction rate asymptotically approaches zero.

c. Prioritizing tests and allocating test resources: Software with high values of number of remaining faults and low fault correction rates are given high priority in testing and allocation of resources, such as personnel and computer time.

Because the attainment of reliability goals (item a) is related to test strategies (items b and c), these factors are integrated in our model. Although we illustrate fault correction modeling using a particular model, the approach is general and could be applied to all software reliability growth models.

Our research hypothesis is that fault correction can be modeled with a function that has the same form as the failure detection function but with a random delay that accounts for fault correction time. The validation of this hypothesis is addressed in section B.3. The originality of this contribution is the development of equations that relate the delay between failure detection and fault correction and the allowable remaining faults, as explained in section B.2. Thus, in practice, it would be possible to predict how much delay in the correction process could be tolerated in order to meet reliability goals at a given time in test or operation. One contribution to the state of the practice is a comprehensive prediction model that contains both failure prediction and fault correction. Another contribution is to remind the software engineering community of the importance of the fault correction component of software reliability. Too often, there is emphasis on the failure and fault detection process, as reliability criteria, to the exclusion of the fault correction process. For example, Jeske and Qureshi state that the failure rate is an important criterion in deciding when the software is ready to be released and they assume that when faults are discovered, they are immediately removed; apparently, faults are removed with zero

[4] Dr. Norman Schneidewind of the Naval Postgraduate School, in Monterey, CA, has written this paper in fulfillment of the cooperative research effort between him and SATC. Variations of this paper will appear in proceedings for the ISSRE '01 and Metrics 2002 conferences.

correction time [JES00]. Huang et al indicate that the number of faults at the beginning of test will not normally be changed except for imperfect debugging when switching from testing to operation [HAU00]. We observe that whether there is perfect or imperfect debugging, the number of faults will change as additional failures occur and as faults are corrected. Singpurwalla [SIN91] states that the decision to test for additional time T^*, after having tested for time T, depends on the number of bugs encountered in time T. We contend that the decision should depend on the number of remaining faults at time T, as we will explain.

Dalal and McIntosh have an interesting model that combines economic and remaining faults criteria as the rule for when to stop testing [DAL94]. However, their model applies to commercial systems with a large number of faults, whereas our interest is in safety critical systems like the Shuttle with a small number of faults. Additional economic models that recognize the cost of fault correction in the stopping rule are those of Ehrlich et al [EHR 93] and Dalal and Mallows [DAL88]. Although cost is important in commercial software, the focus of our research is on increasing the accuracy of predicting the software reliability of safety critical systems by including the fault correction process.

Gokhale and colleagues have addressed the issue of delayed fault correction, when the delay is caused by the queuing of faults to be removed or by the presence of latent faults that are difficult to remove [GOK97]. However, this is not the same as the delay mentioned in section B.1.1 below that is the result of a decision by the developing organization to defer fault correction until the removal of a fault becomes critical to the operation of the software. They also model the possibility of imperfect fault repair (i.e., a fault may not be entirely corrected or a new fault may be inserted during the repair operation) [GOK96]. They use a non-homogeneous Markov Chain to represent a non-homogeneous Poisson process to model failure detection and fault correction. Their approach is interesting and provides greater flexibility than analytical models like ours in a variety of fault correction scenarios. However, analytical models provide greater visibility of the relationships between factors that influence the fault correction process than do complex Markov Chain diagrams. In addition, their models have not been validated against real-world projects. In contrast, we have provided validation tests in Tables B.1, B.2, and B.3 for the Shuttle.

The delayed S-shaped model has the interesting characteristic of an initial increasing failure intensity function, as the test team becomes familiar with the software, reaches a maximum, and then asymptotically approaches zero with test time, as it becomes more difficult to detect failures. Thus, this model gets its name from a *delay in failure detection* and not fault correction, because an assumption of the model is that faults are corrected immediately without introducing new ones [LYU96].

This appendix contains the following sections: B.1. Fault Correction Prediction Model Components, B.2. Applications, B.3. Validation, and B.4. Summary.

B.1 Fault Correction Prediction Model Components

This section develops the equations of the components of the fault correction prediction model, where all references to “time” are elapsed or wall clock times. These components include the following: failure and fault correction counts, measures of progress in fault correction; and

stopping rules for testing to achieve specified reliability goals. The objective is to provide the software engineer with a comprehensive set of predictions for assessing and controlling the reliability of software in test and operation. In addition, we discuss model assumptions and the consequences to model applicability if the assumptions are not met in practice.

B.1.1 Fault Correction Delay

Our approach to fault correction prediction is to relate it to failure prediction, introducing a delay dT, between failure detection and the completion of fault correction (i.e., fault correction time). We assume that the rate of fault correction is proportional to the rate of failure detection [SCH75]. In other words, we assume that fault correction keeps up with failure detection, except for the delay dT. If this assumption is not met in practice, the model will underestimate the remaining faults in the code. Thus, the model provides a lower bound on remaining faults (i.e., the remaining faults would be no less than the prediction). Using this assumption, the number of faults corrected at time T, C (T), would have the same form as the number of failures detected at time T, D (T), but delayed by the interval dT. Interestingly, in the Smidts' model, she recognizes that fault repair time affects reliability and assumes the same form between fault detection and repair of commission-related faults: both follow an NHPP with different failure intensities [SMI99]. In our model, failure detection and fault correction both follow an NHPP but with a delay factor in the latter. Originally, we used a constant dT, which was estimated from the empirical data [SCH75]. As pointed out by Xie, this assumption is too restrictive [XIE92]. He suggests modeling the delay as an increasing function of test time. However, we have not found this to be the case in either the Shuttle or Goddard Space Flight Center (GSFC) data, where the fault correction time appears to be primarily a function of the difficulty of the correction and independent of when the correction occurs. In order to improve the model, we use a random variable for the delay dT. For the Shuttle, this variable was found to be exponentially distributed with mean fault correction time 1/m, where m is the mean fault correction rate in the intervals dT. This distribution was confirmed for the Shuttle, using a sample of 85 fault correction times and the Kolmogorov-Smirnof test, resulting in $p = 0$. In addition, Musa found that failure correction times were exponentially distributed for 178 failure corrections [MUS87]. For other projects, which may have other distributions, the appropriate distribution would be used in the model, as determined by statistical tests. However, the same overall modeling approach would be used, as we will describe.

In the model, there are the following possibilities: a single failure may occur or multiple failures may occur simultaneously; a single fault may be corrected or multiple faults may be corrected in concurrent efforts; failures may occur as faults are corrected; and a variable delay of dT, which could be zero, must occur between a failure detection and fault correction. In addition, we assume that the fault correction *starts* when failures are detected. This assumption is related to the previous assumption of fault correction keeping current with failure detection. In some cases, a software developer may choose to postpone a non-critical fault correction for several releases because it has obtained a waiver to not make the correction in the current release. We do not attempt to model this human, case-by-case decision process *in the current model*. Our current model is based on keeping the software updated with corrections in the current release. In addition, the model does not include the possibility of introducing a fault when correcting one; there is no data available for the Shuttle regarding this factor. These are important factors, but

they are beyond the scope of this effort; these factors will be addressed in a future effort, involving the use of GSFC data. Fault mitigation methods, such as fault tolerance, (the Shuttle has four active computers and a fifth backup) are reflected in the model as a reduction in the number of failures from what would be experienced with no fault tolerance; this result, in turn, decreases the model's failure rate parameters and predicted number of failures.

It is well known that various human queue service times at supermarket checkout stands, gas stations, bank tellers, airline passenger agent stations, etc. can be approximated with an exponential distribution [KLE75], and can be modeled as a birth-death process, where in our case a birth is a detected failure and a death is a corrected fault. Although the parameters of the correction time distribution would be different from those of the cited applications (e.g., longer mean time), the same type (i.e., exponential) distribution is assumed. Musa uses this type of queuing model in his failure correction process [MUS87]. The concept of a fault correction queue is shown in Figure 1[5].

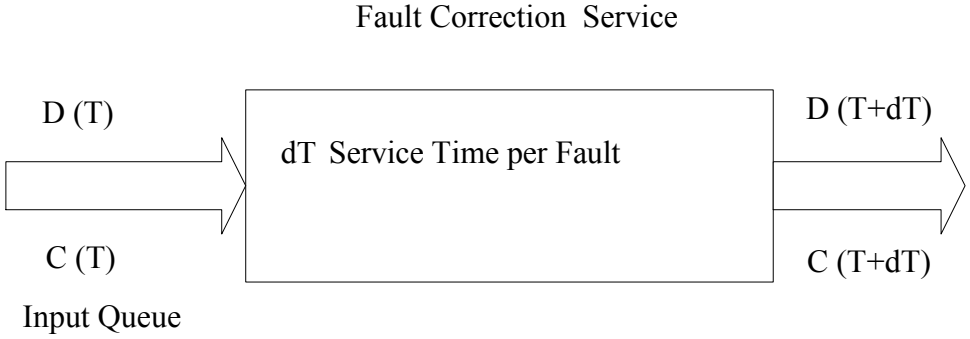


Figure 1. Concept of Fault Correction Service

Due to the great variability in fault correction time that we have found in both the Shuttle and GSFC data, we emphasize predicting limits instead of expected values. This concept is shown in Figure 2. For a given mean fault correction rate m, the cumulative probability distribution F (dT) of the fault correction delay dT is used to specify upper and lower limits of dT. These limits are dT_U and dT_L, corresponding to F_U and F_L, respectively. The concept is to bound the delay time, for example at $F_U = .9$ and $F_L = .1$, as shown in the figure, and to use these limits in the fault correction predictions. Thus, when making predictions, there would be high confidence that actual values lie within the limits (e.g., probability of .80). The equation for F (dT), corresponding to Figure 2 for the exponential distribution, is given by (1):

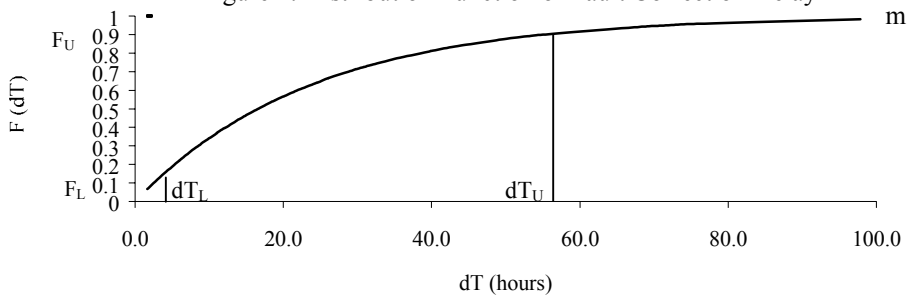
$$F(dT) = 1 - \exp(-(m)(dT)). \quad (1).$$

Equation (1) is manipulated to produce equation (2), which would be used to compute the limits of dT, applying the specified limit values of F (dT):

$$dT = (-\log(1 - F(dT)))/m. \quad (2).$$

[5] Because this appendix was extracted from the complete paper, the figure titles will be numbered from 1 to 7, instead of B1 to B7.

Figure 2. Distribution Function of Fault Correction Delay



In the examples in section B.2, predictions that require F (dT) are made using F_U and *not* F_L in order to provide conservative estimates (e.g., the probability is .90 that the number of corrected faults is less than or equal to its predicted value or .10 that it exceeds this value). The reason for this approach is to mitigate against overly optimistic predictions of number and rate of fault correction.

B.1.2 Number of Faults Corrected

The predicted number of failures detected D (T), for T > (s –1), is given by equation (3) [SCH97]:

$$D(T) = (\alpha/\beta)[1 - \exp(-\beta((T-s+1)))] + X_{s-1}, \quad (3)$$

where the terms have the following definitions:

α : failure rate at the beginning of interval s

β : negative of derivative of failure rate divided by failure rate (i.e., relative rate of change of failure rate)

T: cumulative test or operational time

s : starting interval for using observed failure data in parameter estimation

X_{s-1}: observed failure count in the range [1,s-1].

The parameters α and β are obtained from maximum likelihood estimation techniques [SCH97]. The parameter s is used in the optimal selection of failure data that involves selecting only the most relevant set of failure data for reliability prediction, with the result of producing more accurate predictions than would be the case if the entire set of data were used. The mean squared error criterion, applied to the differences between predicted and actual values in the observed range of the failure data, is used for selecting the optimal value of s. For all equations in which the term T – s + 1 appears, predictions are made for T > (s –1) [SCH92, SCH97].

Using the assumption of section B.1.1 that the number of corrected faults C (T) has the same form as the number of detected failures D (T) but with a variable delay dT, yields equation (4), for T > (s –1), [SCH75]:

$$C(T) = (\alpha/\beta)[1 - \exp(-\beta((T-s+1) - dT))] + C_{s-1}, \quad (4)$$

where C_{s-1} is the observed fault correction count in the range [1,s-1]. Using equation (2), equation (4) becomes equation (5), where we would compute the upper and lower limits of C (T):

$$C(T) = (\alpha/\beta)\,[1 - \exp(-\beta\,(T-s+1+(\log(1 - F(dT)))/m))] + C_{s-1} \qquad (5).$$

B.1.3 Proportion of Faults Corrected

A measure of progress in fault correction of detected failures, at time T, is given by the proportion of faults corrected, as expressed in equation (6):

$$r(T) = C(T)/D(T) \qquad (6).$$

The ideal goal, of course, is to achieve a value of 1. However, the achievement of this goal is constrained by the amount of test time that is economically feasible to allocate to the software under test. This challenge will be addressed in section B.2, when we consider stopping rules and prioritization of tests.

B.1.4 Number of Remaining Faults

The predicted number of remaining faults, after the correction process has been operative for time T, is given by equation (7):

$$N(T) = D(T) - C(T) \qquad (7).$$

This equation is based on the assumption that all the faults that exist in the software have been predicted by D (T). A more conservative prediction is obtained by predicting the detected failures over the life of the software D (T_L), as in equation (8) [SCH97], and then using equation (9) as the predicted remaining faults.

$$D(T_L) = \alpha/\beta + X_{s-1} \qquad (8).$$

$$N(T_L) = D(T_L) - C(T) \qquad (9).$$

B.1.5 Time Required To Correct C Faults

In order to do informed scheduling of test resources, such as personnel and computer time, it is helpful to predict how much cumulative test time would be required to correct a given number of faults during testing. If equation (4) is solved for T, and C (T) becomes a given number of faults C to correct, we obtain equation (10), the predicted time required to correct C faults during testing. If the delay is exponentially distributed, equation (2) would be substituted for dT in equation (10).

$$T_c = [\log[\alpha/(\alpha - \beta(C - C_{s-1}))]]/\beta + (s-1+dT), \text{ for } \alpha > \beta(C - C_{s-1}) \text{ and } C \geq C_{s-1} \quad (10).$$

B.1.6 Fault Correction Rate

The fault correction rate is another useful measure of progress in fault correction. If the rate is decreasing and relatively high, it would be indicative of further gains in fault correction by continuing to test. On the other hand, if the rate is decreasing towards an asymptotic value, it would indicate that further testing would produce little gain in fault correction. The fault correction rate is obtained by taking the derivative of equation (5) to produce equation (11):

$$R(T) = \alpha[\exp(-\beta(T-s+1+(\log(1 - F(dT)))/m))] \quad (11).$$

B.2 Applications

For the purpose of model development and validation, post release failure data from three Shuttle operational increments were used: OID, OIJ, and OIO. An operational increment is a software system comprised of modules and configured from a series of builds to meet Shuttle mission functional requirements. In addition, fault correction data, obtained from Shuttle build inspection files, were used. In order to show model performance on all three OIs , the figures in this section and the tables in section B.3 (Validation) use all three OIs.

This section presents several applications of the model developed in section 2, involving reliability assessment and strategies for efficient testing. Predictions are made for post release failures and fault correction, where OID, OIJ, and OIO experienced 13, 7, and 7 post release failures, respectively (post release failures are sparse for the Shuttle). To make fault correction predictions comparable across OIs, 7 failures were used for each OI (the first seven for OID). All data and predictions are in terms of 30-day intervals. This is the failure count interval used in previous Shuttle reliability predictions [KEL97, SCH97]. Although 7 failures may seem like a small sample size, it is representative of Shuttle post release reliability, and despite the small number of failures, the model is able to predict detected number of failures D (T) fairly accurately (see Tables B.1, B.2, and B.3 in section B.3).

B.2.1 Predicting Whether Reliability Goals Have Been Achieved

Because it is important to gear the fault correction process to the remaining faults, we specify inequality (12), which relates N (T), the number of remaining faults at time T, to R_C, the critical value of remaining faults:

$$N(T) \leq R_C, \text{ or } N(T) = (D(T) - C(T)) \leq R_C \quad (12).$$

When equations (3) and (4) are substituted for D (T) and C (T), respectively, in inequality (12), and assuming that $X_{s-1} - C_{s-1} \cong 0$ because both X_{s-1} and C_{s-1} are small in the range 1,s-1, we obtain inequality (13), the condition for maximum fault correction delay:

$$dT \leq [\log[1 + (\beta/\alpha)(\exp(\beta(T - s + 1)))(R_C)]]/\beta \quad (13).$$

The parameter R_C serves as a reliability threshold. A value of $R_C = 1$ would be appropriate for safety critical systems. If $R_C = 0$, $dT = 0$. This result makes sense because to achieve zero faults, faults must be corrected as soon as failures are detected. We feel that inequality (13) is a significant result, because it says that independent of the distribution of dT, (13) must be satisfied to meet the reliability requirement. Thus for a given value of R_C, we are able to specify the maximum fault correction delay dT that will meet the reliability goal. As a practical matter, the software engineer can control the development and maintenance process to constrain the fault correction delay to (13) by assigning test personnel with the appropriate skills and by allocating sufficient computer resources to the tests.

By plotting inequality (13) in Figure 3, the maximum fault correction delay as a function of given values of critical value of remaining faults, the software engineer can see the effect on the maximum delay of increasing the reliability (i.e., decreasing R_C). Furthermore, a family of plots can be made, with each member representing a different test time T. The software engineer can see that by allowing more test time, for a given reliability goal R_C, the maximum delay can be increased. The parameters that were used in the predictions and plots were estimated using the SMERFS tool [FAR93] and the Shuttle OIJ failure data. The test times are long in the Shuttle because much of the software from previous missions is reused and a given OI is tested essentially on a 24x7 schedule for months, and sometimes years, during pre-release build development, post release system integration, astronaut training, and flight.

The effect of fault correction delay can best be seen in Figure 4 that shows, for OID, the initial lag between cumulative number of corrected faults C (T) and cumulative number of failures D (T), with C (T) eventually catching up to D (T) and progressing in parallel from then on.

B.2.2 Stopping Rules for Testing and Prioritizing Tests and Test Resources

Remaining Faults and Fault Correction Rate

The number of remaining faults N (T), equation (7), plotted as a function of test time, as illustrated in Figure 5 for Shuttle OIJ and OIO, can be used as a stopping rule for testing. We used an upper probability limit of .90, meaning that the probability is .90 that N (T) is less than or equal to its ordinate values for a given test time, or .10 that these values are exceeded. The practical significance of this plot is that it is highly unlikely that reliability could be improved by testing for more than 30 intervals. This figure is interesting because it shows a cross over between the OIs at T = 11.5. This would imply that OIJ should be given higher priority before T = 11.5 and lower priority after it. By priority, we mean the order of testing and allocation of personnel and computer resources. Plotting the fault correction rate R (T), equation (11), would also provide insight for when to stop testing and for prioritizing tests. However, we consider N (T) more useful because it can be used with the critical value of remaining faults R_c -- a reliability threshold. For example, for $R_c = 1$ in Figure 5, testing would be terminated for OIJ at T = 13.5 and for OIO at T = 26.

Figure 4. Predicted Failures and Corrected Faults (OID)

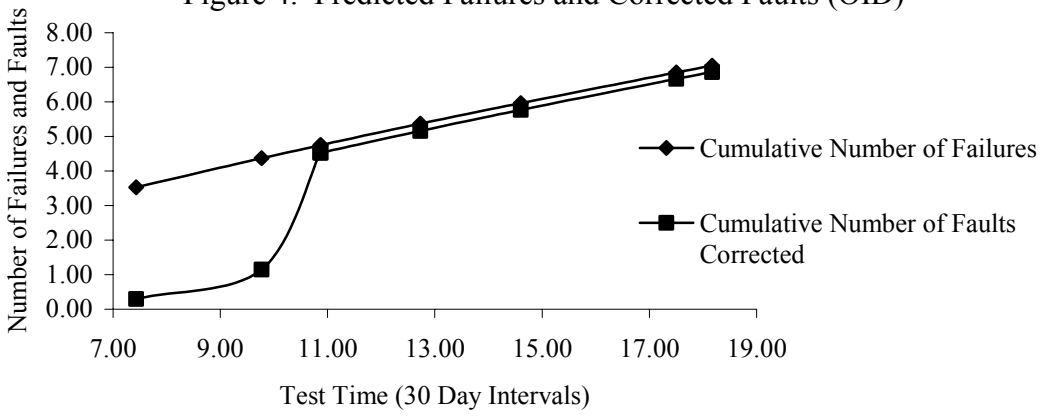


Figure 3. Predicted Maximum Correction Delay (OIJ)

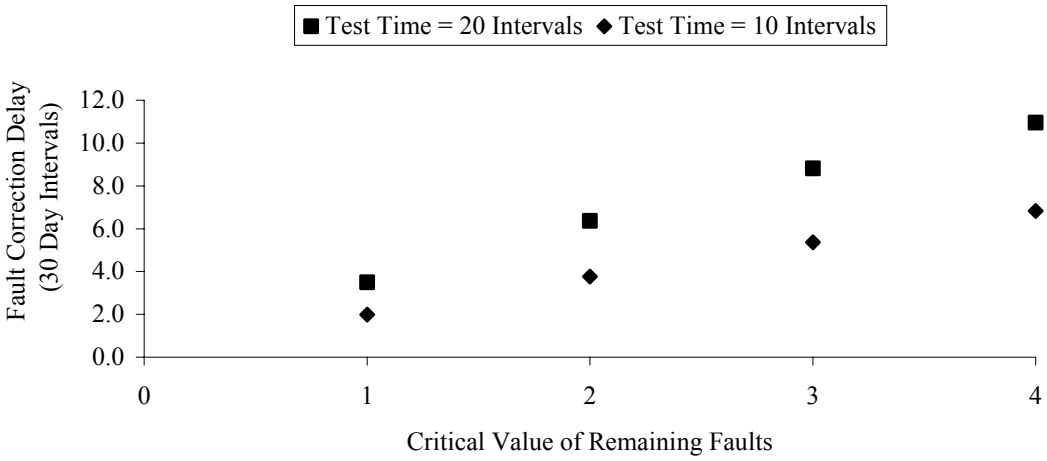
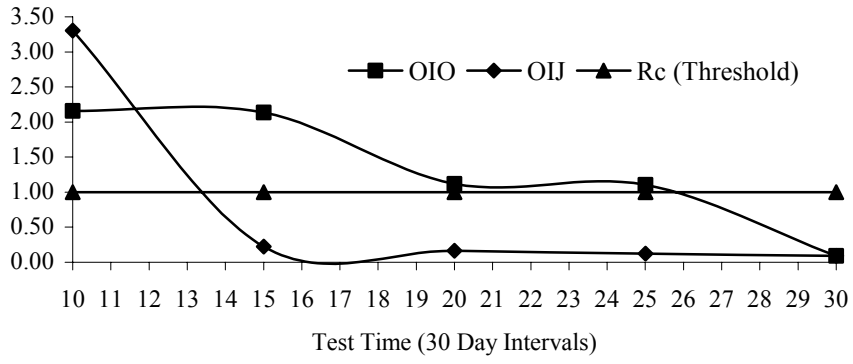


Figure 5. Predicted Number of Remaining Faults



Reliability Improvement

In the previous section, we used absolute quantities (e.g., number of remaining faults) for developing test strategies and assessing reliability. In this section, we use the relative quantity p (T), the proportion of faults remaining at time T, which is related to r (T) from equation (6), the proportion of faults corrected at time T, by equation (14):

$$p(T) = 1 - r(T) = 1 - C(T)/D(T) \qquad (14).$$

A plot of equation (14) is shown in Figure 6 for OIJ and OIO. We do this because two software systems could have experienced different numbers of failures but have equal numbers of remaining faults. In this case, the software with fewer failures would have achieved greater progress in reliability improvement, as measured by p (T). However, the use of a threshold seems more intuitive when applied to N (T). In practice, both measures could be used.

Test Scheduling

We can anticipate test requirements and do proactive test scheduling by using equation (10), the predicted amount of test time required to correct a given number of faults T_c. In addition, a plot of multiple software systems shows how the systems compare in test requirements. An example is shown in Figure 7 for OID and OIO, where the predictions are for the case of zero faults corrected at the time of prediction, and a .90 upper probability limit. Because the values for OIJ are virtually identical to those for OID, the former is not plotted. We see that the difference in test time between the OIs increases with increasing number of faults. This result implies that relatively large quantities of resources – personnel and computer time -- would be required to correct the faults in OIO, and that this need accelerates as the number of faults to correct increases.

Figure 6. Predicted Proportion of Remaining Faults

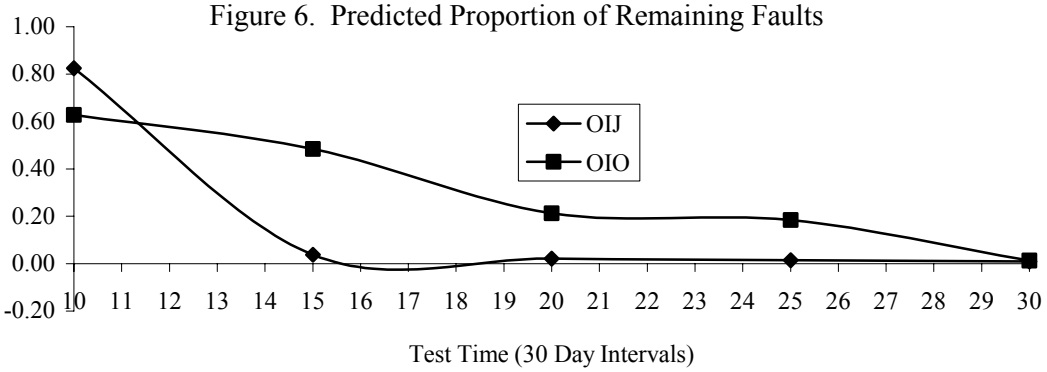
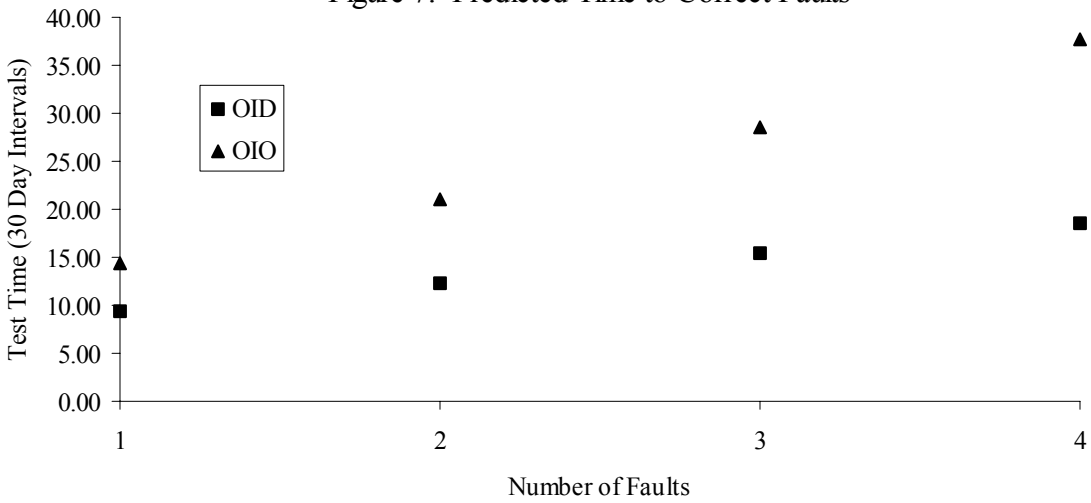


Figure 7. Predicted Time to Correct Faults



B.3 Validation

For the purpose of validation, we ran three scenarios – one each for OID, OIJ, and OIO – shown in Tables B.1, B.2, and B.3, respectively. This involved determining from the collected data when failures occurred and when faults were corrected. The actual delay between failure occurrence and fault correction was estimated by examining, manually, the Shuttle Discrepancy Reports (i.e., reports that document deviations between specified and observed software behavior) to determine the disposition of the fault (i.e., the release and release date on which the fault was corrected). This was a laborious process because although failures are recorded in electronic files, there is no electronic file of fault corrections with correction dates. We had to infer the correction dates from the release dates.

The event column of the tables shows when either a failure occurred or a fault was corrected. In some cases multiple failures or corrected faults occurred in the same interval; these occurrences are signified by the plural form in the event column. The next column shows the test time T when the events occurred followed by the actual values of cumulative number of failures detected D (T), cumulative number of faults corrected C (T), and number of remaining faults N (T), the difference between D (T) and C (T). The next section of the tables shows the predictions for D (T), C (T), and N (T). Notice at the top of each table the statement about the range of prediction, which is for $T > s-1$ (see section B.1.2). For example, for OID in Table 1, $s = 7$; therefore, the predictions start at interval $T = 7.43$.

Table B.1. OID (Predictions for T > s-1 = 6)										
		Actual Values			Predictions			Squared Error		
Event	T	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)
Failure	4.53	1	0	1						
Failure	4.83	2	0	2						
Failure	5.70	3	0	3						
Failure	7.43	4	0	4	3.53	0.30	3.23	0.220	0.09	0.59
Failure	9.77	5	0	5	4.37	1.15	3.22	0.401	1.31	3.17
Corrections	10.86	5	4	1	4.74	4.53	0.22	0.066	0.28	0.62
Correction	10.87	5	5	0	4.75	4.53	0.22	0.065	0.22	0.05
Failure	12.73	6	5	1	5.37	5.16	0.21	0.401	0.03	0.63
Correction	14.60	6	6	0	5.97	5.77	0.20	0.001	0.05	0.04
Failure	17.50	7	6	1	6.85	6.67	0.19	0.022	0.44	0.66
Correction	18.17	7	7	0	7.05	6.87	0.18	0.002	0.02	0.03
					Mean Square Error			**0.147**	**0.31**	**0.72**

Table B.2. OIJ (Predictions for T > s-1 = 8)										
		Actual Values			Predictions			Squared Error		
Event	T	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)
Failure	3.57	1	0	1						
Failure	7.03	2	0	2						
Failure	7.47	3	0	3						
Failure	9.17	4	0	4	3.60	0.28	3.32	0.16	0.08	0.46
Failure	9.23	5	0	5	3.63	0.32	3.32	1.87	0.10	2.83
Corrections	10.60	5	2	3	4.28	2.99	1.29	0.51	0.99	2.92
Failure	13.20	6	2	4	5.38	4.13	1.25	0.38	4.54	7.57
Correction	13.66	6	3	3	5.56	5.32	0.24	0.20	5.36	7.61
Failure	17.17	7	3	4	6.75	6.56	0.20	0.06	12.65	14.47
Correction	24.06	7	4	3	8.47	8.34	0.13	2.16	18.86	8.25
Correction	24.27	7	5	2	8.51	8.39	0.13	2.29	11.47	3.51
Correction	37.43	7	6	1	10.31	10.25	0.06	10.92	18.05	0.89
Correction	37.44	7	7	0	10.31	10.25	0.06	10.93	10.56	0.00
					Mean Square Error			**2.95**	**8.27**	**4.85**

Table B. 3. OIO (Predictions for T > s-1 = 8)										
		Actual Values			Predictions			Squared Error		
Event	T	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)	D(T)	C(T)	N(T)
Failure	5.77	1	0	1						
Failure	5.90	2	0	2						
Failure	7.53	3	0	3						
Correction	9.07	3	1	2	3.20	1.06	2.14	0.04	0.00	0.02
Failures	11.47	5	1	4	3.62	1.49	2.13	1.90	0.24	3.50
Corrections	16.80	5	4	1	4.49	3.37	1.11	0.26	0.39	0.01
Failure	24.67	6	4	2	5.60	4.50	1.10	0.16	0.25	0.82
Corrections	29.40	6	6	0	6.18	6.09	0.09	0.03	0.01	0.01
Failure	36.17	7	6	1	6.92	6.84	0.08	0.01	0.71	0.86
Correction	45.77	7	7	0	7.79	7.73	0.06	0.63	0.54	0.00
					Mean Square Error			**0.43**	**0.31**	**0.75**

The last section of the tables shows the squares of the differences between actual and predicted values for computing the Mean Square Error (MSE) at the bottom of the tables. We computed MSE rather than Mean Relative Error because the latter would have required division by zero in those cases where C (T) = 0. We consider the MSE values for OID and OIO, Tables B.1 and B.3, respectively, to be sufficiently low to validate the predictions. However, we do not reach that conclusion regarding OIJ in Table B.2. The discrepancy between predicted and actual results is due primarily to the long delay between failure detection and the *start* of fault correction that was decided by the Shuttle software developer because the failures were classified as non-critical on the current release and were not considered critical for one to three releases in the future. In contrast, "our model is based on keeping the software updated with corrections in the current release" (see section B.1.1). For the same reason, we were unable to validate inequality (13), the maximum fault correction delay (see section B.2.1). This is a lesson learned from the research that we will further address in section B.4.

Lastly, given that there are only four, six, and four test times when fault corrections were made in Table B.1, B.2, and B.3, respectively, the samples were too small to allow validation of equation (10), the predicted amount of test time required to correct a given number of faults T_c. However, the problem of fault resolution postponement on OIJ shows up again on a relative basis as indicated by the MSE values for T_c of 1.05, 195.75, and 20.73 for OID, OIJ, and OIO, respectively, which are not shown in the tables.

B.4 Summary

It is imperative to include fault correction in reliability modeling and prediction because without it, predictions will understate the reliability of the software. It is important for the field of software reliability engineering to give more emphasis to this issue. Our research is an attempt to provide a framework for integrating failure prediction and fault correction. We have developed a number of equations for assessing the improvement in reliability resulting from fault correction. In addition, we have shown examples of how they apply to stopping rules for testing and prioritization of tests and test resources. We showed that the *number of remaining faults* is better than the *fault correction rate* for assessing reliability and prioritizing tests because the former can be used with a reliability threshold to predict how much testing would be required to meet the reliability goal.

We found that the most important factor in fault correction modeling is the delay between failure detection and fault correction. We modeled this factor, using the concept of a fault correction queuing service with exponentially distributed delay – a highly statistically significant empirical result based on Shuttle data. We assumed that fault correction would commence with failure detection and that the delay would be equal to fault correction time. This assumption is valid for organizations that choose to keep their software updated with corrections in the current release.

We obtained good validation results for two of the three OIs evaluated. The third OI served as a lesson learned that we will apply to future work on the NASA GSFC software projects. We will investigate whether fault corrections are postponed and, if so, whether we can model this delay. This will be a challenge because, whereas failure detection is a machine process, fault correction is part human process (deciding when to implement a correction and analyzing how to make the

correction) and part machine process (verifying the correction on a computer). The human component is difficult to generalize for inclusion in the model. It may be necessary to use an empirical distribution on each project or set of projects.

We were unable to validate *maximum fault correction delay* because, as stated, many fault corrections in the Shuttle are postponed to future releases. However, our theoretical finding that the maximum delay is independent of its distribution is significant. This delay is based on the assumption that corrections are made in the current release. In a future effort, we will attempt to collect data with various delay distributions to confirm or reject this finding.

B.5 References

[DAL94] Siddhartha R. Dalal and Allen A. McIntosh, "When to Stop Testing for Large Software Systems with Changing Code", IEEE Transactions on Software Engineering, Vol. 20, No. 4, April 1994, pp. 318-323.

[DAL88] S. R. Dalal and C. L. Mallows, “When Should One Stop Testing Software”, Journal of the American Statistical Association, Vol. 83, No. 403, 1988, pp. 872-879.

[EHR 93] Willa Ehrlich, Bala Prasanna, John Stampfel, and Jar Wu, "Determining the Cost of a Stop-Test Decision", IEEE Software, March 1993, pp. 33-42.

[FAR93] William H. Farr and Oliver D. Smith, Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) Users Guide, NAVSWC TR-84-373, Revision 3, Naval Surface Weapons Center, Revised September 1993.

[GOK96] Swapna S. Gokhale, Teebu Phillip, and Peter N. Marinos, “A Non-Homogeneous Markov Software Reliability Model with Imperfect Repair”, Proceedings of the International Performance and Dependability Symposium, Urbana-Champaign, IL, 1996, 10 pages.

[GOK97] Swapna S. Gokhale, Peter N. Marinos, Michael R. Lyu, and Kishor S. Trivedi, “Effect of Repair Policies on Software Reliability”, Proceedings of Computer Assurance, Gaithersburg, MD, 1997, 10 pages.

[HAU00] Chin-Yu Huang, Sy-Yen Kuo, Michael R. Lyu, and Jung-Hua Lo, “Quantitative Software Reliability Modeling from Testing to Operation”, Proceedings of the Eleventh International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, October 8-10, 2000, pp. 72-82.

[JES00] Daniel R. Jeski, “Estimating the Failure Rate of Evolving Software Systems”, Proceedings of the Eleventh International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, October 8-10, 2000, pp. 52-61.

[KEL97] Ted Keller and Norman F. Schneidewind, “Successful Application of Software Reliability Engineering for the NASA Shuttle”, Software Reliability Engineering Case Studies,

International Symposium on Software Reliability Engineering, November 3, Albuquerque, New Mexico, November 4, 1997, pp. 71-82.

[KLE75] Leonard Kleinrock, Queuing Systems, Volume 1: Theory, John Wiley & Sons, New York, 1975.

[LYU96] Michael R. Lyu (Editor-in-Chief), Handbook of Software Reliability Engineering, Computer Society Press, Los Alamitos, CA and McGraw-Hill, New York, NY, 1995, pp. 95-98.

[MUS87] John D. Musa, et al, Software Reliability: Measurement, Prediction, Application, McGraw-Hill, New York, 1987.

[SCH75] Norman F. Schneidewind, "Analysis of Error Processes in Computer Software", Proceedings of the International Conference on Reliable Software, IEEE Computer Society, 21-23 April 1975, pp. 337-346.

[SCH92] Norman 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.

[SCH 97] Norman F. Schneidewind, "Reliability Modeling for Safety Critical Software", IEEE Transactions on Reliability, Vol. 46, No.1, March 1997, pp.88-98.

[SIN91] Nozer D. Singpurwalla, "Determining an Optimal Time Interval for Testing and Debugging Software", IEEE Transactions on Software Engineering, Vol. 17, No. 4, April 1991, pp. 313-319.

[SMI99] Carol Smidts, “A Stochastic Model of Human Errors in Software Development: Impact of Repair Times”, Proceedings of the Tenth International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, November 1-4, 1999, pp 94-103.

[XIE92] Min Xie and M. Zhao, “The Schneidewind Software Reliability Model Revisited”, Proceedings of the Third International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, Research Triangle Park, NC, October 7-10, 1992, pp. 184-192.