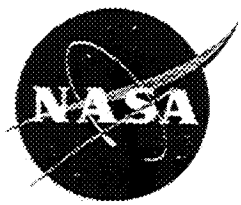
Technical Memorandum 104799

Software Analysis Handbook:
Software Complexity Analysis and Software Reliability Estimation and Prediction

Alice T. Lee
Todd Gunn
Tuan Pham
Ron Ricaldi

August 1994



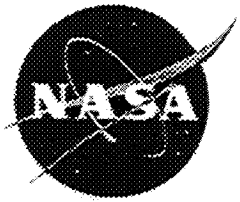
National Aeronautics and
Space Administration

Technical Memorandum 104799

Software Analysis Handbook:
Software Complexity Analysis and Software Reliability Estimation and Prediction

Alice T. Lee
Safety, Reliability, & Quality Assurance Office
Lyndon B. Johnson Space Center
Houston, Texas

Todd Gunn, Tuan Pham, and Ron Ricaldi
Loral Space Information Systems
Houston, Texas



National Aeronautics and
Space Administration

This publication is available from the NASA Center for AeroSpace Information, 800 Elkridge Landing Road, Linthicum Heights, MD 21090-2934, (301) 621-0390.

Summary

The purpose of this handbook is to document the software analysis process as it is performed by the Analysis and Risk Assessment Branch of the Safety, Reliability, and Quality Assurance Office at the Johnson Space Center. The handbook also provides a summary of the tools and methodologies used to perform software analysis. This handbook is comprised of two separate sections describing aspects of software complexity and software reliability estimation and prediction. The body of this document will delineate the background, theories, tools, and analysis procedures of these approaches.

Software complexity analysis can provide quantitative information on code to the designing, testing, and maintenance organizations throughout the software life cycle. Diverse information on code structure, critical components, risk areas, testing deficiencies, and opportunities for improvement can be obtained using software complexity analysis.

Software reliability estimation and prediction analysis is a quantitative means of assessing the readiness of software before it is put into operation. It provides a quantitative approach that will assist the software acceptance process. In addition, the results can aid in forecasting, managing, and scheduling testing resources.

This handbook is not intended to be the exclusive guide to software analysis. This document will be revised as new information on models and their respective processes are gathered.

Contents

Contents
(continued)

Contents
(continued)

Contents
(concluded)

Page

Figures

Tables

SECTION 1: SOFTWARE COMPLEXITY ANALYSIS AND THE DETERMINATION OF SOFTWARE TEST COVERAGE

1.1 Introduction

The purpose of this section of the manual is to provide a practical orientation to software complexity analysis. Software complexity analysis can provide meaningful information to the analyst, information which can be used to determine

- insight into the software structure
- identification of critical software components
- assessments of relative risk areas within a software system
- identification of testing deficiencies
- recommendations for program improvement

Software complexity analysis can make a proactive contribution to improving the quality and reliability of a software system during all phases of the software life cycle, including the preliminary design, detailed design, coding, test, and maintenance phases. Although the impact of a complexity analysis is greater during the latter stages of the software life cycle, its contribution potential is maximized if it is begun early in the life cycle. The methodology presented in this report is applicable once code is available. The concepts in this report can be applied to the early design phases.

Concepts and theories are presented for three distinct types of complexity analyses, as well as the methodology for performing each type of analysis. Some potential applications of these analysis results are also presented. This section will also describe the features of some of the tools used for performing these analyses, as well as various programs written to automate the process as much as possible. It is the intent of this section to provide the reader with a rudimentary knowledge of the theories and concepts associated with software complexity, which will allow the reader to develop some proficiency in performing this type of analysis. It is not the intent of this section to provide an in-depth theoretical discussion on the derivations and concepts associated with software complexity, rather it is to provide a guide for the practitioner. After introducing some of the theories and concepts associated with software complexity, this section will then provide a specific hands-on approach to the methodology associated with performing a software complexity analysis, including detailed procedures and the applications of various commercial software packages which have been developed to assist in performing complexity analysis.

Two different software projects were evaluated before writing this section of the handbook. The projects were dissimilar, written in different languages, and run in different execution environments. The analysis of each project had some unique characteristics which contrasted each other, and the methods for dealing with these types of differences will be addressed in this section. The experiences and difficulties encountered with these pilot programs should provide a benefit to those who will perform software complexity analysis in the future.

This section introduces three aspects of software complexity analysis: static analysis of the software system, a dynamic analysis of the software system, and a component-level analysis of the functional criticality of a software system. Each are addressed in subsequent sections below.

1.1.1 Static Analysis

The first aspect of complexity analysis presented is the evaluation of software complexity by collecting and analyzing complexity metrics. This aspect of complexity analysis addresses questions concerning the complexity of a software system and its impact on errors discovered during development and use. Complexity metrics, when applied correctly, can provide a prediction of errors based on the metric parameters; a method of monitoring the modularity and maintainability of software; a method of measuring the number and nature of a component's interfaces; a method of measuring the ability of a specific component to be tested; and can provide the ease with which a component can be reused or re-integrated in a new environment. In summary, a static complexity analysis can provide valuable data to support software quality evaluations and reliability predictions.

Complexity metrics can be defined as measurements relating to a software's structure, its size, and its interfaces. These metrics range from relatively simple measures such as the number of lines of code, to much more complicated metrics that measure very abstract characteristics of software. Another category of metric which can be collected relate to the development aspects of the software program, such as number of requirements, development time, and number of errors in requirements. A more detailed definition of some of these complexity metrics will be presented in section 1.2.1.1.

It should be emphasized that a static analysis of software deals only with the structure and development of the software source code, and does not require running a piece of software to begin to implement a metrics program. The applicability of different metrics at various stages of the software life cycle will be discussed in section 1.2.1.3.

It is important to note that the interpretation and the evaluation of complexity metrics is still quite subjective at this time. The selection of the metric set and the interpretation of the data are all up to the judgment and expertise of the user. An example of an analytical technique available for the evaluation of the metric set is presented in section 1.2.1.4.

1.1.2 Dynamic Analysis

Another aspect of software complexity analysis is dynamic analysis. A dynamic analysis measures the efficiency and effectiveness of software testing by monitoring the software system during its execution phase. Data collected during the execution is used to evaluate the thoroughness of the testing, to determine if there is adequate utilization of the testing resources, and to prioritize the test case distributions. The efficiency of software testing is gauged by measuring the amount of actual code executed during a software test run. There are several methods of measuring the efficiency of testing, each with a varying degree in the level of detail and level of effort. These methods are described in more detail in section 1.2.2.1.

To perform a dynamic analysis, it is necessary to have an operational version of a software system. This restricts the ability to perform a dynamic analysis until later in the life cycle, at the unit test stage and beyond. A dynamic analysis is performed by actually making modifications to the code before beginning a test or operation. The code is modified at selected points by inserting "traps" in the code. This "instrumented" version of the source code is then re-

compiled, and the test cases are performed on the instrumented version of the code. At the conclusion of the test run, data may be extracted as to the frequency of execution and the identification of unexecuted blocks of code.

1.1.3 Software Functional Criticality

Software functional criticality addresses the failure modes and effects of a specific software component in relation to the system operation. The first step in assessing the functional criticality of a component is to establish the failure effect criteria, including factors such as risk to human life, risk to national resources (vehicle, etc.), loss of the software functional capability, or reduction in software capability. The next step is to approximate the failure modes of each component, and attempt to determine the overall effect of that individual component on the entire software system.

1.2 Software Complexity: Theories and Concepts

1.2.1 Introduction to Complexity Metrics

Software complexity metrics can be defined as measurements used to characterize and quantify various properties of a software system. When they are applied correctly, software complexity measurements can be used to make decisions which can reduce the development cost, increase the reliability, and improve the overall quality of the software system.

The science of software complexity was introduced in the mid 1970s when Halstead proposed measurements designed to determine various software quality characteristics. Since that time, researchers have proposed literally hundreds of measurements quantifying various aspects of software, from relatively simple and comprehensible measurements (such as lines of code), to very abstract measurements (such as entropy or hierarchical complexity). There is much debate over the relative value of one metric over another, and there is no standard set of metrics which has widespread use in industrial or research applications. It is up to the user to make subjective decisions as to what metrics to collect, what metrics are most applicable and useful at specific points in the software life cycle, and the proper and appropriate use of these metrics.

Software complexity metrics can be divided into two general categories: textual metrics and architecture metrics. Textual metrics are measurements taken on an individual software component. Textual metrics stem directly from the original Halstead metrics, measuring attributes and functions which can be determined by looking at the module as a stand-alone component. Some examples of textual metrics are the lines of code, the number of nests, and the number of operators.

Architecture metrics are based on the design of a software system, and how individual components are connected together. Architecture metrics often measure such properties as the modularity of a software system, the identification and use of shared (global) data structures, the communication flow between individual components, and data coupling.

In addition to complexity metrics, other metrics can be collected to determine characteristics of the software being analyzed. These metrics usually measure programmatic factors such as the number of requirements, the number of pages in a requirements document, and the number of changes in requirements. This analysis limits its scope to complexity metrics, and does not utilize the programmatic metrics in the formulation of the results. The analysis example described in section 1.2.1.4 can be easily adapted, however, to include such programmatic metrics.

1.2.1.1 Common Metric Definitions

This section will define some of the metrics which can be used in a complexity analysis. The metrics defined here are not all the metrics used in current studies, but does provide a representative list of some metrics which one might consider to use. The fact that different metrics were collected for these different projects illustrates one shortfall of complexity metric analysis: there is a definite need for a standardized set of complexity metrics which can be used from project to project and language to language.

1.2.1.1.1 Lines of Code

The most familiar software measure is the count of the lines of code. The term lines of code is often abbreviated LOC, or for large programs KLOC (1000 lines of code). There is no consensus to the exact definition of what constitutes a line of code, although a standard is being developed. One common definition is :

> A line of code is counted as the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file. This specifically includes all lines containing program headers, declarations, executable and non-executable statements.

This definition of the lines of code includes non-executable statements in the count of lines of code. This includes comments and header information, which some prefer to eliminate in their count. Care should be taken to ensure that one definition is used consistently throughout an analysis of a program. This is especially true if using different analyzers, which might calculate lines of code in different fashions.

1.2.1.1.2 Halstead's Textual Complexity Metrics

Halstead proposed several measures from which he was able to empirically derive characteristics. Four base measurements were taken from the source code, and the rest of the quantities were derived from the four base measurements.

The four base measurements are

n1	number of distinct operators
n2	number of distinct operands
N1	total number of operators
N2	total number of operands

From these four measurements, Halstead derived the following quantities:

Program Length	N = N1 + N2 This is Halstead's definition of the length of a program.
Program Volume	V = (N1+N2) ln(n1+n2) This is an indication of the number of bits needed to code a program
Program Size	S = (n1) ln(n1) + (n2) ln(n2) This is Halstead's indication of "error aptitude" and program readability.
Program Difficulty	D = [(n1)/2] (N2/n2) This is an indication of the difficulty in developing and understanding a program component.
Mental Effort	E = [(n1) (N2) (N1+N2) ln(n1+n2)] / 2(n2) This is an indication of the effort required to understand and develop a program.
Estimated Number of Errors	B= [E ** (2/3)] / 3000 This is an estimate of the amount of errors resident in a program module.

Logiscope as well as most code analysis tools will calculate the majority of Halstead's metrics, both derived and the base metrics. (For a complete list of the Logiscope standard metric set and the methods of defining new metrics, refer to section 1.4.1.) All the tools researched to date provide at least the four base metrics, from which the derived metrics can easily be calculated.

1.2.1.1.3 McCabe's Cyclomatic Number

McCabe's cyclomatic number is one of the more popular complexity metrics. Derived from principles adapted from graph theory, the cyclomatic number gives the minimum number of independent number of paths through the logic flow of a program. There are actually two versions of McCabe's cyclomatic number, so care should be used when comparing results.

The first version of the cyclomatic number is defined as

$$V(G) = |E| - |N| + 2p \qquad \text{(Eq 1)}$$

where V(G) = McCabe's cyclomatic number
E = number of edges in the control graph
N = number of nodes in the control graph
p = number of connected components or subprograms (for calculating V(G) for single components or modules, p = 1)

There is one problem with this version of the cyclomatic number: the numbers are not additive when stringing together program components. In other words, the whole is not equal to the sum of its parts. For example, consider two software components, A and B, each with a cyclomatic number equal to 6. If A and B were tied together into one larger program (where the exit of A led straight into the beginning of B), one would expect the cyclomatic number of this new component to be 12. That is not the case, as the new component would actually have a cyclomatic number 10. This can be a problem, especially when considering groups of programs or evaluating an entire program.

To correct this problem, McCabe proposed a modification to the definition of the cyclomatic number, which is often referred to as the essential cyclomatic number. The essential cyclomatic number is calculated by

$$V(G) = |E| - |N| + p \qquad \text{(Eq 2)}$$

The essential cyclomatic number is additive.

It is not always clear which version of the cyclomatic number is being calculated, as the term "cyclomatic complexity" is often used to refer to either of the two versions, which may be represented by the same symbol V(G). One should try to use the essential cyclomatic number when possible due to the usefulness of its additive properties. Logiscope, for example, calculates the first version of the cyclomatic number. This can be corrected by programming a modification equation into the reference file. The details of the procedures are found in sections 1.3.1, 1.3.3, and 1.4.4

1.2.1.1.4 The Complexity Measure of Henry and Kafura

Another metric was proposed by Henry and Kafura:

length * (fan-in * fan-out)**2

where
length is the number of lines of code in a program
fan-in is the number of data objects passed into a called procedure plus the number of global data structures from which the procedure retrieves its information
fan-out is the number of data objects received from a called procedure plus the number of global data structures which the procedure updates

1.2.1.2 Utilization of Complexity Metrics

Complexity metrics are collected to measure various properties and aspects of a software program, either at a component level or a system level. Complexity metrics can be used to make quality evaluations of software, to monitor the development growth of software systems, and to estimate and predict the reliability of software. Complex software often leads to high development costs, high maintenance costs, and decreased system reliability. By monitoring the complexity of a software program during design and development, changes can be made to reduce complexity and ensure the system remains modifiable and modular.

1.2.1.3 Applicability of Metrics to the Software Life Cycle

Some metrics might be reflective of the software system (relative to error prediction) during the early stages of the program development, while other metrics might characterize the software system better during the later stages of development or during the maintenance phase of the program. One example might be that during the early design phase (Preliminary Design Review or prior), a simple metric such as the number of requirements might be used to estimate the error count in a software component. After coding and some testing, it might be found that the specific metric might be replaced by metrics more representative of the software structure to give more accurate results. It is up to the analyst to select the appropriate metric set which will be most representative of the software system at a specific moment in the development and usage phase.

1.2.1.4 A Practical Example: Construction of an Error Prediction Model

This example employed complexity metrics to predict the number of latent errors residing in a software program. This was accomplished by adapting the methodology proposed by Khoshgoftaar and Munson [1]. The concept behind this procedure is to develop a regression model which uses complexity metrics to predict the number of errors in a software component.

A similar method for predicting the failure rate of software was proposed by the Rome Air Development Center (RADC)[2]. However, the error prediction model presented here differs from that presented by the RADC mainly in the selection of the model parameters. The RADC model was developed to be applicable throughout the entire software life cycle, so the model emphasizes requirements-oriented metrics such as number of requirements, number of pages in the requirements documents, and requirements development time. The error prediction model presented here does not include requirements-oriented metrics as it is strictly based on the code complexity and structural characteristics. This allows one to investigate the detailed interactions between the structural characteristics of a software program and its failure rate.

The first step in developing this regression model was to collect metric values and error information from previous projects. The following metrics were collected:

-Number of Statements
-Total Number of Operands
-Total Number of Operators
-Henry and Kafura's Complexity Metric
-Comment Frequency
-Larsen HC Metric
-Essential Cyclomatic Complexity
-Number of Inputs
-Number of Outputs
-Number of Direct Calls
-Nesting Depth
-Program Length (Halstead)

With the exception of the complexity metric by Henry and Kafura, each of these metrics has the properties of ratio-scaled measures as defined by Zuse [3]. The selection of ratio-scaled parameters is important, as it allows for such statistical operations as arithmetic mean and the calculation of percentages. It should be noted that the metric set was selected subjectively, and research is continuing to define a minimum standardized complexity metric set. This minimum set would ideally be the minimum number of metrics required to fully characterize the software. For samples of actual metric data, refer to tables 4 and 5 in section 1.4.1.2.4.

A regression analysis was performed using the complexity metric values and associated errors for each component. To handle the multicollinearity between the metric values, we used the method of principle components. The method of principle components reduces the effects of this collinearity by taking only a limited subset of the principle components. For a more complete description of the method of principle components, see Montgomery and Peck [4].

After the regression model is defined, the metrics can be grouped into categories according to their relative effect in areas such as complexity, interface criticality, testability, and maintainability. This method was proposed by McCabe [5].

We developed the error prediction model based on error data collected from two dissimilar projects. Metrics were collected using a commercial code analyzer program and the regression analysis was performed using a PC-based statistical analysis package. The R2 coefficient of determination for the model using the method of principle components was approximately 0.8. We believe that the accuracy of this model will be increased once more data has been collected over a larger portion of the software life cycle.

Four arbitrary levels were selected on the basis of the corresponding error prediction model output, each level corresponding to a higher predicted error count. These levels were referred to as error susceptibility levels, and numbered from 1 to 4, with 1 corresponding to the level having the highest amount of predicted errors. The numbering of the levels was selected in this fashion to be consistent with the failure effects analysis which will be described later. This zoning of the components into levels reduces the granularity of the model; however, this reduction in precision is acceptable until the model can be solidly established with a sufficient amount of data.

The model was demonstrated by considering the number of error reports encountered during testing of a large-scale test support software system developed for NASA. Metrics were collected for this system and inputted into the error prediction model. Components were assigned a level of error susceptibility based on the predicted number of errors. The number of error reports per component was found by dividing the total number of error reports generated for all the components in that susceptibility level by the number of components in that level. The result of this study is shown in table 1. Note that 6.5% of the components (identified as level 1 and 2) contributed to 44% of the total failures.

Table 1 - Demonstration of the Error Prediction Model

Error Susceptibility Level	Number of Components	Number of Error Reports	Number of Error Reports per Component
1	2.93%	30%	2.94
2	3.60%	14%	1.16
3	14.75%	37%	0.76
4	78.72%	19%	0.08

1.2.2 Dynamic Analysis: Definitions and Concepts

Dynamic analysis focuses on the execution of software such as the portions of the actual code which get executed, the frequency of their execution, and the amount of time spent in each application. The frequency of execution and timing of a program are often referred to as profiling.

A typical application of a dynamic analysis is in the determination of test case coverage. A dynamic analysis will track the execution path through the software code, and determine what logic paths are actually executed.

1.2.2.1 Test Coverage Monitoring Techniques

To evaluate the test case coverage, some method of monitoring the execution status of the software component during the execution of the code must be used. Four types of monitoring methods are typically used, listed below in increasing order of complexity and expense as to the amount of execution time required. The method selected for use will depend on the tool used for monitoring as well as the criticality of the component being monitored. Each method requires a technique usually referred to as instrumentation. Instrumenting a software program usually means adding additional code which records the execution status of various parts of code. These status flags can then be post-processed to allow one to trace the execution sequence through the source code itself. The degree to which a software program is instrumented is determined by the monitoring technique which is to be used.

1.2.2.1.1 Entry Point Monitoring

Entry point monitoring tests only to see that a given component is used by a test case but says nothing about which parts of the component were executed. This is the least complicated of the monitoring techniques, and requires the least amount of overhead in terms of computing resources.

1.2.2.1.2 Segment (Instruction Block) Monitoring

Segment monitoring tests to see that all the statements in the code have been executed but does not relate any information as the logic paths.

1.2.2.1.3 Transfer (Decision-to-Decision Path) Monitoring

Transfer monitoring measures the logical branches from one segment to another. Executing all the transfers equates to the execution of all the statements as well as all the logic paths in the code.

1.2.2.1.4 Path Monitoring

Path monitoring attempts to monitor the execution of all the possible paths through the code. Path monitoring is impractical except on very small routines due to the large number of possible paths that result in code with several control statements.

1.2.2.2 Profiling

There are also monitoring techniques for measuring the frequency and execution times of individual components. This type of monitoring can be very useful in determining the actual execution times of individual components, and such information can be used to estimate an operational profile. Profiling information can be extremely useful, especially in future research and work. Most compilers have profiling options, which can provide useful information. Profiling, in this sense, can be referred to as monitoring a program's execution and outputting a detailed procedure-by-procedure analysis of the execution time, including

- how many times a procedure was called
- what procedure called it
- how much time was spent in the procedure
- what other procedures did it call during its execution

Profiling data is typically collected using a combination of compiler directives and UNIX utilities.

1.2.2.3 A Practical Example: Transfer Monitoring During Interface Test Execution

This example evaluated test coverage and used these results to determine the efficiency of testing and the approximate usage over the operational profile. To evaluate the test case coverage, one of the methods described in section 1.2.2.1 must be used to monitor the execution status of the software component. The transfer monitoring technique (sometimes referred to as decision-to-decision path monitoring) was selected for use in this example. The decision-to-decision path monitoring technique requires instrumenting each decision node in the program, recompiling the program, and monitoring the execution status of each node. The data recorded was whether or not a specific path was executed during the test run, and the execution frequency of that path.

A histogram summarizing the distribution of test case coverage for a system containing approximately 140 components is presented in figure 1. The bars represent the number of components whose test coverage fell within the interval represented on the X-axis. The single line represents the cumulative percentage of components whose coverage falls within a specified interval or below. The left vertical axis is the scale for the frequency of components within a given interval, and the cumulative percentage scale is on the right vertical axis. For example, the leftmost bar in figure 1 indicates that approximately 28 components had less than 10% of

the code tested. The cumulative percentage, found by referring to the right vertical axis, indicates that this represents approximately 20% of the total number of components.

The test coverage distribution after the completion of testing operations is shown in figure 1. Note that approximately 20% of the components in this program were not executed at all (0% test coverage) during the testing. The initial response to this situation is that the program was not completely tested. However, after further investigation, it was found that the majority of this untested code represented functions which were no longer used in this release of the software, or represented redundant code. One recommendation made from this study was to eliminate redundant and unused code from the operational version of the software.

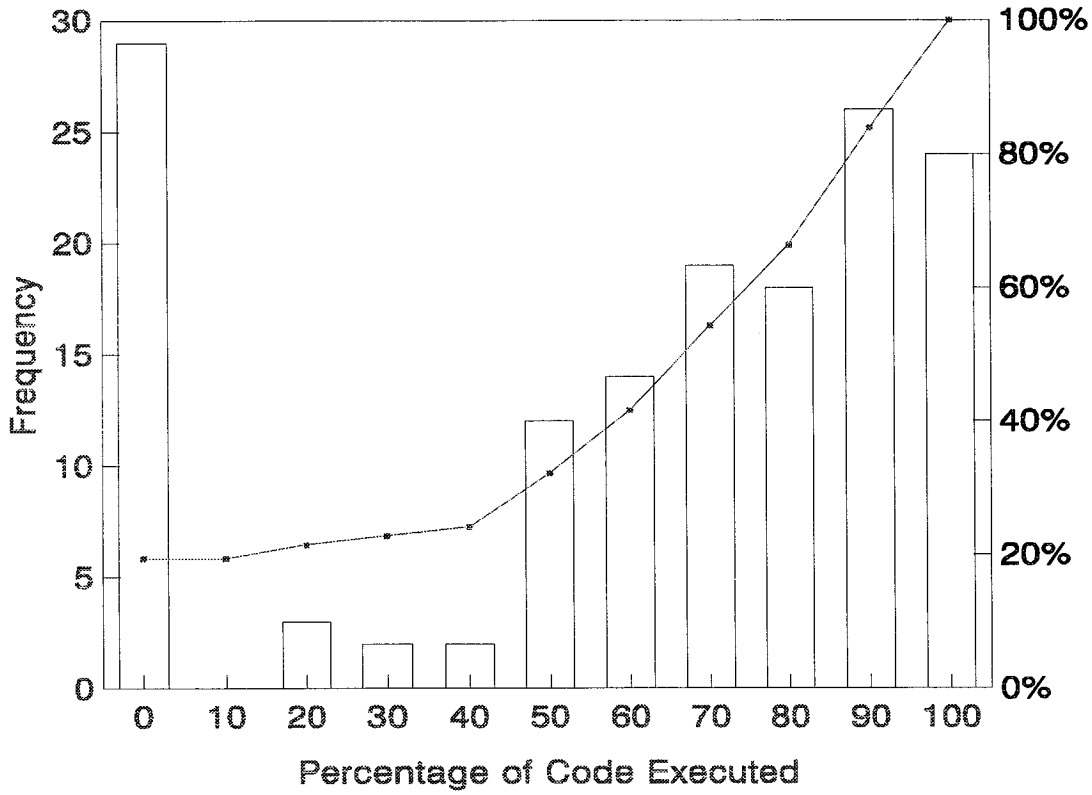


Figure 1 - Distribution of test case coverage.

The results of this study were used to determine the efficiency of testing in relation to the amount of source code covered, and to provide an estimate of the relative amount of code executed per component for normal program operations. When combined with the results from the error prediction model, this result can be used to approximate the number of latent errors remaining in a particular component. For example, suppose a component was predicted to have 10 errors (from the output of the prediction model), and under a nominal operational profile 40% of the entire code was expected to be executed. Assuming the faults are randomly distributed throughout the entire code, six latent faults would be expected to reside in the component at delivery, to be found during future operations of the software.

1.2.3 Software Functional Criticality

The first step in assessing the functional criticality of a component is to establish the failure effect criteria, including factors such as risk to human life, risk to national resources (vehicle, etc.), loss of the software functional capability, or reduction in software capability. The next step is to approximate the failure modes of each component, and attempt to determine the overall effect of that individual component on the entire software system. The propagation of these failure modes across the interfaces can then be considered. The weighting of each individual failure mode by its associated probability and failure criticality was considered but was beyond the current scope of the work.

A Practical Example: Functional Criticality for a Specific System

This example discusses the determination of the functional criticality levels for components which made up a safety-critical ground software system. The failure criteria were established by creating four specific groups, similar to hardware FMEA classifications. Criticality one designates the most critical components, while criticality four components are the least critical. The criteria for assigning a component to a specific criticality group were based on the software system under study being a safety critical item. For other particular systems, the criteria may include other factors such as cost, schedule, and fault tolerance. In this case the following criteria were established:

1) Failure of criticality 1 components would cause a complete loss of program function, possibly resulting in loss of vehicle and human life.

2) Loss of criticality 2 components would cause a recoverable loss of function or severely degrade the overall system performance.

3) Loss of criticality 3 components would affect non-critical functions, such as some plotting and printing functions, but primary program objectives would remain unaffected.

4) Loss of criticality 4 components would result in a recoverable loss or intermittent failure of a non-essential function, but the primary program objectives would remain unaffected.

Cost was not considered as a factor in the definition of the criteria for each level. Further research is needed to evaluate the relation of cost and development time to the criticality level.

Each individual component was evaluated according to the above criteria. The results of this evaluation were then applied to the risk index derivation and described in section 1.2.4.1.

1.2.4 Results Consolidation

The results from these analyses described above—a static analysis, a dynamic analysis, and a functional criticality analysis—can be consolidated to form a single result which can aid in the decision making process.

A Practical Example: Derivation of the Risk Index

The concept of a risk index for software has been developed to provide a quantitative risk assessment of software systems by integrating the results of the complexity analyses. This risk index is a factor that takes into account the likelihood of a failure as well as the consequence of that failure. The underlying concept of associated risk has been adapted from safety hazard analyses performed on hardware systems. Risk is calculated as the product of the severity of a failure occurrence multiplied by the probability of such a failure occurrence. The risk index provides a quantified measure for comparing cases such as a failure mode of a component that is highly unlikely but catastrophic, versus a failure mode with less severe consequences but with a greater chance of occurrence. Previously, engineering judgment has been used to make these comparisons.

The risk index factor is a quantitative measure of risk defined to be the product of the probability of a failure occurrence with the severity of that occurrence. Results from the error prediction model, the test coverage, and the failure effects analysis are used to calculate this factor. An overview of the derivation of the risk index is shown below in figure 2.

The test coverage results can give profiling information for each component if the test cases considered approximate the actual usage expected for the software. Ideally, a well-defined operational profile will be available early in the software life cycle. However, dynamic coverage results can be used in the absence of an adequately defined operational profile. For a description of the development of the operational profile, see Musa [6]. The coverage results provide a measure of the amount of untested code. Multiplying this measure by the results from the error prediction model gives an estimate of the number of latent errors remaining in the component. This assumes that if a given statement or segment has been executed successfully during test, then the number of errors remaining in that statement or segment can be considered to be insignificant.

The test coverage results can also provide the profiling information necessary to determine the approximate probability of execution of a particular component. If the software is not yet mature enough for a dynamic test case analysis, the profiling information and associated probabilities can be approximated based on the expected operational usage.

We wanted the risk index to be normalized to be in the range [0,1], with higher values indicating greater risk. Because the functional criticality levels were chosen to be analogous to existing hardware criticality definitions, a linear transformation was required before computing the risk index. The transformation is:

$$FC' = (1.25 - 0.25 \times FC) \qquad \text{(Eq 3)}$$

in which FC' is the transformed criticality index and FC is the original value.

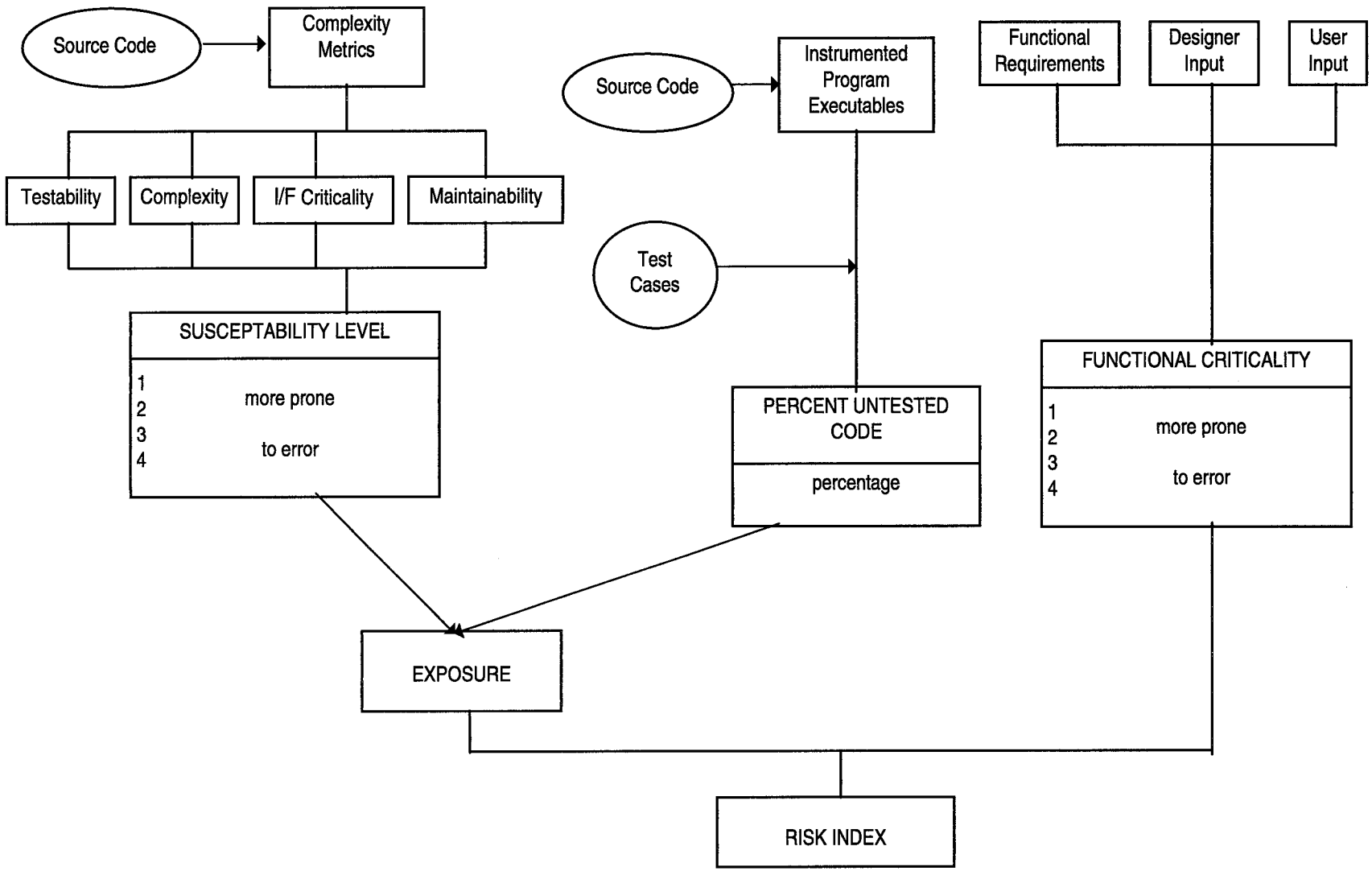


Figure 2 - Derivation of the risk index.

A similar transformation was required for the error susceptibility measure, resulting in an equation for the risk index as:

$$RI = (1.25-0.25xFC)(1.25-0.25xES)(1-TC) \quad \text{(Eq 4)}$$

where RI = risk index
FC = functional criticality
ES = error susceptibility
TC = code coverage

An example of the results for some sample code is given in table 2.

Table 2 - Summary Parameters and the Risk Index

COMPONENT	TOTAL TEST COVERAGE PERCENTAGE	ERROR SUSCEPTIBILITY	FUNCTIONAL CRITICALITY	RISK INDEX
adtgud	85.29	1	1	0.14705882
ardinp	93.75	2	3	0.0234375
ardlin	62.50	2	3	0.140625
b94med	95.24	4	1	0.0119
chkbnd	56.72	3	1	0.2164
convrt	97.14	1	2	0.02145
d2head	81.25	3	1	0.09375
engdta	93.65	4	1	0.015875
f02med	91.30	2	1	0.06525
fstild	97.67	2	1	0.017475
initmd	100.00	2	1	0
k96med	89.47	3	1	0.05265
k85med	90.91	1	1	0.0909
main	59.09	4	1	0.102275
medout	50.00	2	1	0.375
opnfil	72.06	2	3	0.104775
poly	85.71	3	2	0.0535875
rddasc	57.14	4	4	0.0267875
rdhasc	60.00	4	4	0.025
svdlod	83.05	1	3	0.08475
thrott	100.00	2	1	0

The concept of associated risk can be applied to software development and testing improvements. A component with high associated risk should be allocated more development and testing resources than low risk components. Therefore, more test effort and review emphasis should be placed of components with a high risk index factor.

The risk index derivation as summarized in this handbook assumes the availability of source code. One topic of further needed research is the expansion of the risk index concept, making it applicable during earlier phases of the life cycle before source code is available. To evaluate the risk without actual source code, one must first modify the parameters used in the error prediction model. A new model should be constructed using available programmatic metrics, such as the number of requirements for a given component, the number of interfaces for a given component, and the number of errors or changes against a specific component. For more information on the use of programmatic metrics in the construction of an error prediction model, including some metrics and their model parameters, see Friedman [2]. If such metrics are not available, or if it is judged that these metrics will not give adequate results, the predicted number of errors can be based on past data from similar software components. Another method would be to predict the number of errors using programmatic metrics, and then update this prediction using Bayes' Theorem with the historical data. The test coverage can be predicted by a subjective evaluation of the preliminary test plans. The functional criticality, which is not dependent on source code, is evaluated the same as described above. This illustrates an important point: Since the functional criticality evaluation is independent of the life cycle, an evaluation of a component's failure modes and effects should be begun as early as possible in the life cycle. The results of these analyses can then be combined into a factor relating to overall risk.

1.3 Tools

1.3.1 Logiscope

Logiscope is a commercial software package developed to perform static and dynamic complexity analysis of software code. The version used for this study is owned by the JSC Software Technology Branch and was made available by special permission. Logiscope runs on a Sun Sparq workstation and can be configured to analyze several different languages. Presently Logiscope is configured to analyze ADA, C, and five different language versions of FORTRAN.

Logiscope has the capability to calculate over forty standard "built-in" complexity metrics. The user is allowed to define the functions for new custom metrics as well. In addition, Logiscope has the capability to perform three different types of dynamic monitoring. This combination makes Logiscope a very powerful tool for software quality assessment, software complexity analysis, and software metric collection.

Logiscope performs its functions in a three step process: the analysis of the raw code, archiving the analysis results, and editing the archived results. The analysis process looks at the raw code and generates a results file (unintelligible on its own). The archive sequence allows the user to group various components and analyze them together. The edit sequence allows the user to view the results and manipulate them into a more meaningful form.

1.3.2 Compiler Tools

A variety of utilities and tools are also available and extremely helpful in the execution of a software complexity analysis. One of these includes a set of UNIX procedures and compiler directives useful for the determination of the usage profile and test case coverage. These tools and directives are discussed here in section 3, but it should be noted that these tools and procedures are concerned with dynamic operations which require a compiled and executable version of the code.

1.3.3 Other Tools

A wide variety of commercial tools have been developed to perform source code analysis and software complexity analysis, including the McCabe Tool and the ASAP Tool. These are just two of the many programs available today, and it is up to the user to evaluate and select the tool most appropriate for his applications.

Thomas McCabe, of McCabe and Associates, is well known in the software complexity area as the developer of the cyclomatic complexity metric. The McCabe Tool, developed by Thomas McCabe, provides capabilities very similar to Logiscope. The McCabe Tool has a superior graphical user interface, and superior graphical representations on the software hierarchy; while Logiscope is superior in metric calculation and flexibility in selecting and defining the metric set.

Another tool is the ASAP tool available through COSMIC. This tool has been used successfully by the JSC Engineering directorate to collect a limited set of metrics. The ASAP tool can collect some metrics, but the metric set is limited. Converting these metrics to another format proved time-consuming and inefficient. Despite its disadvantages, ASAP can provide a low-cost alternative to the expensive code analyzers available today.

1.4 Detailed Analysis Procedures

1.4.1 Collection of Metrics

1.4.1.1 Source Code

The first step of any static analysis is to collect the source code. The source code should be in an ASCII text file format, and can be loaded from tape or by using ftp, if available. Tape loading and ftp procedures are discussed in more detail in the UNIX procedures section. It is often more efficient to check with the UNIX systems administrator before loading any large amount of files on a local machine.

When transferring a lot of files, it is often easier to create a compressed tar file. This file is a single compressed file which contains all the files to transfer. The file is also compressed, thus saving transfer time, and, being a single file, is easier to handle. After the tar file is transferred, it can be extracted, recreating the directory structure (including multiple subdirectories) from the original.

To create a tar file, first create a file which is a concatenation of all the files or directories by issuing the following UNIX command:

tar cvf <target filename> <source files or directories>

where the tar flags are c - create
v - verbose

This creates a single file which can then be compressed by issuing:

compress <filename>

The **compress** command creates a compressed version of the file which has the same name, but with a .Z extension. After transferring, the files can be uncompressed and extracted by issuing the following sequence of commands:

uncompress <compressed filename>

tar xvf <filename>

This will recreate the directory structure and extract the files on the new machine.

To run Logiscope, it is necessary to have all the files resident on the machine licensed to run Logiscope, as well as an account on that machine. At this time that machine is galileo (ftp address 134.118.101.24), which is owned by the JSC Information Systems Directorate. The creation of the tar file can be useful when transferring file from the local machine to galileo, and vice versa.

1.4.1.2 Logiscope Procedures

1.4.1.2.1 Static Analysis of Code

Since Logiscope operates in an analyze - archive - edit sequence, the first step in performing a static analysis is to run the appropriate Logiscope code analyzer with the raw source code. Logiscope has a variety of analyzers corresponding to the programming language, such as ADA, many versions of FORTRAN, and several different versions of C. A listing of the executables associated with the analyzer for each of the languages is given in table 3.

After initiating the appropriate analyzer, Logiscope will query the user. The queries will vary slightly depending on the language of the program being analyzed:

```
Source_File name    ?
Basic counts (1=screen, 2=file, 3=not)    ?
Instrumentation  (y/n)    ?
Ignore Statement  (y/n)    ?
```

Table 3 - Logiscope Analyzer Executables

LANGUAGE	EXECUTABLE
FORTRAN	log_fort_std
FORTRAN 77 PLUS	log_fort_plus
IBM FORTRAN	log_fort_ibm
HP FORTRAN	log_fort_hp
FORTRAN 32	log_fort_f32
VAX FORTRAN	log_fort_vax
C	log_c_sun
Microsoft C	log_c_msoft
VAX C	log_c_vax
ADA	log_ada

The appropriate answers to the last three questions are 3, n, n. When performing ADA analysis, Logiscope will also prompt for the ADA library directory. A simple script file can also be written to answer all the questions. The following script file (a copy of which resides on Bilbo in the users/tgunn/utility subdirectory) can be used to perform a static analysis of all the ADA files in the current subdirectory:

```
set *.ada
while test $1
do log_ada <<!
$1
3
n
/tmp/gal_home/alee/cmslib/testdir/ada_lib
!
echo $1
shift
done
```

This script file can be easily revised to accommodate whatever parser you are using by revising the file search string in line 1 and the analyzer executable (see table 3) in line 3.

The Logiscope analyzers are particular to the order of code analysis as well as to the structure of the code itself. If there are any compiler dependencies associated with the program, Logiscope requires that the program be analyzed in the same order as it is compiled. If there is a make file (or its equivalent) associated with the program which is to be analyzed, it is necessary to obtain that file to analyze the code properly.

A specific example of this peculiarity occurred during the analysis of an ADA program. The analysis of the source code was failing since the Logiscope code analyzer could not determine that all the parent files had been analyzed for a specific file. We determined that many of the program components called non-standard utility routines, or Bootch utilities, were not part of the Logiscope standard ADA library files. When a program component called one of these utilities, Logiscope would detect that this utility had not been analyzed and was not part of the library routines, so it would abort the process. A workaround was developed which created dummy Bootch utility files; i.e., a procedure which used valid ADA structures but did not accomplish anything. These empty files had the following structure:

```
procedure <Bootch Utility Name> ;

end procedure;
```

These empty files were analyzed, and then all the daughter files could be analyzed as well. This type of workaround could be used in other situations, but keep in mind that this type of "quick fix" should only be used as a last resort. This type of fix will distort the end results, the extent of which is dependent on the number of files being dummied and the number of components which are daughters to the dummied component.

Another problem we encountered was the lack of the compilation dependencies for an ADA program. Components could not be analyzed until their parents were analyzed. We decided to attempt to analyze this program in an iterative fashion, attempting to analyze the entire program but only succeeding in obtaining valid results for the first level of the software hierarchy. The analyzer can then be re-run on the entire program, this time picking up the next level down as well. This cycle will be repeated until all levels of the program are able to be analyzed. Although there is no reduction in the accuracy of the final result, this method can be very time-consuming, both in machine time and in user input time. Obtaining the compiler directives is a much more preferred and time-efficient process.

Another problem we discovered with the Logiscope analyzer deals with incompatibilities with the parser and the source code. This problem occurred primarily with the FORTRAN parsers, and some could be rectified by trying another parser on the problem file. If that doesn't work, Logiscope has the ability to exclude certain lines from the file. This can be done by entering a "y" when Logiscope prompts "Ignore Statement (y/n) ?" and then entering the statement.

When implementing workarounds of any sort, keep in mind that the metric output obtained might not be 100% accurate. If accuracy is going to be sacrificed, one should make an estimate of the extent of the distortion of the data, compared to the amount of extra work required for the desired accuracy.

After attempting to analyze a file, Logiscope creates a results file with the identical name as the source file with the extension .res. These .res files contain the information Logiscope will use in its archive and edit sequences. Troubleshooting the analysis can be accomplished by looking at the size of the .res file. If the file size of the .res file is 85, this is a good indication that the analysis has failed for this particular file. The Logiscope code analysis can fail for a number of reasons, such as the parent module has not been previously analyzed or there is an unrecognized syntax in the program file. For more information on Logiscope analyzer failure codes, refer to the Logiscope manuals.

1.4.1.2.2 Archiving Results Files

Archiving the .res files is probably the simplest of the Logiscope analyze - archive - edit sequence. Archiving groups the individual analysis results files (.res files) into a single .arc file representing the entire program. The capability also exists to update individual program files as they are modified into an existing archive file. To archive a group of static analysis results (.res) files, simply issue the following command:

```
log_arcsta
```

Logiscope will prompt you for the name of the archive file and the name(s) of the analysis results files. Standard wild card characters (*, ?) are accepted. To archive all the results files in the current subdirectory, simply type in "*.res" when prompted for the analysis results file name.

The static archiver also has the capability to store several versions of a program, and note the modifications of each different version. For more information about this function and other archiving capabilities, refer to the Logiscope manuals.

1.4.1.2.3 Editor Procedures for Generating Output

When the program is analyzed and archived, users can run the static editor to create, manipulate, and print results in tabular and graphical formats. The editor is where most of the user interface takes place, as this is where results and output are generated. Examples of some of the editor functions are the generation of control graphs, graphical representations of the distributions of metrics, textual representations of the code structure, distributions of the metric functions, and metric statistics.

The editor can perform operations on both archive files or individual results files, and there are workspace manipulation commands to look at subsets of archive files for large programs. Some of the basic editor functions are described in the sections below, and it is assumed that an archive file was loaded into the editor rather than a results file.

The static editor is invoked by issuing the following command from the command line:

```
log_edsta
```

The program will then prompt for the name of the archive or results file. After loading in the file (which can take several minutes for large programs), Logiscope will respond by displaying the editor prompt (>). Two commands to remember: "h" to display the help file and "end" to exit the editor session.

To get acquainted with the Logiscope editor, a quick tour of some of the capabilities and log on procedures is given in section 1.4.4. This tour utilizes a user-generated menu system and provides a graphical interface to initiate the Logiscope editor.

1.4.1.2.3.1 Metrics Generation

One of the easiest tasks to accomplish in the editor is the generation of the metrics file. This is an ASCII text file which can be imported to a spreadsheet or database. Issuing the **metfile** command will generate a metrics file, which will be found in the current directory with the same filename as the archive file, but with a .met extension. This metric file will contain all the metrics which have been selected for collection as defined in the reference file. For information on the selection of metrics for collection, see section 1.4.1. The structure of this metrics file for two components is shown in table 4.

Table 4 - Metrics File Format

```
Components: CHARACTER_MAPPING/MATCHES:STRING:NATURAL:PICTURE_MAP:
            BOOLEAN:INTEGER 17/Sp/92-11:57:24
Application: itve
Language:    ADA
Version:     1
Metric:      N_STMTS PR_LGTH N_ERRORS EFFORT VG MAX_LVLS N_PATHS
             N_JUMPS N_EXCEPT N_ACCEPT DRCT_CALLS N_RAISES COM_FREQ
             AVG_S 20 306    0.48 54666.62 13 3 13 0 0 0 0 0    0.00    15.30

Components: PARSER_SERVICES_PKG_TT/FUNCTION_PARSE_TT:CURRENT_RECORD_
            TYPE:ECHO 17/Sp/92-12:51:40
Application: itve
Language:    ADA
Version:     1
Metric:      N_STMTS PR_LGTH N_ERRORS EFFORT VG MAX_LVLS N_PATHS
             N_JUMPS N_EXCEPT N_ACCEPT DRCT_CALLS N_RAISES COM_FREQ
             AVG_S 34 231    0.33 31349.49 11 4 66 0 1 0 9 8    0.38    6.79
```

Notice that the actual values of the metric parameters are space-delimited in the seventh line of the section. When transferring this data to a spreadsheet or database, it is often easier to eliminate all but the component name and the actual metric values, thus removing the filler information. This can be accomplished easily using the find and replace feature of any word processor, replacing each occurrence of "Application: itve" with nothing. Cleaning up the metrics file makes it much easier to transfer to a database or spreadsheet. The file can be transferred into EXCEL as a space-delimited text file, where data analysis functions such as generation of the correlation coefficients, generation of the frequency histograms, and regression against errors can be performed.

Statistics can also be generated for each metric collected by issuing the stat command. A sample output of the stat command is shown in table 5.

Table 5 - Sample Statistic Table

Metrics	Mnemonic	Average	Standard Deviation	Min	Max	% ok	% Undef
Number of statements	N_STMTS	27.61	40.12	1	497	96%	0%
Number of comments	N-COM	14.28	10.18	0	86	99%	0%
Total operand occurrences	TOT-OPND	87.52	143.14	0	2070	85%	0%
Different operands	DIFF-OPND	26.08	23.84	0	139	77%	0%
Total operator occurrences	TOT-OPTR	126.12	200.13	2	2949	83%	0%
Different operators	DIFF-OPTR	15.17	5.89	2	37	76%	0%
Program length	PR-LGTH	213.64	342.29	2	5019	83%	0%
Estimated error count	N-ERRORS	0.36	0.77	0.00	12.38	84%	0%
Mental effort	EFFORT	66983.54	3.91E+05	36.18	7.15E+06	84%	0%
Coding time	CODE-T	3721.30	21754.91	2.01	3.97E+05	84%	0%
Cyclomatic number	VG	7.30	10.79	1	106	91%	0%
Max number of levels	MAX-LVLS	2.62	1.29	1	7	97%	0%
Comments frequency	COM-FREQ	0.93	0.94	0.00	13.00	63%	0%
Average size of statements	AVG-S	7.75	3.28	2.00	25.73	44%	0%
Number of IN_OUT nodes	N-IO	2.07	0.27	2	4	93%	0%

Statistics Table	
Application:	sorter
Version:	VERSION1
Language:	FORTRAN
Components:	350

Logiscope also has the capability to print out frequency histograms for each metric. This histogram gives the overall percentage of components which exhibit a specific value range of a metric. After issuing the **metbars** command, a graphical window will appear with the distribution of the first metric of the set. Pressing the enter key with the mouse pointer in the Logiscope text window will cycle through the metrics list. If a hard copy is desired of any of the screens, issue the **p** command. At the conclusion of the session, Logiscope will generate a postscript (.pos) file containing all the graphical print requests. The hard copy can be generated by sending the postscript file to the printer. One word of warning: Logiscope sends all graphical hard copy requests to a single file. If the user requests a large number of hard copies, this file might be too large to send to the printer. Text hard copies are saved in a similar manner, but in an ASCII file with a .al extension. Text files are usually much smaller, so the file size is not really a problem.

1.4.1.2.3.2 Control Graph Procedures

A control graph is a graphical representation of a program's structure, illustrating graphically the decision paths and logic structure. To create a control graph for a given component, type "control" at the editor prompt. Logiscope will bring up a graphical window which displays the control graph. The control graphs of all components in the current workspace can be cycled through by pressing the enter key in the text window. Typing "prv" at the prompt will return the user to the previous graph. Figure 3 provides the definitions of some of the symbols represented on the control graphs.

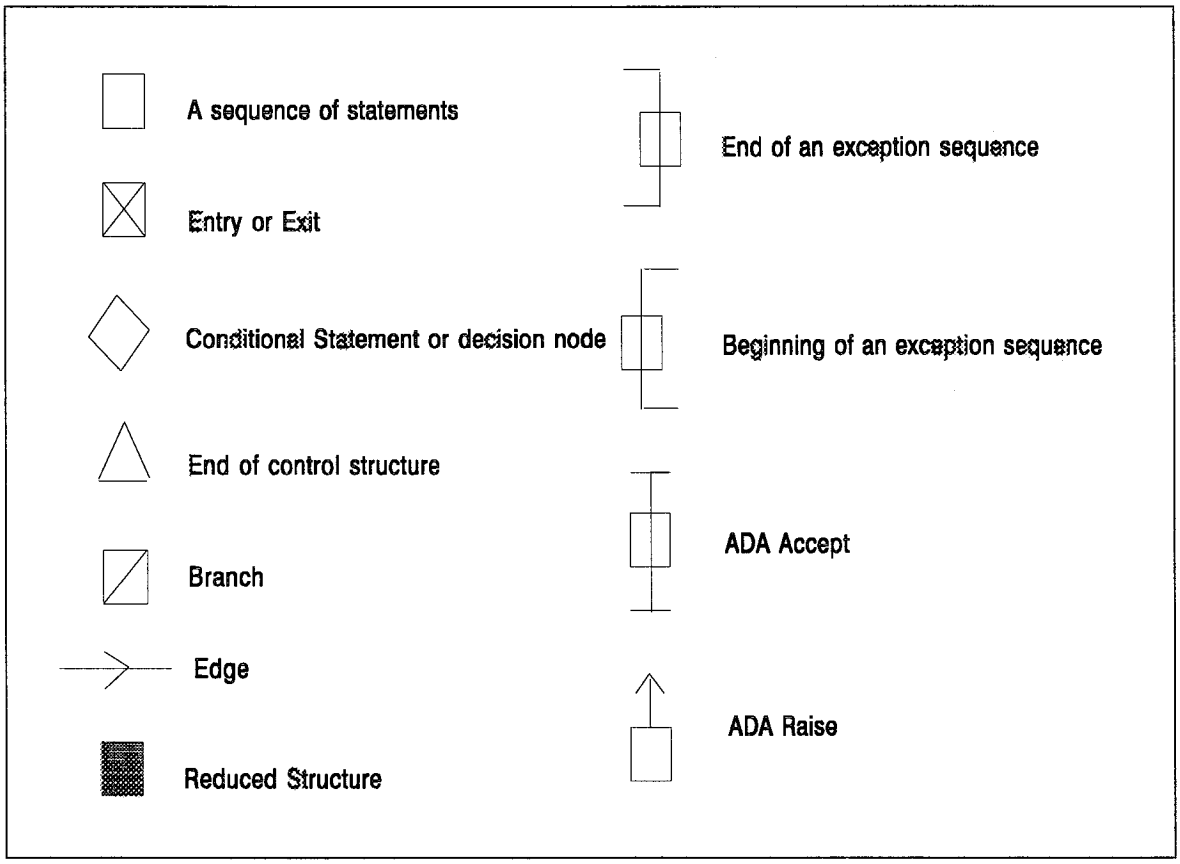


Figure 3 - Control graph symbols.

Control graphs can be used to measure the level of structuredness of a program's individual components. This is accomplished by a technique known as control graph reduction. Reducing a control graph entails representing a nested structure with a single node, as long as the structure can be represented with a single entry and a single exit. After successive reductions, a well-structured component will have an equivalent cyclomatic complexity equal to one. This number is often referred to as the essential complexity. To reduce a given control chart, type "red" at the control graph prompt. The graphical window will then show a modified control graph with nested structures represented by the reduced structure symbol. Successive iterations of the "red" command will show a complete reduction of the component.

Pseudo code generation and representation of decision nodes by the appropriate line numbers is also possible when looking at control graphs. Typing "txt" will generate pseudo code for the selected component. Typing "nn" will give the line numbers on the control graph itself.

1.4.1.2.4 Kiviat Graph Procedures

A Kiviat graph is a method of graphically displaying a given set of metric values against a predetermined set of limits for these values. Kiviat graphs can be displayed by issuing the **kiviat** command at the Logiscope prompt.

1.4.1.2.5 Workspace Management Procedures

When working with a large program, it is sometimes useful to reduce the size of the workspace to look at only portions of the program. This can provide an increase in speed and efficiency of Logiscope, as well as allowing the user to focus in on a specific subset of programs. This can be accomplished by the **wssel** command.

For example, to consider only the components with a cyclomatic number greater than 20, issue the following command:

```
wssel (vg > 20)
```

This would limit the workspace to only those components with cyclomatic number greater than 20. The metric distribution histograms, control graph displays, kiviat graphs, etc., would only include components with a cyclomatic number greater than 20. The workspace management commands can also be chained together. For example, to look at only files with a cyclomatic number greater than 20 and with the number of statements less than 70, issue the following command sequence:

```
wssel (vg > 20)
wssel (n_stmts < 70)
```

This sequence of commands would limit the workspace to those components which satisfy both of the search conditions.

Once a workspace has been defined, the workspace (actually the subset of components which match a given criteria) can be saved by issuing the **wssave** command. This workspace can be read from the file by issuing the **wsrest** command.

Another useful workspace management command is the **wslistcont** command. Issuing this command displays a list of all components in the current workspace. Table 6 provides a list of some of the workspace management commands and their uses. For more information on more command or details on how to use these commands, refer to the Logiscope manuals.

Table 6 - Workspace Management Commands

COMMAND	DESCRIPTION
wssel	selects the workspace given a search criteria
wssrest	restores a previously saved workspace
wssave	saves the current workspace
wslistcont	lists the components in the workspace
wsinit	returns to the initial workspace
wsadd	adds a selected component to the workspace
wsinter	finds the intersection between two workspaces
wsdiff	find the differences in two workspaces
wscreate	creates an empty workspace

1.4.2 Metrics Database Format and Input

A software failure history database can be created to improve the error prediction modeling capabilities and accuracy. This database should be defined to include metric data, software failure data, and reliability growth information. Such a database could be used to expand research in areas such as: reliability characteristics of reused or modified code, software failure effects analysis, and the estimation of the reliability growth of a software system prior to the beginning of testing. Some proposed input parameters are given in table 7.

Table 7 - Basic Parameters for Input Into Metrics Database

Component name
Language
Failure history
Structural complexity metrics
Run time history
Reliability growth curve parameters
Development type metrics
Requirement metrics

This list is very rudimentary, but can give an idea of what is required. A lot of work will be needed to standardize the data collection process. The specific metrics for collection need to be defined so the information can be used from project to project.

1.4.3 Dynamic Analysis for Determination of Test Case Coverage

1.4.3.1 Dynamic Analysis Using Logiscope

Logiscope provides the capability to perform dynamic analysis of a software system. The program has instrumentation options which correspond with the above techniques. Logiscope automatically instruments the source code, which must then be recompiled along with a specific file (provided by Logiscope). The procedures for analyzing, archiving, and editing are very similar to the procedures for performing static analysis.

The dynamic analysis capability of Logiscope can be obtained from the Logiscope manuals and a report from Dennis Braley of the JSC Software Technology Branch entitled "Test Case Effectiveness Measuring Using the Logiscope Dynamic Analysis Capability" [8]. Mr. Braley has successfully performed dynamic analysis on some of his code, and his report of the details of his study is available in the STIC.

1.4.3.1.1 Code Instrumentation

The instrumentation of the source code is performed during the analysis of the source code much in the same fashion as the static analyzer. When first accessing the analyzer, the same following questions are asked:

```
Source_File name  ?
Basic counts (1=screen, 2=file, 3=not)    ?
Instrumentation  (y/n)   ?
```

The main difference will be noted by answering y to the third question. The program will then prompt for three more inputs:

```
Creation of a listing  (y/n)  ?
Instrumentation of control data  (y/n)  ?
Instrumentation of call data  (y/n)  ?
```

The following script can be used to analyze all the files in the subdirectory:

```
set *.f
while test $1
do /case/Logiscope/bin/log_fort_vax <<!
$1
3
y
y
y
y
y
n
!
echo $1
shift
done
```

1.4.3.1.2 Compilation of Instrumented Code

Compilation of the Logiscope instrumented code is accomplished in the same manner. To allow the SUN FORTRAN compiler to compile the files, they must end in the .f extension. Logiscope gives all the files a .for extension. The following shell script can be used to change the filenames of the .for files to the .f extension:

```
for i in `ls *.ftn|awk -F"." '{print $1}'`
do
mv ${i}.ftn ${i}.f
done
```

1.4.3.1.3 Other Procedures

To perform dynamic analysis using Logiscope, a user also will need to execute test cases, to run editor procedures for generating output, and to archive results. Detailed information regarding these processes can be found in the Logiscope manual.

1.4.3.2 Dynamic Analysis Using Compiler Directives

Dynamic analysis can also be performed using compiler directives. Many compilers have this option, and the procedures described in this section are relevant to the SUN FORTRAN compiler. This compiler is resident on the workstation modeller. Compilation directives can be used to perform a decision-to-decision path test monitoring on a particular software component as well as identifying the frequency of execution of a particular decision path.

1.4.3.2.1 Code Instrumentation and Compilation

Instrumenting the code (or inserting the traps in the code) can be performed at the compiler command line. For the FORTRAN code, this was done by:

```
f77 -c -a <source filename>
```

This creates the instrumented versions of the object file(s). An instrumented executable can be created after all objects have been created by :

```
f77 -a *.o -o <executable filename>
```

An instrumented version of the executable is now created.

There is a loss of performance depending on the level of instrumentation. The more components which are instrumented the slower the program will run. Instrumentation will compromise the accuracy of profiling data (for more information on profiling see section 2.2.2). It is recommended that profiling be done separately from instrumentation.

1.4.3.2.2 Test Case Execution

Now that the instrumented code is compiled and linked, the test cases of the program can begin to be executed normally. Test case execution procedures will depend on the particular system being analyzed, and test execution procedures should be followed as documented in the test plan produced by the developer. For the monitoring to work properly, the program must terminate normally without any errors in the execution of the program. Upon conclusion of each individual test case, a file will be created in the executable directory with the component name followed by a .d extension. This .d file is what will be used by the **tcov** utility to determine the test coverage for that particular component.

Since the .d files are created at the termination of a test case execution, we recommend that these files be processed and archived for each individual test case before executing another test case. This allows all the coverage information for one particular test case to be processed and collected before proceeding on to the next case. An alternative process is to move the *.d files into another archive directory and complete the execution of the next test case. At the conclusion of the testing, there should be a separate subdirectory for each test case, each subdirectory containing the raw test coverage files (the .d files).

1.4.3.2.3 Collection and Consolidation of Results

Once the test case has been executed and the .d files have been generated, you are now ready to generate the output files by running the UNIX **tcov** utility. The **tcov** will generate a file with a .tcov extension corresponding to the .d file it was run on. The **tcov** utility is run by typing "tcov" and then the component name:

```
tcov <component name>
```

This creates a text file with the name of the component and a .tcov extension. An overview of the consolidation of the results is shown in figure 4.

The command

```
more <component name>.tcov
```

will display the file on the screen. The file is a listing of the source code, with a couple of modifications. Listed in the left margin prior to each decision node is the number of times each decision node has been executed. If a particular node has not been executed, ##### will be displayed as the number of times. At the end of the source listing, additional information is given. This includes the top 10 blocks (relative to execution frequency), the number of blocks in the file, the number of blocks executed, and the percentage of the file executed. The structure of the .tcov file is shown in figure 5.

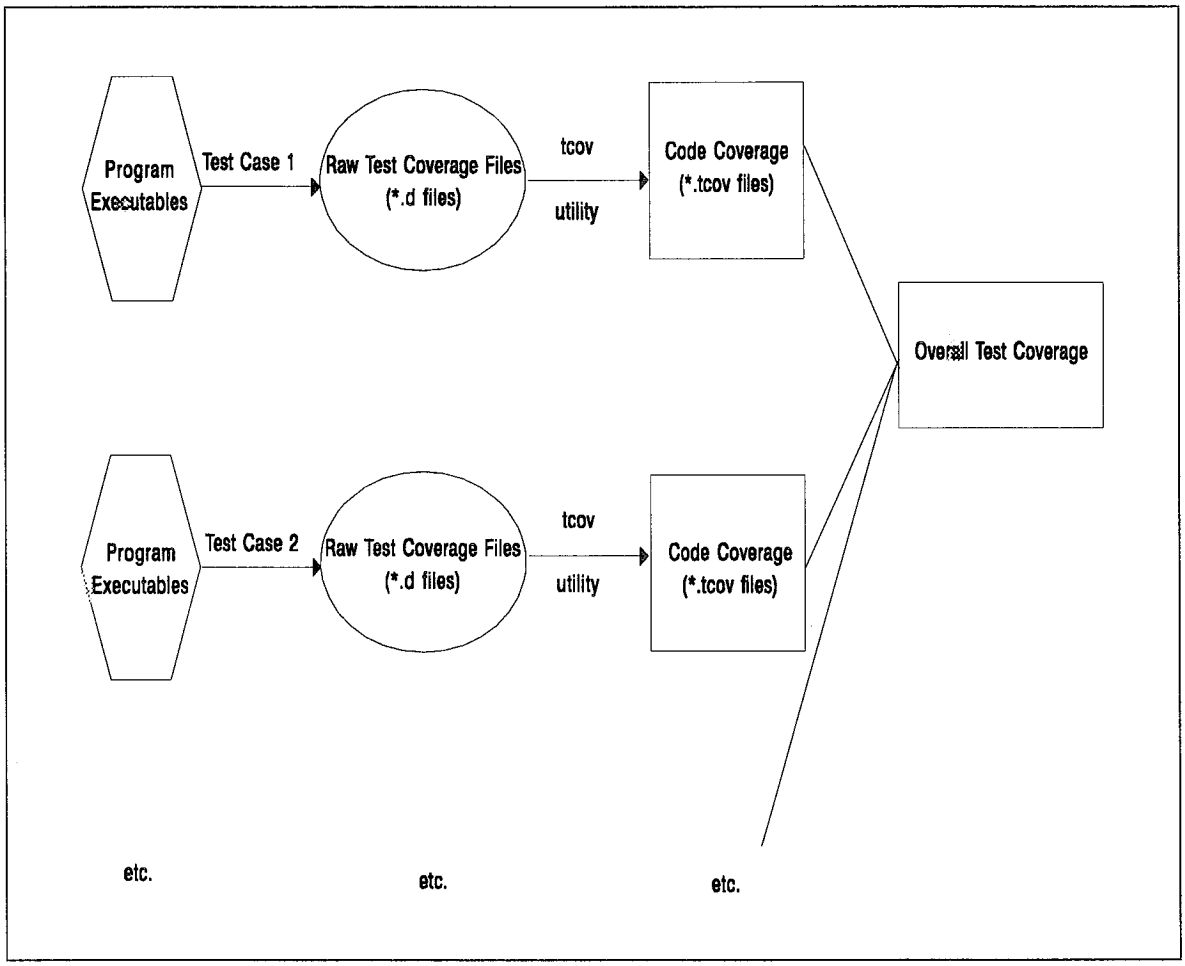


Figure 4 - Test coverage results consolidation.

```
        C
C*************************************************************
        C
             IMPLICIT DOUBLE PRECISION (A-H, O-Z)
             CHARACTER*(80) SCCSID
        C
             DATA SCCSID /'@(#)amdval.f 4.1 10:01:27 90/09/27'/
        C
        C...ORDER POINTS A AND C
        C
   4004 ->         IF ( A .LE. C ) THEN
   4004 ->           PT1 = A
                   PT2 = C
                ELSE
  ##### ->           PT1 = C
                 PT2 = A
                ENDIF
        C
        C...COMPUTE MIDDLE VALUE
        C
   4004 ->        IF ( B .GT. PT1 ) THEN
   1954 ->         IF ( B .LE. PT2 ) THEN
   1954 ->           AMDVAL = B
                  ELSE
  ##### ->           AMDVAL = PT2
                  ENDIF
   1954 ->        ELSE
   2050 ->         AMDVAL = PT1
                ENDIF
        C
C*************************************************************
        C
   4004 ->        RETURN
                END

          Top 10 Blocks

          Line    Count

             59     4004
             60     4004
             69     4004
             81     4004
             76     2050
             70     1954
             71     1954
             75     1954

 10       Basic blocks in this file
  8       Basic blocks executed
80.00     Percent of the file executed

23928     Total basic block executions
2392.80   Average executions per basic block
```

Figure 5 - .tcov file structure.

A simple script can be created to automatically process all the output files in the current directory. The following script will run the **tcov** utility on all the files in the current subdirectory:

```
set *.d
while test $1
do tcov $1
echo $1
shift
done
```

When processing the **tcov** utility on individual components to determine the coverage of individual test cases, check the amount of available disk space. The **tcov** utility creates a file which is just slightly larger than the original source code listing. The output files for a single test case (for full code monitoring) will take as much room as the original program, plus about 10-15%. When executing multiple test cases, consumption of excess disk space can be a problem.

It is less time-consuming to create a script file to execute each individual test case as well as move the output to its own subdirectory and run the **tcov** utility output files. Each test case can be run successively using the batch mode, using the **at** feature of the batch mode. The **at** feature allows one to execute a given command sequence at a specified time. Enough time must be allowed for the previous execution to complete.

There was a problem associated with the dynamic analysis. During the execution of the test cases and the processing of the output data, the workstation would freeze up, requiring a reboot to restart. After the reboot, all the data files would be lost, forcing a rework of what was already accomplished. After a period of troubleshooting, the problem was rectified. The program being run along with the monitoring overhead and tcov processing became very disk- and memory-intensive. If any other processes were initiated during the execution of the test case and tcov processing, the system would crash. Therefore, a dedicated processor is needed when performing the dynamic analysis. This can be a problem in a multi-user environment during normal working hours, so the analysis should be done during off hours. One additional problem with performing this type of run during the off hours is that the machine (modeller) has an accounting program set to run daily. It might be necessary to have your systems administrator either disable the accounting program or work around the expected hours of operation using the **at** command.

Once the test coverage files have been generated for each test case, it is now necessary to calculate the total amount of test coverage for the entire set of test cases. The output from the **tcov** utility provides the output from a single test case, but there is no real straightforward way to combine the outputs into one overall number. The following is a summary of a method to combine outputs:

- create a file for each test case formatted with the name of the component and the unexecuted decision blocks in that component for that particular test case
- compare the coverage outputs of two separate test cases on the same component to record the test case deviations between the cases on that component
- assume that the frequency of execution of a block is unimportant (executing a block once is the same as executing a block 1000 times for this analysis)

- continue comparing the outputs of the two files, monitoring the test case deviations until all the test cases are covered

The above method is the method used to determine the overall test coverage, but it might not be the most efficient one. Further experimentation might yield better results.

To automate this process as much as possible, the users may use script files. For example, the actual script files used to perform this task for the DADS system are resident in **tgunn** on a Sun workstation. They are summarized below. To create a file with the file name and the unexecuted decision blocks, use the **grep** utility:

```
grep '#####' *.tcov > $1_b.unc
```

where the $1 is the variable representative of the test case name. Since each unexecuted decision block was preceded by the ##### string, the .unc file contained all the untested blocks for that particular test case. After collecting the untested blocks for the test cases, the **diff** utility could be used to compare the list of the untested blocks by:

```
diff <list 1> <list2>
```

This listing was manually compared and the deltas in the test case coverages were noted. A sample script, which compares the untested decision nodes in test case P1A to case P1B, is shown below:

```
cd P1A_mod/a
grep ##### *.tcov >/users/tgunn/tmp/p1a
cd ../../P1B_mod/a
grep ##### *.tcov >/users/tgunn/tmp/p1b
diff /users/tgunn/tmp/p1a
   /users/tgunn/tmp/p1b>/users/tgunn/tmp/p1a_b
```

The total number of blocks for each file were extracted again by using the **grep** utility:

```
cd $1/b
grep 'Basic blocks in this file' *.tcov  > bblocks.out
cd ../../$1/a
grep 'Basic blocks in this file' *.tcov  >ablocks.out
cd ../../$1/s
grep 'Basic blocks in this file' *.tcov  >sblocks.out
```

and the percentage of total coverage per test case was exported by:

```
cd $1/b
grep 'Percent of' *.tcov  > $1_b.out
mv $1_b.out $BIAS
cd ../../$1/a
grep 'Percent of' *.tcov  > $1_a.out
mv $1_a.out $ARDG
cd ../../$1/s
grep 'Percent of' *.tcov  > $1_s.out
mv $1_s.out $SHAP
```

We then imported the results from these script files into an EXCEL spreadsheet by performing an ASCII import of the data. We calculated the total test coverage by dividing the actual number of blocks executed (found by reviewing the **diff** outputs) by the total number of blocks in the file.

One problem did occur for several files in one portion of the FORTRAN project. The **tcov** utility could not process the information in these files, and NaN (not analyzed) was listed as the test coverage and 0 was listed as the total blocks in that file. The number of components with this problem was found by the following script:

```
grep 'Basic blocks in this file'
/users/tgunn/DADS_TEST/bin/EXECUTABLES/$1_mod/b/*.tcov>/
   users/tgunn/tmp/temp
grep ' 0' /users/tgunn/tmp/temp |wc -l
```

Test coverage for calculation of the risk index for this particular program was estimated by the amount of reused code resident in this piece of code. The reason for the inability to process these files is still undetermined, but we assume that it is either due to the iterative nature of the particular program's execution or due to the inability of the **tcov** utility to process files beyond the first parent/daughter level. Experimentation can be done by altering the calling script to further isolate this problem.

After the total test coverage percentages have been calculated, the coverage distributions can be represented by frequency histograms. Frequency histograms can be compiled either by using EXCEL or by using Statgraphics. To create a histogram under EXCEL, simply highlight the total test coverage column, select tools on the main menu, then select analysis tools, and then frequency histograms. This graph is usually more meaningful if the Pareto option is not selected. To create a frequency histogram using Statgraphics, it is necessary to import the data into Statgraphics.

1.4.3.2.4 Profiling

Profiling data is collected using a combination of compiler directives and UNIX utilities. Profile data was collected for the FORTRAN project, and the procedures in this section are applicable to profiling FORTRAN code using the SUN FORTRAN compiler. Profiling in other languages can be accomplished by modifying the compiler directives, dependent upon the language and the compiler used.

There are two methods of profiling options using the SUN FORTRAN compiler: the **-p** option and the **-pg** option. Each option must be used with its corresponding UNIX utility; **prof** for the **-p** option and **gprof** for the **-pg** option. The **-pg/gprof** option is what was used to provide a much more extensive profile information, but both methods will be described in detail. We later determined that the added information obtained through the use of the **-pg/gprof** options didn't seem to be applicable for our purposes. The -p/prof combination is recommended.

To profile a program (in FORTRAN), issue the following commands when compiling:

```
f77 -p <program name> <executable name>
```

You can then execute the program. Upon conclusion of the program run, a mon.out file is created which contains the profile information. The **prof** utility can then be used to interpret the results stored in this file by the following command:

```
prof <executable name>
```

Five columns of data scroll across the screen: name, % time, cumsecs, #call, and ms/call. Each of the columns has the following meanings:

name	the name of the function being monitored
%time	the percentage of the total execution time of the program spent in that particular function
cumsecs	the cumulative seconds spent in the various parts of the program
#call	the number of times a specific function was called during the execution of the program
ms/call	the number of milliseconds per call of the specific function

To profile a program using the **-pg/gprof** option, simply replace the above **-p** with a **-pg** and **prof** with **gprof** in the above procedure. This method provides similar data, as well as an abundance of data which is not really applicable for our use. More information on profiling can be found in the SUN FORTRAN User's Guide and McGilton and Morgan's Introducing the UNIX System [9].

1.4.4 Logiscope Log On Procedures

1.4.4.1 Remote Log On Procedures

Logiscope can be remotely logged on with proper setup. The ftp id's for some of the machines are listed below. Keep in mind that if the data is to be extracted data from galileo, any machine can be used (as long as it is set up to access it), but if Logiscope is needed, an X Windows terminal is required.

```
Ned_jsc1        141.205.3.61
venus           141.205.3.6
galileo         134.118.101.24
modeler         141.205.3.60
mercury         134.118.124.10
```

To remotely log on, simply use the rlogin or telnet commands to access the desired machine. The rlogin and telnet commands are described in more detail in section 1.5.

To use Logiscope, first start up X Windows on the local machine and set up the machine so galileo can display on the machine by typing:

```
xhost  +galileo       (or host  +134.118.101.24)
```

Then use the rlogin or telnet command to remotely log on to galileo, and press return when the terminal asks for the display type. Next set the display variable on the machine by typing:

```
setenv DISPLAY <local machine ftp address>
```

1.4.4.2 Terminal Configuration

To run the Logiscope program, the working environment must be configured properly. This includes allowing the local workstation to be controlled by galileo (using the **xhost** command) and making sure the account settings on galileo are properly configured and the environment is properly sey up. This is accomplished by making modifications to the .cshrc file on the galileo machine (not the local machine). Add the following lines for access to Logiscope to the .cshrc file:

```
set path = ($path /case/logiscope/bin)
setenv  REF  /gal_home/adp/logidir/reference
alias   fa   echo '32, hp, ibm, nd, plus, std, vax'
```

These modifications will allow Logiscope to run properly. If the REAP menu system is to be used, the following line should be added to the .cshrc file:

```
source ~bmears/REAP/reap_setup
```

The REAP menu system is described in more detail in section 1.4.4.4, A Quick Tour of Logiscope.

The .cshrc file should already be set up, but these commands are convenient in case something happens. Two other additions to the .cshrc file which were useful were two alias commands, used to set up the display environment:

```
alias seethe   'setenv DISPLAY <local machine ftp address>'
alias showd    'echo $DISPLAY'
```

These commands allow the user to quickly set up the display environment string to the machine being used by typing in "sethome" rather than something like "setenv DISPLAY 141.205.3.60". The **showd** command is just a way to verify that the user is configured at the right machine. If the display environment is not set properly, graphics images requested could be sent to another machine.

1.4.4.3 Logiscope Startup Procedures

1.4.4.3.1 Using the Menu System

The Information Systems Directorate has created a menu system to allow the Logiscope tool to be accessed with more ease than the command line interface. To initiate the menu system, type "reap". This will bring up two menus, a controller menu, and a main menu. Logiscope is found under the General Support section of the main menu. Double click on the General Support with the left mouse button, and double click on the Logiscope Tools selection. The

REAP menu system will then prompt for what the desired operation is. Double click on the operation and fill in the appropriate blocks as necessary.

When accessing the static or dynamic editor, it is necessary to insert the path of the appropriate reference file for the analysis. Simply change the defaults to the file needed. The same is done for the archive filename when archiving results files.

In general, the REAP system provides a user friendly alternative to Logiscope's command line interface. One slight difficulty was noted with the menu system, which will be changed for future releases. Only one FORTRAN analyzer exists on the menu, which defaults to the FORTRAN standard analyzer. If you want to use other FORTRAN analyzers, it is necessary to set an environment variable before running REAP. This is accomplished by the following command (which must be issued before running REAP):

```
setenv LSCOPE_FORTRAN_ANALYZER <Logiscope FORTRAN analyzer
   executable>
```

The Logiscope FORTRAN executables were listed in table 1. An example of this command would be to type "setenv LSCOPE_FORTRAN_ANALYZER log_fort_plus" to run the F77+ analyzer. An alias has been set up to list the FORTRAN analyzers on the fly. Typing "fa" will list the FORTRAN analyzers available.

1.4.4.3.2 From the Command Line

There are a variety of run commands and execution switches which can maximize the execution efficiency of the program for a specific application. For more information on these types of command line options, refer to the Logiscope manuals.

1.4.4.4 A Quick Tour of Logiscope

Logon to local terminal (with X Windows capability)
startx - runs the X Windows program

xhost + <your resident terminal name>

telnet <your resident terminal name>
username:
password:
Terminal Type: **<xterm>** (should default to this, press return)

sethome - directs the graphical display to your resident terminal name; when on a different machine, type "setenv DISPLAY <node id>"; example: when accessing from a different terminal, type "setenv DISPLAY 130.40.10.86:0"

showd - verify display is set to node id

Logiscope can be accessed in two ways, using the menu system or accessing from the command line. The following command will start the menu system. To access from the command line, the Logiscope tool is resident in the /case/logiscope subdirectory.

reap - starts the menu system

The Logiscope tool is under the general support section of the main menu. Logiscope contains three main components, an analyzer, an archiver, and an editor. The analyzer is used to analyze/instrument raw source code. It will create a .res file for each component analyzed. The archiver is used to group .res files together into a .arc file, which can be read by the editor. The editor is where we will do the majority of our work. Some sample editor commands are:

control	- displays the control graph of the selected component (enter cycles through the components, **q** quits)
call	- displays the call graph of the system (usually takes a while)
crit	- displays the criteria graph of the selected component
kiviat	- displays the kiviat graph of the selected component
metbars	- displays the metric distribution charts (enter cycles through the metrics, **q** quits)
h	- displays the general help screen
h <cmd>	- displays help on specific command
p	- outputs screen to a print file
end	- ends session

The editor can be toured using the output data from one of our programs (or some of the sample files in the /case/logiscope directory). The REAP menu should default to the data generated for our program. If this fails, the filename is cmslib/cmsout/itve.arc and the reference file is reference in the home directory alee.

1.5 UNIX Basics and Commands

These are several simple commands which enabled an idiot like me to get around. I thought it might be handy to have a "cheat sheet" of some nifty commands.

cp - copies files

rm - removes files

ls - lists files in directory
- **-l** - displays a more detailed file listing (date, size, etc)
- **-a** - displays all files, including hidden

man <cmd> - displays an on-line manual for the selected command.
sample usage: man ls - displays the manual for the list command

mv - moves files from one directory to another, or renames files
sample usage: mv nulan geek - renames file nulan to geek

ftp <hostname> - Enters the file transfer procedure. Some options are:
get <file> - get a file from <hostname> and copy to current directory
mget <files> - get multiple files (use with wildcard) from <hostname> and copy to current directory
put <file> - puts file on hostname
Hint for use: If you have a lot of files to transfer, you can eliminate that annoying need for answering "Y" prior to each file by typing "ftp -i."

ps - displays process status (processes active on the machine)
-a - displays all users processes

vi <file> - invokes the vi test editor. Some helpful commands are:
:w - writes the file to disk
:q - quits vi
dd - deletes entire line
x - deletes character
i - inserts characters (>esc< to quit inserting)

rlogin <name> - remote login to another machine (use when accounts are the same)

telnet <name> - remote login to another machine (use when accounts are the different)

alias - allows substitution of specified keystrokes for commonly used (usually longer) commands. For example, if you wanted ll to display a long list, type "alias ll 'ls -l'."

Unalias - removes the aliases on a command. For example, **unalias ll**

more <file> - lists file, one screen at a time (v to directly edit screen)

cat <file> - lists file

diff <file1><file2> - compare two files and prints out the differences on the screen

grep - searches files for specific string

batch - invokes the UNIX batch command processor

at - similar to batch, allows one to execute commands at specific times

SECTION 2: SOFTWARE RELIABILITY ESTIMATION AND PREDICTION

2.1 Introduction

Since the early 1970's tremendous growth has been seen in the development of software. Software has increased in complexity and size at a rapid rate. Part of this change is a result of the development of faster microprocessors, distributed processing systems, and networking. Software today performs a myriad of complex tasks that require accuracy and reliability. The development of failure-free software has always been a major goal in any organization. Testing until the software is totally failure free, however, may be an impossible task for large programs (it would require checking all the combinations of logic paths and inputs). Software reliability estimation and prediction, or software reliability, can provide a practical insight method into the failure state of the code during the testing process.

Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment. The probability of failure-free operation can be expressed using failure intensity. Failure intensity is defined as the rate of change of the number of failures per unit time (usually expressed as failures per 1000 CPU hours). Therefore, a specific failure intensity objective (or goal) can be set for a piece of software depending on the requirements and the application of such software (usually defined to be 0 to 10 failures per 1000 CPU hours for flight software, and 10 to 20 failures per 1000 CPU hours for ground support software as used in industry, academia, and the Department of Defense). Other expressions of software reliability are described in terms of the mean time to the next failure (when the next failure will occur), the maximum number of failures (total number of failures expected from the software), and the hazard rate (the conditional failure density).

The current status of software reliability can forecast the projected testing termination date. It can also aid in determining how to best allocate testing resources (manpower, computer time) among the various program modules. Software reliability can also be used in making decisions regarding design tradeoffs between reliability, costs, performance, and schedule. The main thrust behind software reliability, however, is to provide a quantitative means to measure the probability of failure-free operation of a computer program.

2.1.1 Models

There are currently over 40 software reliability models. These models can be categorized into time domain, data domain, and fault seeding approaches. The focus of this section will be on the time domain approach. Detailed description of the models can be found in section 2.2.1.

2.1.2 Tools

Two tools used by JSC SM&A are the Software Reliability Engineering (SRE) written at the AT&T Bell Laboratory, and the Statistical Modeling and Estimation of Reliability Function for Software (SMERFS Version 4.0) tool developed by the Naval Surface Warfare Center. Concepts behind the SRE Toolkit can be found in Software Reliability Measurement, Prediction, Application by Musa, Iannino, and Okumoto [12]. Detailed information on the SMERFS Tool can be found in Statistical Modeling and Estimation of Reliability Function for Software (SMERFS) Users Guide by Farr [14]. These tools accept test data such as computer

execution time for software, failure occurrence time, and resource utilization and related calendar time information as input data (see section 2.3). The tools then estimate the reliability of the software using the approaches of the different software reliability models.

2.1.3 Data Collection and Analysis

Software reliability uses the software testing data at and beyond the software integration phase. This data includes total execution time, failure description comments, and the exact time the failure occurred. Some models require additional data that include computer resource, computer utilization, and manpower resources. Detailed description of the data collection and analysis can be found in section 2.4.1.

2.1.4 Modeling Procedure and Analysis

The SMERFS Tool and the SRE Toolkit contain mathematical models that are used in estimating and predicting failure occurrences. The input failure data is collected and analyzed for applicability to each model's assumptions. Failure data that matches a particular model's assumptions is used to estimate and predict the present and future failure occurrences. The modeling results are presented in graphical or tabular form, depending on the particular model's interface. A detailed description of the procedures for each tool can be found in section 2.5.

The model results, in tabular or graphical form, show the estimated and predicted failure occurrences. In all cases, the models will report the approximate time to the next future failure. Models can report total predicted number of failures that will be encountered, time to reach a future failure, number of failures left to correct, and the maximum number of failures predicted (results are model dependent and may not be reported the same from model to model).

2.2 Background

Software reliability estimation and prediction is usually referred to as software reliability. This section will define software reliability, and will provide a table that outlines the differences between software and hardware reliability. In addition, a brief description of some of the software reliability models will be included.

Software reliability is defined by the Institute of Electrical and Electromechanical Engineers [9] as:

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 and the use of the system, and is also a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered."

and

2) "The ability of a program to perform a required function under stated conditions for a stated period of time."

Another definition by Shooman [10] is "the probability that the program performs successfully, according to specifications, for a given time period."

A more precise definition, as stated by Musa, Iannino, and Okumoto [11], is "the probability of failure-free operation of a computer program for a specified time in a specified environment."

A failure is defined as the departure of the external results of a program operation from the requirements. A fault is a defect in the program that, when executed under particular conditions, causes a failure. For example, a fault is created when a programmer makes a coding error. A failure is a manifestation of that fault. The reverse argument, however, is not true.

To understand the software approach to reliability, an understanding of the differences between hardware and software reliability needs to be established. The following table outlines those differences.

Table 8 - Contrasts of Software and Hardware Reliability

HARDWARE FAILURES	SOFTWARE FAILURES
Failures can be caused by deficiencies in design, production, use and maintenance	Faults are caused by design errors, with production (copying), use, and maintenance (excluding corrections) having negligible effect
Failures can be due to wear or other energy- related phenomena	There is no wearout phenomenon
Failures of components of a system is predictable from the stresses on the and components, other factors	Faults are not predictable from analysis of separate statements. Errors are likely to exist randomly throughout the program
Reliability can depend upon operating time (i.e. burn-in, wearout)	Reliability does not depend on operational time. It is a function of the effort put into detecting and correcting errors
Reliability is related to environmental factors	External environment does not affect reliability, except it might affect program inputs
Repairs can be made to increase reliability of the equipment	The only repair is by redesign (reprogramming)

2.2.1 Description of Models

Over 40 models that estimate and predict software reliability exist. A comprehensive theoretical discussion of many of the models may be found within A Survey of Software Reliability, Modeling, and Estimation, Naval Surface Warfare Center, NSWC TR 82-171 [12]. Another source of theoretical and practical approaches may be found in Software Reliability Measurement, Prediction, Application [11].

The models described in this section generally assume one of three approaches in estimating and predicting reliability:

- Time Domain: uses specific failure time data to estimate and predict reliability
- Data Domain: uses the ratio of successful runs vs. total runs
- Fault Seeding: depends on artificially placing faults within a piece of code

2.2.1.1 Time Domain Models

Time domain models estimate program reliability based on the number of failures experienced during a certain period of testing time. This can be either the computer execution CPU time, or the calendar time that was spent in testing. Specifically, time domain models predict the number of failures remaining, the time to next failure, and the test time required to achieve a reliability objective (goal). A brief description of the models is supplied in this section. The majority of the described models use the time domain approach.

Brooks and Motley's Models—This model assumes that in a given testing period not all of the program is tested equally, and in the development of a program, only some portion of the code or modules may be available for testing. In addition, through the correction of discovered failures, additional failures may be introduced.

Duane's Model—This model employs a non-homogeneous Poisson process for the failure counts. This model was originally proposed as a hardware reliability growth model.

Generalized Poisson Model—This model assumes that failures occur purely at random and that all faults contribute equally to unreliability. In addition, all fixes are perfect and the failure rate increases by the same amount except within the failure count framework.

Geometric Model—This model assumes 1) that a number of program failures that will be found will not be fixed; and 2) that failures are not equally likely to occur, and as the failure correction process progresses, the failures become harder to detect.

Goel-Okumoto—This model accounts for the fact that failures occur purely at random and that all failures contribute equally to unreliability. In addition, fixes are perfect and the failure rate improves continuously over time.

Geometric Poisson Model—This model assumes that the reporting of software failures occurs at a periodic basis. Only the numbers of failure occurrences per testing interval are needed. The testing intervals, however, are all assumed to be the same length with respect to time (a testing period is composed of a day, week, etc.). Additionally, since the model assumes a constant rate of failure occurrence during a time period, the model is best applied to situations in which the length of the reporting period is small in relationship to the overall length of the testing time.

Jelinski and Moranda "De-Eutrophication" Model—This model assumes that failures occur purely at random and that all failures contribute equally to unreliability. In addition, fixes are perfect; thus, a program's failure rate improves by the same amount at each fix. This model should only be applied to complete programs.

Jelinksi and Moranda's Model 1 and Model 2—The basic Jelinski and Moranda's "De-Eutrophication" model described above cannot be applied to software programs that are not complete. These models (model 1 and model 2) deal with software under development. If at any point in time a failure is discovered, an estimate of the reliability can be obtained based upon the percentage of the program completed.

Bayesian Jelinski-Moranda—This model assumes that failures occur purely at random and that all failures contribute equally to unreliability. In addition, fixes are perfect; thus, a program's failure rate improves by the same amount at each fix.

Littlewood's Bayesian Debugging Model—This model reformulates the Jelinski-Moranda Model into a Bayesian framework. It assumes that each failure does not contribute equally since the correction of failures in the beginning of the testing phase has more of an effect on the program than ones corrected later (a program with two failures in a rarely executed piece of code is more reliable than a program with one failure in a critical section of code).

Littlewood and Verrall's Bayesian Reliability Growth Model—This model assumes that the size of the improvement in the failure rate at a fix varies randomly. This represents the uncertainty about the fault fix and the efficiency of that fix. The model accounts for failure generation in the corrective process by allowing for the probability that the program could be worsened by correcting the failure. The tester's intention is to make a program more "reliable" when a failure is discovered and corrected, but there is no assurance that this goal is achieved. With each failure correction, a sequence of programs is actually generated (code is recompiled with each failure corrected). Each is obtained from its predecessor by attempting to correct a failure. Because of the uncertainty involved in this correction process, the relationship that one program has with its predecessor cannot be determined with certainty.

Littlewood—This model assumes that failures occur purely at random, have different sizes, and contribute unequally to unreliability. In addition, fixes are perfect; thus, a program's failure rate improves by the same amount at each fix. Larger failures, however, tend to be removed early, so there is a law of diminishing returns in debugging.

Littlewood's Semi-Markov Model—This model incorporates the structure of the program in developing its reliability. Littlewood adopts a modular approach to the software and describes the structure via the program's dynamic behavior using a Markov assumption. The program comprises a finite number of modules with exchanges of control between them that follow a semi-Markov law.

Littlewood Non-Homogeneous Poisson Process—This model assumes that failures occur purely at random, have different sizes, and contribute unequally to unreliability. In addition, fixes are perfect; thus, a program's failure rate improves by the same amount at each fix, and assumes a continuous change in failure rate (instead of discrete jumps) when fixes take place.

Keiller-Littlewood—This model assumes that failures occur purely at random, have different sizes, and contribute unequally to unreliability. In addition, fixes are perfect; thus, a program's failure rate improves by the same amount at each fix. Larger failures, however, tend to be removed early, so there is a law of diminishing returns in debugging. This model is similar to the Littlewood model, but it uses a different mathematical form for reliability growth estimation and prediction.

Modified Geometric "De-Eutrophication" Model—This model assumes that the number of program failures that will be found will not be fixed. In addition, failures are not equally likely to occur, and as the failure correction process progresses, the failures become harder to detect and do not have the same chance of detection.

Musa-Okumoto—This model accounts for the fact that failures occur purely at random and that all failures contribute equally to unreliability. In addition, fixes are perfect and the failure rate improves continuously over time. Later fixes, however, have a smaller effect on a program's reliability than earlier ones.

Musa Execution Time Model—The model is based upon the amount of central processing unit (CPU) execution time involved in testing rather than on calendar time; however, the model attempts to relate the two. By doing this, Musa is able to model the amount of limiting resources (failure identification personnel, failure correction personnel, and computer time) that may come into play during various time segments of testing. In addition, this model eliminates the need for developing a failure correction model since the failure correction rate is directly related to the instantaneous failure rate during testing. It assumes that the expected number of failures exponentially approaches an asymptote with time. This model is also known as the Musa Exponential Time Model (the reliability growth function increases exponentially with time).

Musa Logarithmic Time Model—This model is similar to the Musa Execution Time Model described above, except it assumes that the reliability growth function is a logarithmic function (increases logarithmically with time) instead of an exponential function.

Non-Homogeneous Poisson Process—This model assumes that the failure counts over non-overlapping time intervals follow a Poisson distribution. The expected number of failures for the Poisson process in an interval of time is assumed to be proportional to the remaining number of failures in the program at that time.

Rushforth, Staffanson, and Crawford's Model—This model assumes that failures occur purely at random and that all failures contribute equally to unreliability. In addition, fixes are not perfect and allow introduction of new faults into the program.

Schick-Wolverton Model—This model assumes that the hazard rate function is not only proportional to the number of failures found in the program, but proportional to the amount of testing time as well. As testing progresses on a program, the chance of detecting failures increases because of convergence on those sections of code in which the faults lie.

Shooman Model—This model assumes that the number of faults in the code are fixed, and that no new faults are introduced into the code through the correction process. The failures detected are independent of each other and the rate is proportional to the number of failures remaining in the code.

S-Shaped Reliability Growth Model—This model accounts for the fact that failures occur purely randomly and all faults contribute equally to total unreliability. Fixes are perfect, and failure rate improves continuously in time. It also accounts for the learning period that testers go through as they become familiar with the software.

Schneidewind's Model—The basic philosophy of this model is that the failure detection process changes as testing progresses, and that recent failure counts are of more consequence than earlier counts in predicting the future. This means that more weight should be given to the most recent intervals, where the failure rate has dropped, when estimating and predicting reliability.

Thompson and Chelson's Bayesian Reliability Model—This model assumes that a given software program might be failure-free (an infinite mean time between failures, or MTBF). It also assumes that the software redesign and repair after failures in a given test phase will be conducted at the end of the testing intervals in which the failures occurred.

Trivedi and Shooman's Many State Markov Models—This model is used in providing estimates of the reliability and availability of a software program based upon a failure detection and correction process. Availability is defined as the probability that the program is operational at a specified time.

2.2.1.2 Data Domain Models

Data domain models estimate reliability by using the number of successful runs vs. the total number of testing runs completed during specified intervals. These models estimate the average reliability and the number of remaining faults.

Nelson Model—This model combines the results of a given set of runs and tries to take into account the input values needed to execute the program.

LaPadula's Reliability Growth Model—The approach is to fit, using least squares, a reliability curve through the success/failure counts observed at various stages of the software testing.

2.2.1.3 Fault Seeding Models

This approach requires programs to be seeded with faults and released to a group for testing. It assumes that the seeded fault distribution is the same as the inherent fault distribution. The estimates of the remaining failures are made based on the number of faults discovered by the test group. It is the opinion of this group that fault seeding models should not be employed because information on the algorithms and their effectiveness are not readily available or proven.

2.3 Tools

There are currently various tools available that address software reliability. The Analysis & Risk Assessment Branch has acquired and used the SMERFS Tools and the SRE Toolkit. SMERFS is used because it contains more models than any other tools mentioned above, and is a public domain utility. The SRE Toolkit is used because it takes CPU time as input and gives a precise display on the plotting results. It is commercially available through AT&T Bell Laboratories. This section will briefly summarize the workings of the SMERFS and SRE tools.

2.3.1 SMERFS Tool

The Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) version 4.0 includes 10 models described below. This tool is a public domain software application that can run on an IBM PC and compatible platform. The tool was developed by the Naval Surface Warfare Center (Dr. William Farr [13]).

SMERFS calculates the reliability of software by using execution time or interval counts and lengths data as input. SMERFS models that use execution time data require software CPU failure times or failure counts as input. Interval counts and lengths data models, on the other hand, require the number of failures that occurred for a particular interval. An interval is defined as an equal time partition of the total testing time (CPU time or calendar time).

The models included in SMERFS are:

Execution Time Data Models

- Littlewood and Verrall's Bayesian Model
- Musa's Basic Execution Time Model
- Musa's Logarithmic Poisson Execution Time Model
- Geometric Model
- Non-homogeneous Poisson Model for execution time data

Interval Counts and Lengths Data Models

- Generalized Poisson Model
- Non-Homogeneous Poisson Model for Interval Data
- Brooks and Motley's Discrete Model
- Schneidewind's Model
- S-Shaped Reliability Growth Model

Refer to the SMERFS User's Guide [13] for assumptions to each model.

Specifically, SMERFS is divided into the following options:

Access and Activation Option—An option exists for an optional history file that saves all screen output for later review. A plot file that stores all graphical output data as text is also available.

Input Option—SMERFS allows the user two types of input methods for data. Data may be entered manually by keyboard or automatically through file insertion. SMERFS has the capability to load a maximum of 1000 entries.

Editing Option—The edit option permits altering of the software failure data entries in case editing is needed.

Transformations Option—This option permits the scaling of a software failure data entries. Five types of transformations are allowed along with the options of restoring the data entries to it's original state. The basic purpose of this module is to transform the data by multiplication or addition into a refined and transformed set of numbers.

General Statistics Option—This option provides general summary statistics of the software failure data. Statistics such as the median of the data, the upper and lower bounds, the maximum and minimum values, the average, standard deviation, sample variance, skewness, kurtosis, number of failures found, and total testing time are reported.

Raw Data Plots Option—This option generates plots of the software failure data. The raw data is the original input of the software failure data. This module will also provide smoothing if the sample size contains more than six failure entries.

Execution Time Data Option—This option executes the five execution time data models.

Interval Count and Lengths Data Option—This option executes the five interval count and lengths data models.

Analysis of Model Fit Option—This option allows the plot of the raw and predicted values for the last model executed to be displayed and saved. The prediction must be run, and the future values stored in memory so that a plot can be created. Residual plots and goodness-of-fit statistics are also provided within this subroutine.

Termination and Follow-Up Option—Normal termination will write and close any open or recently created files.

2.3.2 SRE Toolkit

The SRE Toolkit, developed by AT&T Bell Laboratory, consists of two models: the Musa Exponential Time model and the Musa Logarithmic Time model. It is written in "C" and can be run on IBM PC and compatibles. The current version is 3.10A, published on April 5, 1991. The SRE Toolkit uses only CPU execution data as input; therefore, the failure occurrence time has to be precisely recorded by the computer during testing.

The general assumptions of the Musa models are:

- The software is operated in a similar manner as anticipated operational usage.
- The CPU execution times between failures are piece-wise exponentially or logarithmically distributed.
- The hazard rate is proportional to the number of failures remaining in the program.
- Failure intensity decreases with failures experienced and corrected.

The SRE Toolkit is not a menu-driven software. When the toolkit is run, all options have to be entered into an ASCII file, the .fp file. The .fp file contains user's choice of model type, failure intensity objective, and confidence level. Section 2.5.2.2 will describe in detail the .fp file.

2.4 Data Collection and Analysis

The input data to software reliability modeling is software test data software failure occurrences, test time, and resources. The required data has to be defined, and a collection procedure needs to be established prior to the start of the testing.

For any analysis efforts conducted on projects already started, concluded, or missing a data collection procedure, a collection of failure reports of the test can be used for modeling using the interval and counts lengths models (see section 2.3.1).

2.4.1 Descriptions of Data Required

2.4.1.1 Test Time Data

The most accurate test time data is recorded by the computer. The test time data required consists of time that a test run starts and stops. Any messages associated with the test failures should usually accompany the execution time data.

2.4.1.2 Resource Data

This data contains the manpower and computer resources allocated for the software test. Manpower data consists of the number of hours that testers and developers spent to correct software failures. Computer resource data consists of total computer user wall clock time reported. Both manpower and computer resource data will be used as parameters input to the Calendar Time Model which is embedded in the SRE tool. This model will convert the estimation and prediction results from CPU execution time to calendar time, such as testing days remaining to reach the reliability objective and completion date.

Table 9 is a list of manpower and computer resource parameters. The parameters are stored in an ASCII file (the .fp file). Data types and ranges of the parameters are also described in the table.

Table 9 - Manpower and Computer Resource Parameters

Parameter Name	Description	Type	Value Range
muc	computing resources expended per failure	real	0 - 999
muf	correction work expended failure	real	0 - 999
mui	identification work expended failure	real	0 - 999
pi	identification personnel available	real	1 - 999
pc	available computer time in prescribed work periods)	real	0.01 - 9999
pf	correction personnel available	real	1 - 999
rhoi	utilization of identification personnel	real	0.001 -1
rhoc	computer utilization factor	real	0.001 -1
rhof	utilization of correction personnel	real	0.001 -1
thetai	identification work expended per CPU hour	real	0 - 99999
thetac	computer time expended per CPU hour	real	0 - 9999
thetaf	correction work expended per CPU hour	real	0 - 99999
workda	average hours per workday default = 8	real	0.1 - 24 default = 8 hrs
workwk	average workdays per week default = 5	real	0.1 - 7 default - 5 days

2.4.1.3 Test Failures Reports

Trouble Report (TR) or Discrepancy Report (DR) contains a description of the failure, failure occurrence date, severity level of the failure, and corrective action. TRs or DRs related to a failure during a software test are generated by testers, and the corrective actions can be provided by either designers or testers. The information found in the TRs or DRs could be very helpful in the event that computer-automated failure information is not available or complete.

2.4.1.4 Test Script

Test script is a computer file written by tester in a specific language that specifies the software test sequence of a specific test case. Test script can help an analyst validate a software failure, specifically when a test case is designed to intentionally fail the software.

2.4.1.5 Software Source Code

This is the computer program which is usually stored in ASCII format. In addition to the source code analysis which is discussed in Section 1, Software Complexity Analysis and the Determination of Software Test Coverage, source code can be used to identify which part of the code contains the software fault that caused a software failure during a test run. This, in turn, will be used to approximate the time that the software failure occurs.

2.4.2 Data Collection Procedure

Before gathering and evaluating failure data, the software test process needs to be considered for appropriateness to the modeling assumptions. Software reliability works best on the software code which is ready to be integrated. The reliability measured during testing should closely resemble the actual reliability of the software during operation. Failures obtained during unit testing are not recommended for use in modeling.

The most appropriate level of testing to apply to software reliability includes the software integration test for multiple computer software configuration items (CSCIs), the software user acceptance test, and a system test which combines software and hardware.

This section provides the step-by-step procedure of how to obtain data for the modeling process. Before testing, the analyst should coordinate with the testing group to set a routine time for data collection to ensure that data are promptly collected. The frequency of data collection depends on the project, number of tests, number of testers, and the amount of data accumulated. The data obtained may include the automated test logs, tester's resource data, trouble reports, discrepancy reports, software source code, and test scripts. The software source code and test scripts should be available before testing starts. Tester's resource data should be available when testing starts.

Step 1: Transfer raw data from testing group to the analysis group.

This step describes the procedures of downloading files from VAX system located at the off-site tester to the JSC Engineering VAX Laboratory, then transferring to the SR&QA facilities. Currently, there is no direct transfer from the off-site tester to SR&QA facilities. The downloading procedures will transfer the computer-automated test files. However, there is information that is provided by the testers manually, such as the failure reports (TRs, DRs) and the testing resource data. Therefore, this section will also discuss the hard copy data acquisition.

(1) Download files from tester directory on the VAX to your directory on the Avionics Software Development (ASD) VAX.

   a) Copy all files into the designated tester's specified VAX account
   b) Make a subdirectory using the current day's date
   c) Move these files into the above subdirectory
   d) Rename all files before moving them over to the ASD VAX (ASD VAX will save only five versions of a given file)
   e) Copy all files to the analyst's ASD VAX account
   f) Log on to the analyst's ASD VAX account and verify that all files were transferred correctly (you should get a message about which files were not transferred; just copy those files again)

g) Create a new subdirectory using the current day's date
h) Move these files into the new subdirectory
i) Delete files that were on the root directory
j) Go back to the tester's disk and delete the .log files, and old files from the directory. First purge, then delete any remaining files, except the .com file and the .dat file.

(2) Download files from ASD VAX to UNIX system.

a) Log on to the UNIX system
b) Remote log on to NASA VAX system
c) Create a directory in UNIX (directory name should be the current date)
d) Change to current directory
e) Transfer files
f) Enter VAX ID and password
g) Quit
h) Verify that all files were transferred
I) Rename all files

(3) Download files from UNIX system to IBM floppy disk.

a) Insert IBM diskette to disk drive
b) Log on to UNIX
c) Change to the directory that you want to copy the files from
d) Copy all files to a directory
e) Verify that all files are copied
f) Eject the diskette
g) Log off

(4) Hard copy data.

If the data is logged by testers manually on paper, check the data for any entry errors, missing data, or questionable entries. The tester should be contacted for any missing or questionable data.

Step 2: Input raw data into the database.

Note: In the case of JSC SR&QA's practice, a database named SWREL_DB with instructions is set up on the LAN in Shared Directory using R:BASE for DOS. The SMERFS, SRE, and failure reports are already built in to this database.

2.4.3 Data Analysis Procedure

The main purposes of evaluating the raw data are to:

- Determine the CPU time that the computer spends executing test cases.
- Determine the exact CPU time when a failure is triggered during the execution of a test case.
- Eliminate all duplicate executions of a particular test case due to non-software failures problems such as hardware failures, communication problems, or test script re-writes.

- Identify and eliminate failures that are dependent on previous failures. Only independent failures are applicable for software reliability models.
- Identify any differences in the testing machine CPU speeds and normalize the execution time with respect to the slowest CPU. Normalization can be done by multiplying execution time of a test run by a conversion factor that is determined by comparison of the speeds of the different processors.

The data analysis process could be very time-consuming. A good understanding of the software functions, structures, and test cases will help to ease the process.

The following procedures are based on the assumption that all the recommended data are available.

2.4.3.1 Analysis of Test Plan and Test Procedure

Review the test plan to ensure that the data collection procedure is included, and that the collection procedure will allow access to all necessary data.

Review the test procedure to identify the objective of each individual test case. Examine test scripts to identify the expected output. Description of the test cases and test scripts will help the analyst identify any failure that occurs during the test, and to verify if any test data is missing.

2.4.3.2 Test Data Analysis

Step 1: Examine the computer automated test log (i.e. Error History Logs) to determine the start time, end time, and failure time of the software test run.

Step 2: Determine any duplicate run in the computer log and discard.

Step 3: Identify software failures in the computer logs (if possible). Discard hardware-related failures.

Step 4: Determine and discard any duplicate or dependent software failures. Keep a list of the assumptions used for interpreting a dependent failure to be consistent throughout the analysis

Step 5: Examine tester's log to identify any test cases that don't appear in the computer log, and to identify any software failure that was logged by testers but were not identified in the computer logs.

Step 6: Update the database.

2.4.3.3 Failure Reports Analysis

The TRs or DRs are usually submitted by testers or developers to report significant problems after the test run. We recommend cross referencing between the TRs or DRs and the automated computer logs to enhance the data analysis process.

Step 1: Set up criteria to categorize failure criticality.

Step 2: Determine if the failure is a software-related failure.

Step 3: Determine if the failure is independent of previous failures.

Step 4: Categorize failure based on criticality level.

Step 5: Cross reference with failures identified in computer log and tester's log and determine any discrepancy. Communicate with testers or developer to clarify any discrepancies found.

2.4.3.4 Example of Data Analysis

The following data was collected from a software test project. The data was automatically logged by computer in an ACII file every time a software test run executed. The data includes a tag which indicates the time when a certain event occurred, such as test start time, failure occurrence time, and test end time.

```
07:39:12.29 INFORMATIVE Message from SES Parser ==> Test Configuration is
   INTEGRATED
07:44:52.60 WARNING Message from Dkmc.BUFFER_SERVICES  ==> PATCH Command
   Address_Conversion.  Process_Not_Located exception occurred!  Raising
   Constraint_Error!
07:44:52.63 WARNING Message from DKMC_CP/IF ==> Exception Occurred!!
07:44:52.63 WARNING Message from DKMC_CP/IF ==> Exception occurred on PATCH
   Command Sddu#= 5 Log_Id= 1
07:44:52.64 INFORMATIVE Message from DKMC_CP/IF ==> Received CMPR Command
   => END_BLOCK
07:44:52.64 INFORMATIVE Message from Dkmc.TEST_CMD_EXEC  ==> END_BLOCK Com-
   mand completed.
```

Preliminary analysis of the data above indicates that the software run starts at 07:39:12.29. At 07:44:52.60 a software failure is observed. All other failures reported afterward appeared to be dependent on the first reported failure. Thus we have only one software failure in this case. The software test run was terminated at 07:44:52.64.

2.5 Modeling Procedure and Analysis

Once all the data has been collected and analyzed for applicability, specific models need to be chosen to estimate and predict reliability. The selection criteria used to choose a model are the data input, the nature of the software, and the test level needed to comply with the various assumptions found within each model. The current tools available are the SMERFS tool and the SRE Toolkit. Both tools have specific models that may be used. The SRE model assumptions are listed in section 2.3.2, while the various models assumptions in SMERFS are described in the SMERFS User's Guide [13]. The SMERFS tool also lists the model assumptions

on screen and in the output ASCII file. Once the model assumptions are satisfied and the tool chosen, proceed to convert the data stored in the database to input files that the tools will accept. The following sections describe the procedure of database conversion and model plotting.

2.5.1 SMERFS

The SMERFS tool runs on an IBM PC or compatible systems. DOS 3.0 or above is required. The procedures outlined below assume that R:BASE was used as the repository of the failure data described in section 2.4.0. The procedures can easily be adapted for use with any database desired.

2.5.1.1 Data Conversion

SMERFS has two data input options. Manual input should be used for small amounts of failure data (20 lines or less). If the failure data collected is greater than 20 lines, however, then the file insertion method should be used. Data file editing is necessary before SMERFS can read the data produced by the database.

To transfer the data into the SMERFS program without resorting to manual input, do the following:

Step 1: R:BASE Output

There exists a report type within the R:BASE file SWRDB.DBF (this database can be obtained from the Analysis & Risk Assessment group) that is called *smerep*. While in the command line mode within R:BASE, direct the output from the screen to a file using the following set of commands:

R>out *filename.dat*	(redirect output to file)
R>print *smerep* sorted by *option1 option2*	(output sorted report)
R>out screen	(redirect output to screen)

where:

filename.dat is the output file name that R:BASE will write the sorted failure data to. The output file contains headings and data pertaining to time between failures, accumulated non-failure time, and start and stop times.

Smerep is the report used to create the sorted failure data file (*filename.dat*).

option1 and *option2* are R:BASE (database) defined columns that contain the dates and CPU times of all the test jobs.

The following example shows the contents of the R:BASE *smerep* report output (this file is automatically made by *smerep*).

SMERFS REPORT FOR XXXXXX

Date 0X/27/9X

Page 1

F#	TBE1	TBE2	TBE3	TBE4	TBE5	TBE6	TBE7	TBE8	ET_ST	TBD1	TBD2	TBD3	TBD4	TBD5	TBD6	TBD7	TBD8
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	252.695847	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	169.289017	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	32.4508667	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.5410614	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.6234589	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.1565399	0.	0.	0.	0.	0.	0.	0.	0.

Where:

F#: The number of failures accumulated for one job.

TBEx: The time between the previous failure or start of job, and the current failure. (**Note:** The SMERFS program uses the term TBE to represent time between failures, also commonly referred to as TBF. When referring to the SMERFS program only, these are interchangeable. In other cases, TBE and TBF have different definitions.)

ET_ST: The time between the end and the start of the job.

TBDx: The time between the end of the job and the current failure.

Step 2: Editing

Once the R:BASE output data file (filename.dat) has been created, delete any heading and blank lines within this file (this is done because the conversion program discussed in step 3 will halt if any of these are present). In addition, any special characters or printer commands need to be removed. The next example shows the R:BASE smerep Report once the editing has been accomplished.

-0-	-0-	0.	0.	0.	0.	0.	0.	0.	252.695847	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	169.289017	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	32.4508667	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.5410614	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.6234589	0.	0.	0.	0.	0.	0.	0.	0.
-0-	-0-	0.	0.	0.	0.	0.	0.	0.	31.1565399	0.	0.	0.	0.	0.	0.	0.	0.

Step 3: Basic Conversion Program

This program will convert an R:BASE sorted failure file into a SMERFS input file. Note: the file contains only data (no headings or blank lines are acceptable—see step 2). The conversion program (SMERFS.BAS) is written in GWBASIC format. Its input is the edited file output of step 2 of this section. Comments and instructions are included in the file SMERFS.BAS.

Run SMERFS.BAS from any machine that has GWBASIC loaded (generally all machines with DOS 5.0).

SMERFS.BAS will ask for the edited file output name from step 2 of this section. The program will echo the input file. It will then ask for the output file names for the time between failures and interval files (they may not be the same). The last input asked will be the interval time in hours. This interval time will be used to partition the total execution time into equivalent lengths of the desired time (as used by the interval and count lengths models).

Execution is automatic after the interval time in hours is entered by the user.

The SMERFS.BAS program will output two user-specified files. One will be used with the CPU TBF models, and the other with the interval models. These output files will be used as the input files to the SMERFS modeling application.

2.5.1.2 SMERFS Modeling Procedure

This section will describe the actual input, execution, and output procedures for the SMERFS application. Further information may be obtained from the SMERFS manual (see reference [13]).

2.5.1.2.1 SMERFS Inputs

SMERFS allows a maximum of 1000 data point entries for each of the following data types:

1) Wall Clock Data
This data type consists of the starting and ending times of testing (24-hour units - 13:23:45 = 13.39583), the number of failures for that session, and the time of any software failures that occurred within that session (24-hour units).

2) CPU Data
This data type consists of the expended CPU time (in seconds only) and a flag of 0 or 1. A flag of 1 indicates that the testing session was terminated with a failure; a 0 indicates that a failure was not detected during the testing session.

The time between failures input file, as shown in the example below, contains the number of entries that will contain time between failures data, the failure state of the last time between failures data entry, and the time between failures (reported in seconds). The number of entries that will be included is reflected as the first number of the first line in the table. The failure state flag is reflected as the second number of the first line in the table. This flag indicates whether the last entry in the table is a failure (1) or additional testing time that had no failure (0). The entries found within the rest of the input file reflects the CPU failure execution time between failures collected (in line 2 of the example below - 15 seconds elapsed before the first failure occurred). SMERFS requires that all CPU data input files contain 1000 data entries. Therefore, it is necessary to fill the unused entries with zeros. The example shows only the first 15 lines of the SMERFS CPU data file (total of 1001 lines exist in the actual file).

```
      15         1
.15000000E+02
.23000000E+02
.20000000E+02
.28000000E+02
.35000000E+02
.37000000E+02
.43000000E+02
.52000000E+02
.49000000E+02
.58000000E+02
.67000000E+02
.62000000E+02
.76000000E+02
.82000000E+02
.10100000E+03
```

3) Wall Clock and CPU Data

This data type consists of the starting and ending times of testing (in 24-hour units) and the number of failures detected in that testing session. If any software failures were detected in that session, SMERFS prompts for the failure time (in seconds) for each failure. If the last failure occurred before the end of the testing session, SMERFS prompts for the expended CPU time between the last failure occurrence and the end of the testing session. If no software failures were detected in the session, SMERFS prompts for the entire CPU time expenditure for the session.

4) Interval Data

This data type consists of the number of failures detected in a testing interval and the associated testing length of that interval. An interval is defined as a user-selected length of time that equally divides the total testing time of the software. The use of interval data is beneficial if exact failure times are not available (total testing time is still critical).

The interval input file to SMERFS, as shown in the next example, contains the number of entries that will be used as intervals for modeling, the interval data, and the interval length. The first number on the first line depicts the number of entries in the file that will contain data. The second number of the first line is not used. The next 1000 entries (only the first 15 entries before the dashed line are shown on the example) will contain failure data as found within each respective interval (the example below shows that there are 15 intervals that correspond to the first 15 entries that contain failure data). The data in each entry represents the number of failures found within that particular interval. The next 1000 entries (only the first 15 entries after the dashed line are shown on the example) contain the length of each interval (this is by default 1 for SMERFS). The total length of this file should be 2001 lines (1 for the first two numbers, 1000 for entry data, and 1000 for interval lengths of the first 1000 entries).

```
   15          0
.35000000E+02
.38000000E+02
.27000000E+02
.21000000E+02
.18000000E+02
.19000000E+02
.14000000E+02
.20000000E+02
.90000000E+01
.40000000E+01
.00000000E+00
.20000000E+01
.10000000E+01
.00000000E+00
.20000000E+01
--------------------
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
.10000000E+01
```

This handbook will only cover CPU and interval data procedures.

2.5.1.2.2 Output

An optional user-selected history file (see Appendix B for an example) that saves all screen output for later review contains all the user interactions with the SMERFS application as seen on the screen. A plot file (see Appendix B for an example) that stores all graphical output data as text is also available if selected by the user. This file contains all of the graphical output provided by SMERFS in text form. This file may be used to create plots of the data using other applications (such as Lotus 1-2-3, Harvard Graphics, and Microsoft EXCEL).

2.5.1.2.3 Execution Procedure

This section contains a stepwise description of a typical SMERFS modeling session. There is an optional history file that saves all screen output for later review. A plot file is also optional.

The plot file will save all the SMERFS-created plot data onto a text file. An example session can be found within the SMERFS User's Guide [13].

Step 1: Check that all of the SMERFS executables are within the same directory as the data files are.

Step 2: Prepare the SMERFS input file.

Step 3: Run SMERFS by typing "SMERFSIV" at the prompt.

Step 4: Provide file names for the history and plot files as applicable.

An option exists for an optional history file that saves all screen output of the SMERFS session for later review. A plot file that stores all graphical output data as text is also available.

Step 5: Enter "<smf4rdc.lis>" when prompted with the file name of the SMERFS model assumptions and data requirements file.

Note: The file <smf4rdc.lis> contains all the model assumptions in text form. These are exclusively used by SMERFS and should not be altered. This file is accessed by SMERFS whenever there is a user request for a listing of assumptions.

Step 6: Enter the appropriate data type selection at the prompt.

Step 7: Choose file or keyboard input at the next prompt.

SMERFS allows for two methods of data input. Data may be entered manually by keyboard or automatically through file insertion. The input files may be created by the SMERFS editor when the keyboard option is chosen. Input files created with another utility can be used if such files mimic the structure of the SMERFS created files (see 2.5.1.2).

Step 8: Enter a "1" or a "0" to list or skip the listing at the list prompt.

Step 9: Enter a "1" at the next prompt only if a keyboard input was chosen at step 7, else, enter a "0."

Step 10: Enter a "0" to view a list of the main module options, or skip to step 11.

Step 11: Enter the desired option from the main module options.

1 - data needs to be input

2 - data editing is needed

The edit subroutine permits altering of the software failure data. All data updates should be performed using the same units of measurements. Editing choices are:

- Change specified element
- Delete specified element(s)
- Insert up to ten elements

- Combine two or more adjacent elements
- Change the Time Between Failure (TBE) flag
- List the current data

3 - data transformations are needed

This option permits the scaling of software failure data entries. Five types of transformations are allowed along with the options of restoring the data entries to their original state. This transformation operates on one data entry at a time. Two time between error entries cannot be transformed together, nor can the interval counts and lengths. The transformed data is held locally. If any change is desired, it must be done in the edit module.

The purpose of this subroutine is to transform the data by multiplication or addition into a refined and transformed set of numbers. The data will still contain the basic characteristics of the original data. The difference, however, will be in the presentation of the data (in graphical form). The output data may appear smoothed, expanded, multiplied, and scaled. See the SMERFS Users Guide [13] for further information on execution flow.

4 - data statistics are required

This option provides general summary statistics of the software failure data. Refer to SMERFS User's Guide [13] for explanation of the terms.

5 - create plots of the raw data

This option generates plots of the software failure data. The raw data are the original inputs of the software failure data. This module will also provide smoothing if the sample size is greater than 6 failure data entries.

6 - module execution

This option allows the execution of the models.

7 - analyses of the model fit

This option allows the plotting of the raw and predicted plots for the last model executed. The model must be run, and the future values from the prediction stored in memory so that they may be plotted. Plots may be directed to an outside file when quitting this module. This module also provides residual plots. A residual plot is the difference between the predicted and "raw" data. The plot may be used to determine the validity of the data. A goodness-of-fit option is also available. This option provides the chi-square goodness-of-fit statistic for the predicted interval failure counts only.

8 - termination of SMERFS

Step 12: Repeat step 11 as necessary.

Step 13: End session by choosing option 8 from the main SMERFS menu.

2.5.1.3 Results Interpretation

This section will discuss the estimation plot results of the Littlewood & Verral Bayesian Model and the Schneidewind Model. Other models will be added in the future.

Littlewood and Verral Bayesian Model

Figure 6 shows the estimation failure results obtained from properly developed and debugged software code (this code is referred to as "mature"). The chart shows the actual and fitted estimation failure data. The actual and estimated data should approximate an asymptote parallel to the Y axis on some X axis point (failure). As the estimated curve reaches the asymptote, the time between failures (as read from the Y axis) is increasing with each consecutive failure. The degree to which this convergence occurs also points to code that shows a decrease in the failure rate (defined as the inverse of the time between failures).

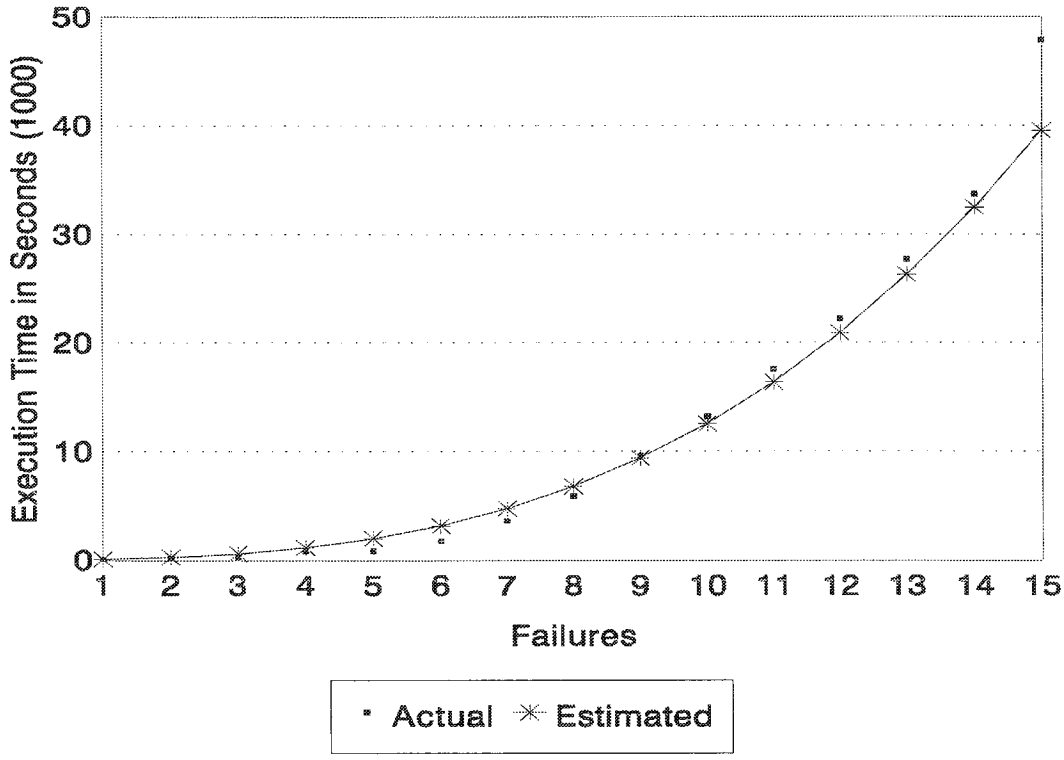


Figure 6 - Littlewood & Verral failure estimation.

If the estimated failure data curve shows a linear or does not converge on an asymptote (as shown in figure 7) then there exist problems with the test data gathered, a violation of the model assumptions, or the software under test is not mature. If the estimation failure curve is linear (as shown as Estimated-1 on figure 7), the failure rate is constant with time (the time between failures is relatively equal from failure to failure). This case shows that more testing and debugging are needed before this code shows a decreasing failure rate.

In the case of non-convergence (shown as Estimated-2 on figure 7), the failure rate is increasing with time (the time between failures is decreasing with each new failure). In this case, new failures are being introduced into the code. This can occur if new software is added, the testing is not truly random, or the debugging is introducing failures.

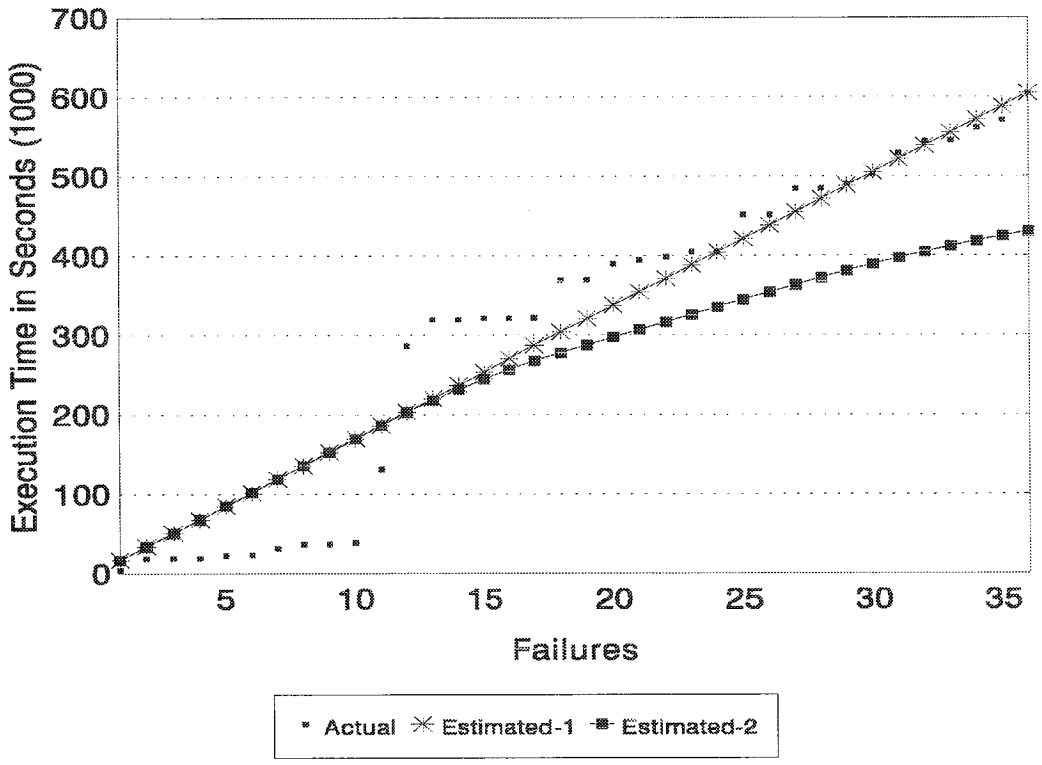


Figure 7 - Littlewood & Verral failure estimation.

Schneidewind Model

Figure 8 shows the estimation failure results obtained from properly developed and debugged software code (this code is referred to as "mature"). The chart shows the actual and fitted estimation failure data. The actual and estimated data should converge on an asymptote parallel to the X axis on some Y axis point (failure number). As the estimated curve reaches the asymptote, the time between failures (as read from the X axis) is increasing with each consecutive failure. The degree to which this convergence occurs also points to code that shows a decrease in the failure rate (defined as the inverse of the time between failures).

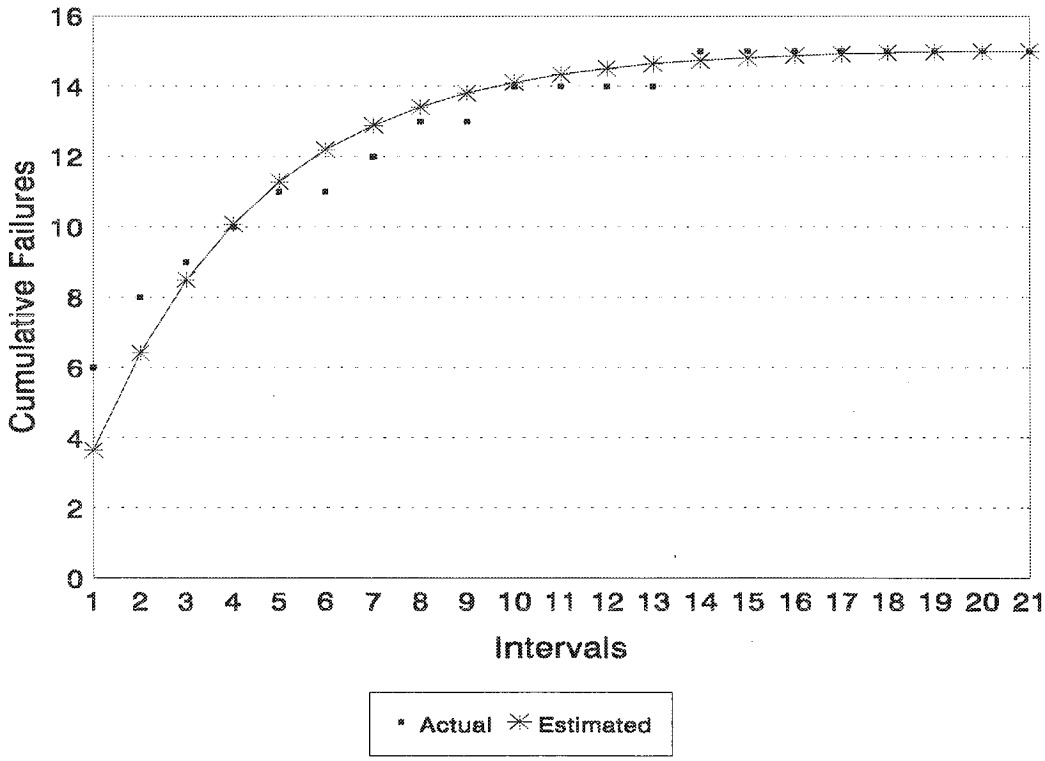


Figure 8 - Schneidewind Method 1 cumulative and predicted failures - 1.

If the data is linear or does not converge on an asymptote (as shown in figure 9) then there could be problems with the test data gathered, violation of the model assumptions, or the software under test is not mature. If the estimation failure curve is linear (as shown as Estimated-1 on figure 9), the failure rate is constant with time (the time between failures is constant). This case shows that more testing and debugging are needed before this code shows a decreasing failure rate.

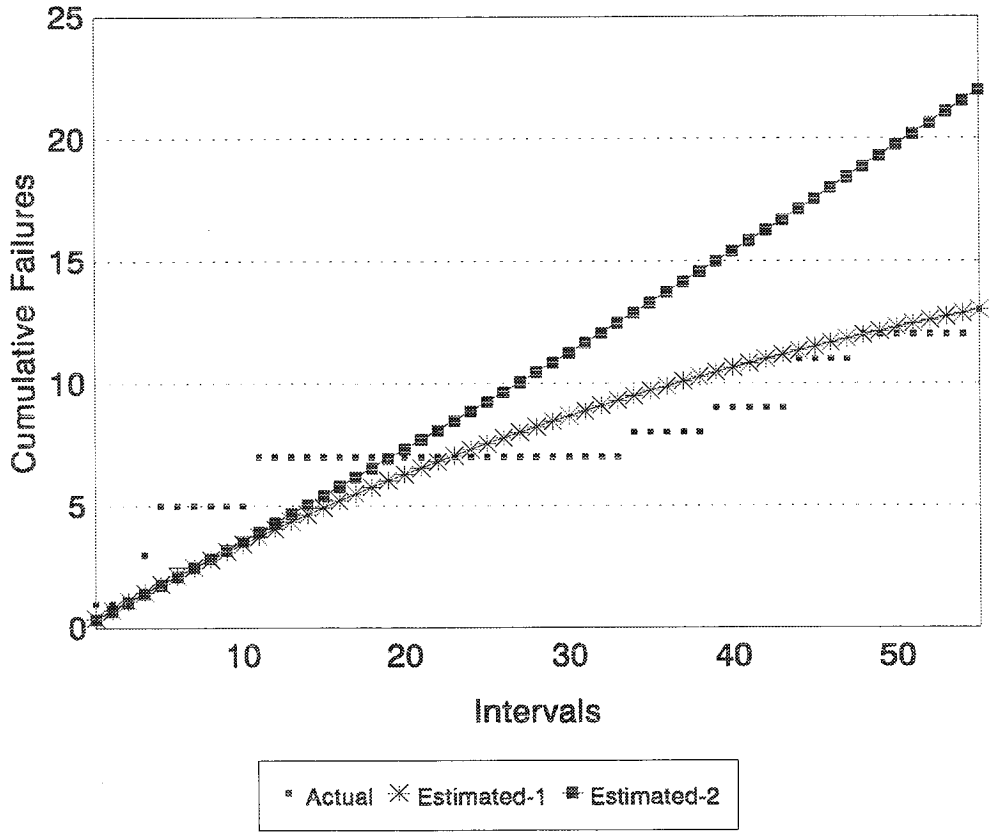


Figure 9 - Schneidewind Method 1 cumulative and predicted failures - 2.

In the case of non-convergence (shown as Estimated-2 on figure 9), the failure rate is increasing with time (the time between failures is decreasing). In this case, new failures are being introduced into the code. This can occur if new software is added, the testing is not truly random, or the debugging is introducing failures.

2.5.2 SRE

2.5.2.1 Data Conversion

To transfer the data into the SRE Toolkit without resorting to manual input:

Step 1: R:BASE Output

There exists a report type within the R:BASE file SWRDB.DBF (this database can be obtained from the Analysis & Risk Assessment group) that is called *srerep*. While in the command line mode within R:BASE, direct the output from the screen to a file using the following set of commands:

R>out *filename.dat*	(redirect output to file)
R>print *srerep* sorted by *option1 option2*	(output sorted report)
R>out screen	(redirect output to screen)

where:

filename.dat is the output file name that R:BASE will write the sorted failure data to. The output file contains headings and data pertaining to time-between-failures, accumulated non-failure time, start, and stop times.

Smerep is the report used to create the sorted failure data file (*filename.dat*).

option1 and *option2* are R:BASE (database) defined columns that contain the dates and CPU times of all the test jobs.

The following example shows the contents of the R:BASE *smerep* report output (this file is automatically made by *smerep*).

SRE REPORT FOR XXXXX

Date XX/XX/XX

Page 1

Testdate	Starttime	Endtime	Fail#	FT1	FT2	FT3	FT4	FT5	FT6	FT7	FT8	FT9	F10
930802	14.62694	14.68184	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.73194	14.77539	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.82111	14.82953	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.84306	14.85128	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.86667	14.87463	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.88722	14.89568	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.91361	14.92176	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.94028	14.94863	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.96528	14.96546	1	14.96546	0	0	0	0	0	0	0	0	0
930802	16.01583	16.07115	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.1525	16.19469	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.2625	16.27051	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.28611	16.29364	-0-	-0-	0	0	0	0	0	0	0	0	0

930802	16.30611	16.3138	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.32667	16.33449	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.35	16.35788	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.37306	16.38095	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.63139	16.67976	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	8.890833	8.894263	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	9.225	9.288181	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	9.261666	9.314458	-0-	-0-	0	0	0	0	0	0	0	0	0

Where:

Testdate: The date tag of the job specified as year, month, date.

startime: The start time of the job in hours.

Endtime: The end time of the job in hours.

Fail#: The number of failures in the job.

FT1-FT10: The actual failure times for each failure.

Step 2: Editing

Once the R:BASE output data file (*filename.dat*) has been created, delete any heading and blank lines within this file (this is done because the conversion program discussed in step 3 will halt if any of these are present). In addition, any special characters or printer commands need to be removed.

930802	14.62694	14.68184	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.73194	14.77539	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.82111	14.82953	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.84306	14.85128	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.86667	14.87463	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.88722	14.89568	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.91361	14.92176	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.94028	14.94863	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	14.96528	14.96546	1	14.96546	0	0	0	0	0	0	0	0	0
930802	16.01583	16.07115	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.1525	16.19469	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.2625	16.27051	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.28611	16.29364	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.30611	16.3138	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.32667	16.33449	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.35	16.35788	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.37306	16.38095	-0-	-0-	0	0	0	0	0	0	0	0	0
930802	16.63139	16.67976	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	8.890833	8.894263	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	9.225	9.288181	-0-	-0-	0	0	0	0	0	0	0	0	0
930803	9.261666	9.314458	-0-	-0-	0	0	0	0	0	0	0	0	0

Step 3: Basic Conversion Program

This program will convert an R:BASE sorted failure file into an SRE Toolkit input file. Note: the file contains only data (no headings or blank lines are acceptable—see step 2). The conversion program (SRE.BAS) is written in GWBASIC format. Its input is the edited file output of step 2 of this section. Comments and instructions are included in the file SRE.BAS.

Run SRE.BAS from any machine that has GWBASIC loaded (generally all machines with DOS 5.0).

Input to the SRE.BAS is the edited output file. The program will echo the file. It will then ask for the output file name to direct the output.

Execution is automatic after the output file name is entered by the user.

The SRE.BAS program will output one user-specified file. This output file is the input file for the SRE Toolkit.

2.5.2.2 SRE Modeling Procedure

SRE Toolkit runs on an IBM PC or compatible. DOS 3.0 or above is needed when SRE Toolkit runs in IBM PC or compatible. Modify the autoexec.bat file to have a direct path toward the SRETOOLS directory.

2.5.2.2.1 Input

To run the SRE tool, two input files are required: the .ft file which contains the execution time and the failure time, and the .fp file which contains the parameters which specify reliability objective, type of model used, plot options, and labor and computer resource parameters. The two files are described separately in the following sections.

.ft file

From the database, generate a text file with extension .ft, using any text editor, with the following format:

A dd.dddd YYMMDD

where :

A is an alphabet which represents either "S", "F", or "E" as follows:

S: time the test case execution starts
F: time a software failure occurs
E: time the test case execution ends

dd.ddd is the military time of the day when the event (Start, Fail, or End) occurs.

YYMMDD is year, month, and date when the test case is run.

Here is an example of execution time data as input file to the SRE:

```
S 12.13000 920827
F 12.16433 920827
F 12.18968 920827
F 12.19858 920827
F 12.36725 920827
F 12.36825 920827
E 12.36925 920827

S 12.58194 920828
E 12.82595 920828

S  9.88194 920829
F  9.89000 920829
E 10.15405 920829

S 10.28222 920829
E 10.52841 920829
```

.fp file

This file is used in conjunction with the .ft file. Both files should be stored in the same directory. The .fp file is used to indicate what option the user wishes to select for the modeling, such as model type (logarithmic, exponential), failure intensity objective, computer resource, labor resource, and type of output plots to be displayed on screen.

Descriptions of the parameters used for the .fp file, parameter names, descriptions, data types, and ranges, are listed in table 10.

Table 10 - List of Parameters for the .fp File

Parameter Name	Description	Type	Value Range
model	0 - exponential 1 - logarithmic default value = 0	integer	0 - 1
grp data	group data 0 - failure time analysis 1 - group data analysis default value = 0	integer	0 - 1
ttl	title for output report	string	none
compf	test compressions factor default value = 1	real	0.1 - 99999
lambf	failure intensity objective	real	0 - 999
adjflg	adjustment flat 0 - no adjustment 1 - failure time adjustment default value = 0	integer	0 - 1
genplt	generate plot commands 0 - no plot 1 - yes default value = 0	integer	0 - 1
conlvl	confidence level 0 - no confidence 1 - 95% conf. limit 2 - 95% conf. limit 3 - 75% conf. limit 4 - 50% conf. limit default value = 3	integer	0 - 4
plot	"y versus x"	string	none

SRE has the capability to generate many different plots, depending on what plots the user wishes to obtain. To obtain a plot from an SRE run, the user needs to insert a line in the .fp file which has the following format:

plot = " Y versus X "

where Y and X are defined as follows:

Y	X
failures	failures
calendar time	calendar time
execution time	execution time
completion date	present data
initial intensity	
present intensity	
total failures	
failure decay rate	
additional failures	
additional execution time	
additional calendar time	

The format and description of all parameters used in this file are described below:

Example: parameters data

```
# model type and objective parameters
model = 0       lambf = 20

# calendar time parameters
muc = 4.09      muf = 10.0      mui =5.0        workday = 10
pc =10.00       pf = 15         pi = 4          workwk = 6
rhoc = 1.0      rhof = .250     rhoi = 1.
thetac =8.37    thetaf =20.93   thetai =10.46

# plot option parameters
genplt = 1
plot = "failures vs execution time"
plot = "present intensity versus execution time"
plot = "present intensity vs failures"
```

2.5.2.2.2 Output

Output data for the SRE Toolkit consist of a tabular chart and plots that are user-specified.

The Tabular Chart Output

The tabular chart appearing on the screen can be printed on hard copy by simply applying the print screen key or printing it to an ASCII file.

An example of the tabular chart output data is shown in figure 10:

SOFTWARE RELIABILITY ESTIMATION
EXPONENTIAL (BASIC) MODEL
TEST DATA SET

BASED ON SAMPLE OF	15 TEST FAILURES
TEST EXECUTION TIME IS	20.9872 CPU-HR
FAILURE INTENSITY OBJECTIVE IS	20 FAILURES/1000-CPU-HR
CURRENT DATE IN TEST	920910
TIME FROM START OF TEST IS	15 DAYS

	CONF. LIMITS				MOST	CONF. LIMITS			
	95%	90%	75%	50%	LIKELY	50%	75%	90%	95%
TOTAL FAILURES	15	15	15	15	15	15	15	16	16
********** FAILURE INTENSITIES (FAILURES/1000-CPU-HR) **********									
INITIAL	2148	2475	3011	3543	4312	5089	5641	6216	6582
PRESENT	0.660	1.04	2.11	4.14	10.50	25.84	47.65	87.67	127.1
*** ADDITIONAL REQUIREMENTS TO MEET FAILURE INTENSITY OBJECTIVE ***									
FAILURES	0	0	0	0	0	0	0	0	1
TEST EXEC. TIME	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
WORK DAYS	0.0	0.0	0.0	0.0	0.0	0.616	2.49	5.30	7.87
COMPLETION DATE	920910	920910	920910	920910	920910	920911	920915	920918	920921

Figure 10 - Sample SRE Musa exponential time model result.

SRE .pc Output File

This file is generated by the SRE automatically when the user runs the SRE tool with the parameter PLOT set to 1 (plot=1) in the .fp file. The .pc file will provide numerical results of the prediction and the associated statistical data. This will allow users to make plot files, using appropriate application software such as EXCEL or LOTUS 1-2-3. The method of using EXCEL is described in the following section.

Generate Plots

Plots appear on the screen using the PLOT command from the SRE tool. However, SRE can not generate plot files to print plots. To get the hard copies of the plot, the following procedures are recommended using Microsoft Word and Microsoft EXCEL for IBM PC or compatible.

- Using Microsoft Word to replace blanks with the commas in the .pc file.
- Save the file and quit Microsoft Word.
- Use Microsoft EXCEL to load up the modified .pc file with a comma as a delimiter
- Generate plots using graphic features in EXCEL or using appropriate user-created macros.

2.5.2.2.3 Execution Procedures

Step 1: Create two input files in ASCII format, the *filename*.ft and the *filename*.fp, in the same directory.

Step 2: Type "EST *filename*." The computer will prompt the tabular chart on screen which includes the summary of the input data and the estimation/prediction results.

Step 3: Capture the tabular chart onto hard copy.

Step 4: Capture the plots onto hard copies.

2.5.2.3 Results

SRE Toolkit produces two types of results: a tabular chart and plots for each of the two models available in the Toolkit.

Tabular Chart

Musa Exponential Time Model

This model provides the estimation of the failure intensity and the total failure of the software. In addition, the model predicts the number of additional failures needed to be detected to reach the objective failure intensity. If calendar time model parameters are provided to the model, a prediction of how many calendar days needed to complete the test is returned. The results are reported with confidence bounds. See figure 10 for a sample output.

Musa Logarithmic Time Model

This model provides the estimation of the failure intensity and the failure intensity decay parameter, which indicates how fast the failure intensity will drop as the software test progresses. Similar to the Musa exponential model, the logarithmic model predicts the number of additional failures needed to be detected to reach the objective failure intensity and calendar time model prediction. The results are also reported with confidence bounds. Tabular chart for this model is similar to that of the Musa exponential time model. See figure 10 for a sample output.

SRE Plots

Typically, three plots are generated along with the tabular chart for each model. The exponential time model and logarithmic time model results are similar.

Cumulative Failures vs. Execution Time Plot

This plot provides a quick comparison between the predicted curve and the actual curve in terms of cumulative failures occurred during the execution period. The raw data and predicted curves should converge on an asymptote. The degree of convergence gives a measure of maturity. See figure 11 for a sample result of the exponential model.

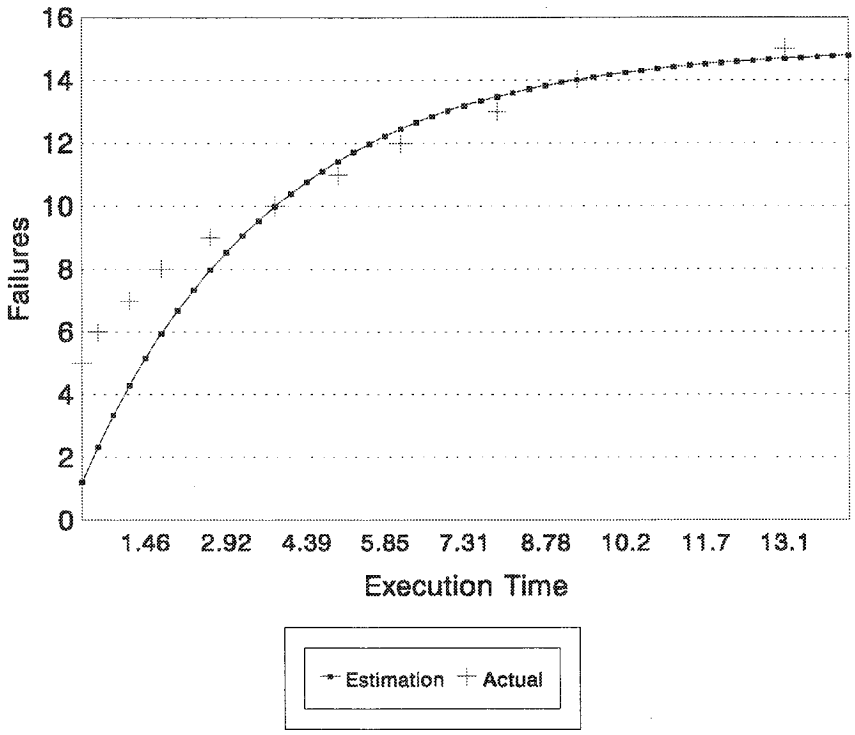


Figure 11 - Sample SRE failure vs. execution time plot.

Failure Intensity vs. Execution Time Plot

This plot provides a visual aid in determining whether the software has reached the failure intensity objective in terms of execution time spent. See figure 12 for a sample result.

The sample result shows that failure intensity of the software was initially very high (10,000 failures per 1000 CPU hours). It then gradually declined as testing progressed. Finally, after 15 hours of CPU execution times, the software reached the failure intensity objective of 10 failures per thousand CPU hours. In the case of inadequate software testing, figure 13 will show that the actual failure intensity line never reaches the failure intensity objective (10 failures per 1000 CPU hours), and, sometimes, the line even diverges upward at the end of testing.

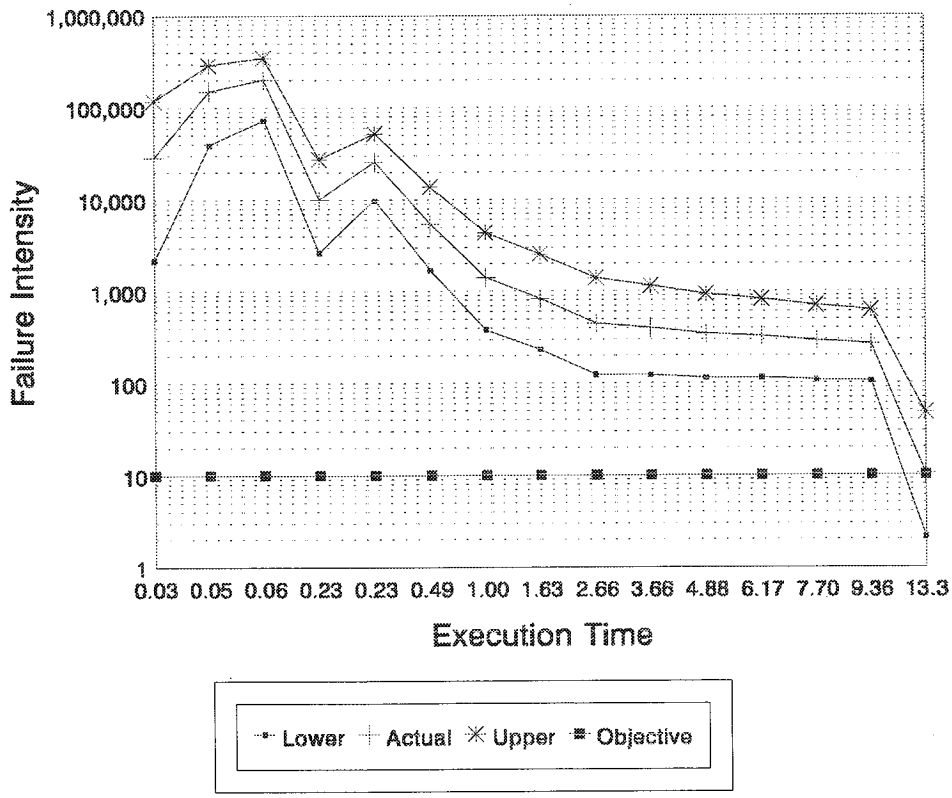


Figure 12 - Sample SRE failure intensity vs. execution time.

Failure Intensity vs. Cumulative Failure Plot

This plot provides visual aid in determining whether the software reaches the failure intensity objective in terms of number of cumulative failure occurrences. See figure 13 for a sample result.

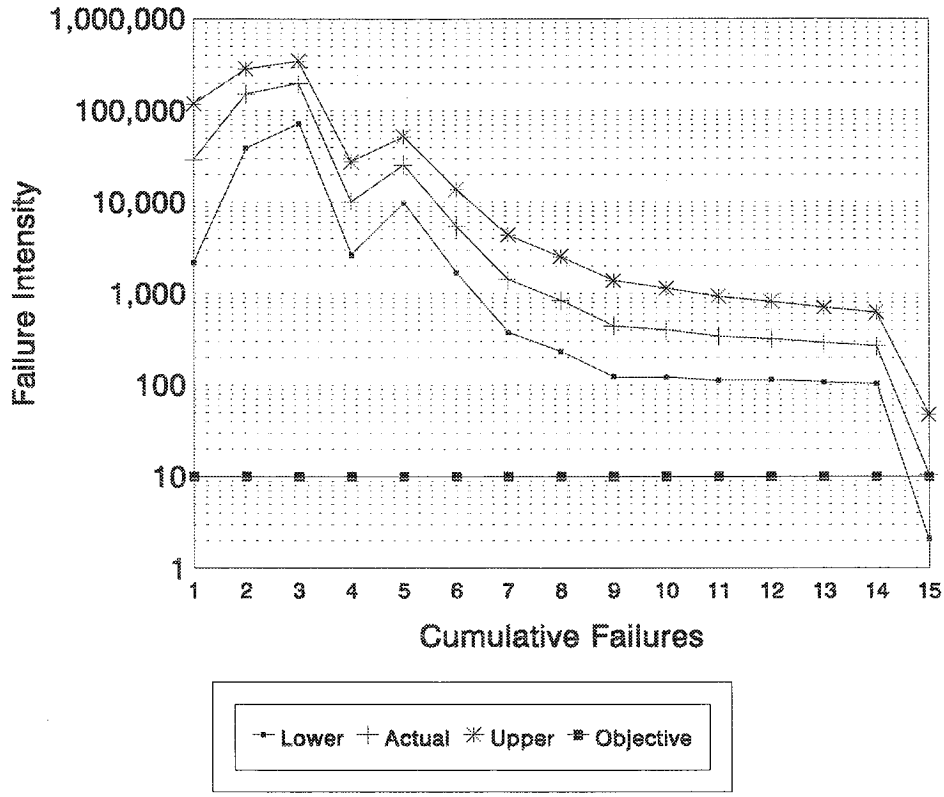


Figure 13 - Failure intensity vs. cumulative failures.

Appendix A - Definitions

Calendar time: normal 24 hour clock time

Clock time: the elapsed time from the start to the end of program execution

CPU execution: see execution time

Criticality: the degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics

Criticality: the relative measure of the consequences of a failure mode

Cyclomatic complexity: measurement to determine the structural complexity of a coded module; the measure is a count of the basic executable paths in the module expressed in terms of nodes and edges

Execution time: the actual time spent by a processor in executing the instruction of that program; also known as CPU execution time

Environment: described by the operational profile

Failure: the departure of the external results of program operation from requirements

Failure intensity function: the rate of change of the mean value function or the number of failures per unit time

Fault: a defect in the program that, when executed under particular conditions, causes a failure; created when a programmer makes a coding error

Model: a mathematical equation used to model one software reliability estimation and prediction of software using failure data

Non-homogeneous: a random process whose probability distribution varies with time

Operational profile: a set of run types that the program can execute along with the probabilities with which they will occur

Run type: runs that are identical repetitions of each other

Software reliability: the probability of failure-free operation of a computer program for a specified time in a specified environment

Software availability: the expected fraction of time during which a software component or system is functioning acceptably

Tool: a compilation of models

Appendix B - SMERFS Files

SMERFS History File:

Sample Execution Time Analysis

```
SSSSSSS  M      M  EEEEEEE  RRRRRRR  FFFFFFF  SSSSSSS  III  V     V
S        M M  M M  E        R      R  F        S          I   V   V
SSSSSSS  M  M  M  EEEE     RRRRRRR  FFFF     SSSSSSS    I    V V
      S  M      M  E        R   R    F              S    I    V V
SSSSSSS  M      M  EEEEEEE  R     R  F        SSSSSSS  III    V
```

```
SOFTWARE BASELINE NUMBER    :      4
SOFTWARE BASELINE DATE      :      21 JUNE    1990

SOFTWARE REVISION LETTER    :      0
SOFTWARE REVISION DATE      :      __ _____ 19__
```

PLEASE ENTER OUTPUT FILE NAME FOR HISTORY FILE; A ZERO IF THE FILE IS NOT DESIRED, OR A ONE FOR DETAILS ON THE FILE.

THE HISTORY FILE IS A COPY OF THE ENTIRE INTERACTIVE SESSION. THE FILE CAN BE USED FOR LATER ANALYSIS AND/OR DOCUMENTATION.

tryone.hst

PLEASE ENTER OUTPUT FILE NAME FOR THE PLOT FILE; A ZERO IF THE FILE IS NOT DESIRED, OR A ONE FOR DETAILS ON THE FILE.

THE PLOT FILE IS A COPY OF THE DATA USED TO PRODUCE THE PLOTS. THE FILE CAN BE USED (BY A USER-SUPPLIED) GRAPHICS PROGRAM TO GENERATE HIGH QUALITY PLOTS. THIS PROCESSING IS HIGHLY SUGGESTED IF THE INTERNAL LINE PRINTER PLOTTER IS BEING USED; THE GENERATED PLOTS ARE RATHER CRUDE.

tryone.plt

PLEASE ENTER INPUT FILE NAME FOR THE SMERFS MODEL ASSUMPTIONS AND DATA REQUIREMENTS FILE.

smf4rdc.lis

PLEASE ENTER THE DESIRED DATA TYPE, OR ZERO FOR A LIST.

0

THE AVAILABLE DATA TYPES ARE
1 WALL CLOCK TIME-BETWEEN-ERROR (WC TBE)
2 CENTRAL PROCESSING UNITS (CPU) TBE
3 WC TBE AND CPU TBE
4 INTERVAL COUNTS AND LENGTHS
PLEASE ENTER THE DESIRED DATA TYPE.

2

PLEASE ENTER 1 FOR FILE INPUT; ELSE ZERO.

0

PLEASE ENTER 1 FOR KEYBOARD INPUT; ELSE ZERO.

1

A RESPONSE OF NEGATIVE VALUES FOR THE PROMPT "PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED" WILL END THE PROCESSING.

PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.

15.000000000000000 1

PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.

23.000000000000000 1

PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
20.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
28.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
35.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
37.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
43.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
52.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
49.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
58.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
67.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
62.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
76.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
82.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
101.000000000000000 1
PLEASE ENTER CPU; AND 0 FOR ERROR-FREE OR 1 FOR ERRORED.
-1.000000000000000 -1

PLEASE ENTER 1 TO LIST THE CURRENT DATA; ELSE ZERO.
1

ERROR NO	TIME-BETWEEN
1	.15000000E+02
2	.23000000E+02
3	.20000000E+02
4	.28000000E+02
5	.35000000E+02
6	.37000000E+02
7	.43000000E+02
8	.52000000E+02
9	.49000000E+02
10	.58000000E+02
11	.67000000E+02
12	.62000000E+02
13	.76000000E+02
14	.82000000E+02
15	.10100000E+03

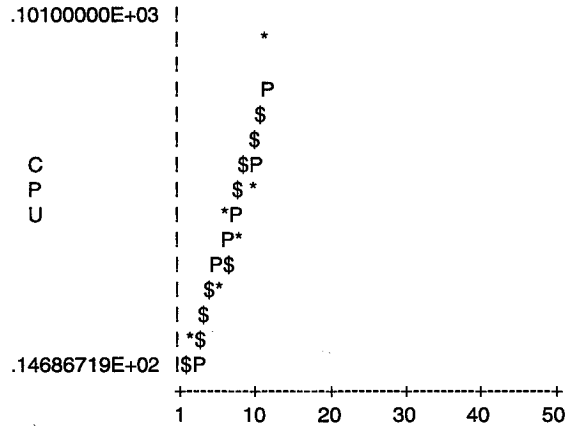
PLEASE ENTER 1 FOR THE PROGRAM TO MAKE NEW DATA FILES; ELSE ZERO.
(YOUR RESPONSE WILL BE USED THROUGHOUT THE EXECUTION; AND A ZERO
WILL VOID THE DATA RESTORE OPTION IN DATA TRANSFORMATIONS.)
1

PLEASE ENTER OUTPUT NAME FOR CPU TBE DATA.
tryone.dat
THE OUTPUT OF 15 CPU TBE ELEMENTS WAS PERFORMED.
1

PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.
0

```
THE AVAILABLE MAIN MODULE OPTIONS ARE
 1 DATA INPUT                    5 PLOT(S) OF THE RAW DATA
 2 DATA EDIT                     6 EXECUTIONS OF THE MODELS
 3 DATA TRANSFORMATIONS          7 ANALYSES OF MODEL FIT
 4 DATA STATISTICS               8 STOP EXECUTION OF SMERFS
PLEASE ENTER THE MAIN MODULE OPTION.
      4

                        CPU TIME-BETWEEN-ERROR
                        WITH TOTAL TESTING TIME OF  .74800000E+03
                        AND TOTAL ERRORED TIME OF  .74800000E+03
                        ************************************************
MEDIAN OF THE DATA      *              .49000000E+02                    *
LOWER & UPPER HINGES    *   .31500000E+02    .64500000E+02              *
MINIMUM AND MAXIMUM     *   .15000000E+02    .10100000E+03              *
NUMBER OF ENTRIES       *                15                             *
AVERAGE OF THE DATA     *              .49866667E+02                    *
STD. DEV. & VARIANCE    *   .24761337E+02    .61312381E+03              *
SKEWNESS & KURTOSIS     *   .42596916E+00   -.64196038E+00              *
                        ************************************************
1

PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.
      6

PLEASE ENTER THE TIME MODEL OPTION, OR ZERO FOR A LIST.
      0
THE AVAILABLE WALL CLOCK OR CPU TIME MODELS ARE
 1 THE LITTLEWOOD AND VERRALL BAYESIAN MODEL
 2 THE MUSA BASIC EXECUTION TIME MODEL
 3 THE GEOMETRIC MODEL
 4 THE NHPP MODEL FOR TIME-BETWEEN-ERROR OCC.
 5 THE MUSA LOG POISSON EXECUTION TIME MODEL
 6 RETURN TO THE MAIN PROGRAM
PLEASE ENTER THE MODEL OPTION.
      1
PLEASE ENTER 1 FOR MODEL DESCRIPTION; ELSE ZERO.
      0

PLEASE ENTER 1 FOR MAXIMUM LIKELIHOOD, 2 FOR LEAST SQUARES, OR
 3 TO TERMINATE MODEL EXECUTION.
      1

WHICH OF THE FOLLOWING FUNCTIONS DO YOU DESIRE TO USE AS THE
 PHI(I) IN THE GAMMA DISTRIBUTION?  THE GAMMA IS USED AS THE
 PRIOR WITH PARAMETERS ALPHA AND PHI(I)

 1. PHI(I) = BETA(0) + BETA(1) * I      (LINEAR)
OR
 2. PHI(I) = BETA(0) + BETA(1) * I**2  (QUADRATIC).
      1

THE INITIAL ESTIMATES FOR BETA(0) AND BETA(1) ARE:
   BETA(0) :  .65523810E+01
   BETA(1) :  .54142857E+01

 PLEASE ENTER 1 TO USE THESE INITIAL ESTIMATES FOR BETA(0)  AND
  BETA(1), OR 2 TO INPUT INITIAL ESTIMATES.
      1
PLEASE ENTER THE MAXIMUM NUMBER OF ITERATIONS.
     100

ML MODEL ESTIMATES AFTER 25 ITERATIONS ARE:
   ALPHA  :  .13544290E+05
   BETA(0) :  .13173016E+06
   BETA(1) :  .67176336E+05
```

```
 THE FUNCTION EVALUATED AT THESE POINTS IS  .71757203E+02

 PLEASE ENTER 1 FOR AN ESTIMATE OF THE MEAN TIME BEFORE THE NEXT
  ERROR; ELSE ZERO.
      1
 THE EXPECTED TIME IS  .89088509E+02

 PLEASE ENTER 1 FOR MAXIMUM LIKELIHOOD,  2 FOR LEAST SQUARES, OR
  3 TO TERMINATE MODEL EXECUTION.
      3

 PLEASE ENTER 1 TO COMPUTE THE PREDICTED TBES; ELSE ZERO.
      1

 PLEASE ENTER THE TIME MODEL OPTION, OR ZERO FOR A LIST.
      0
 THE AVAILABLE WALL CLOCK OR CPU TIME MODELS ARE
  1 THE LITTLEWOOD AND VERRALL BAYESIAN MODEL
  2 THE MUSA BASIC EXECUTION TIME MODEL
  3 THE GEOMETRIC MODEL
  4 THE NHPP MODEL FOR TIME-BETWEEN-ERROR OCC.
  5 THE MUSA LOG POISSON EXECUTION TIME MODEL
  6 RETURN TO THE MAIN PROGRAM
 PLEASE ENTER THE MODEL OPTION.
      6
1

 PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.
      0
 THE AVAILABLE MAIN MODULE OPTIONS ARE
  1 DATA INPUT                    5 PLOT(S) OF THE RAW DATA
  2 DATA EDIT                     6 EXECUTIONS OF THE MODELS
  3 DATA TRANSFORMATIONS          7 ANALYSES OF MODEL FIT
  4 DATA STATISTICS               8 STOP EXECUTION OF SMERFS
 PLEASE ENTER THE MAIN MODULE OPTION.
      7
 THE ANALYSES PORTION WILL OPERATE ON YOUR LAST SELECTED PREDIC-
 TIONS FROM THE LITTLEWOOD AND VERRALL BAYESIAN MODEL. PLEASE EN-
 TER A ONE TO CONTINUE WITH THE ANALYSES; OTHERWISE A ZERO TO RE-
 TURN TO THE MAIN MODULE MENU.
      1

 PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.
      0
 THE AVAILABLE ANALYSES OPTIONS ARE
  1 GOODNESS-OF-FIT TESTS AND DATA LISTINGS
  2 PLOT OF ORIGINAL AND PREDICTED DATA
  3 PLOT OF THE RESIDUAL DATA
  4 RETURN TO THE MAIN PROGRAM
 PLEASE ENTER THE ANALYSES OPTION.
 PLEASE ENTER 1 FOR THE DATA LISTING; ELSE ZERO.
      0

 PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.
      0
 THE AVAILABLE ANALYSES OPTIONS ARE
  1 GOODNESS-OF-FIT TESTS AND DATA LISTINGS
  2 PLOT OF ORIGINAL AND PREDICTED DATA
  3 PLOT OF THE RESIDUAL DATA
  4 RETURN TO THE MAIN PROGRAM
 PLEASE ENTER THE ANALYSES OPTION.
      2
 PLEASE ENTER THE PLOT TITLE (UP TO 30 CHARACTERS).
      (type your title)
```



PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.

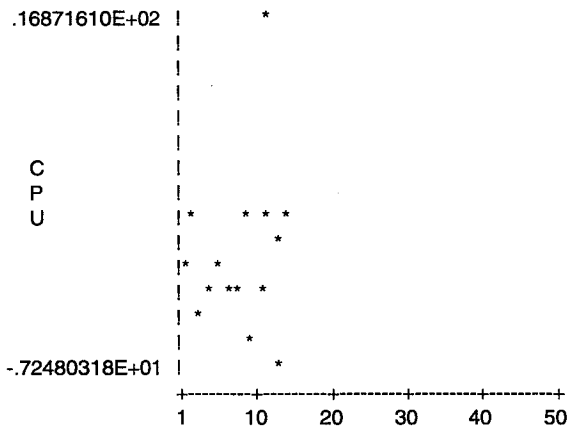
3

PLEASE ENTER THE PLOT TITLE (UP TO 30 CHARACTERS).

(type your title)

PLEASE ENTER 1 TO SMOOTH THE RESIDUALS; ELSE ZERO.

0



PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.

0

THE AVAILABLE ANALYSES OPTIONS ARE
1 GOODNESS-OF-FIT TESTS AND DATA LISTINGS
2 PLOT OF ORIGINAL AND PREDICTED DATA
3 PLOT OF THE RESIDUAL DATA
4 RETURN TO THE MAIN PROGRAM
PLEASE ENTER THE ANALYSES OPTION.

4

PLEASE ENTER ONE TO PLACE THE PLOT DATA ON THE OPTIONAL SMERFS PLOT FILE; ELSE ZERO.

1

PLEASE ENTER A PLOT TITLE FOR ALL DATA (UP TO 40 CHARACTERS).

Littlewood & Verral Maximum Likelihood L

1

PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.

0

THE AVAILABLE MAIN MODULE OPTIONS ARE

1 DATA INPUT	5 PLOT(S) OF THE RAW DATA
2 DATA EDIT	6 EXECUTIONS OF THE MODELS
3 DATA TRANSFORMATIONS	7 ANALYSES OF MODEL FIT
4 DATA STATISTICS	8 STOP EXECUTION OF SMERFS

PLEASE ENTER THE MAIN MODULE OPTION.

8

THE SMERFS EXECUTION HAS ENDED.

SMERFS PLOT File:

Sample Interval Data Analysis

```
SSSSSSS   M      M  EEEEEEE  RRRRRRR   FFFFFFF  SSSSSSS  III  V     V
S         MMMMM  E        R      R  F        S        I    V   V
SSSSSSS   M M M  EEEE     RRRRRRR   FFFF     SSSSSSS  I     V V
      S   M   M  E        R    R    F              S   I     VV
SSSSSSS   M      M  EEEEEEE  R      R  F        SSSSSSS  III    V
```

SOFTWARE BASELINE NUMBER : 4
SOFTWARE BASELINE DATE : 21 JUNE 1990

SOFTWARE REVISION LETTER : 0
SOFTWARE REVISION DATE : __ _____ 19__

PLEASE ENTER OUTPUT FILE NAME FOR HISTORY FILE; A ZERO IF THE FILE IS NOT DESIRED, OR A ONE FOR DETAILS ON THE FILE.

trytwo.hst

PLEASE ENTER OUTPUT FILE NAME FOR THE PLOT FILE; A ZERO IF THE FILE IS NOT DESIRED, OR A ONE FOR DETAILS ON THE FILE.

trytwo.plt

PLEASE ENTER INPUT FILE NAME FOR THE SMERFS MODEL ASSUMPTIONS AND DATA REQUIREMENTS FILE.

smf4rdc.lis

PLEASE ENTER THE DESIRED DATA TYPE, OR ZERO FOR A LIST.

4

PLEASE ENTER 1 FOR FILE INPUT; ELSE ZERO.

0

PLEASE ENTER 1 FOR KEYBOARD INPUT; ELSE ZERO.

1

A RESPONSE OF NEGATIVE VALUES FOR THE PROMPT "PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH" WILL END THE PROCESSING.

PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
35.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
38.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
27.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
21.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
18.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
19.000000000000000 1.000000000000000

PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
14.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
20.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
9.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
4.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
0.000000000000000E+000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
2.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
1.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
0.000000000000000E+000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
2.000000000000000 1.000000000000000
PLEASE ENTER THE ERROR COUNT AND TESTING LENGTH.
-1.000000000000000 -1.000000000000000

PLEASE ENTER 1 TO LIST THE CURRENT DATA; ELSE ZERO.
1

INTERVAL	NO. OF ERRORS	TESTING LENGTH
1	.35000000E+02	.10000000E+01
2	.38000000E+02	.10000000E+01
3	.27000000E+02	.10000000E+01
4	.21000000E+02	.10000000E+01
5	.18000000E+02	.10000000E+01
6	.19000000E+02	.10000000E+01
7	.14000000E+02	.10000000E+01
8	.20000000E+02	.10000000E+01
9	.90000000E+02	.10000000E+01
10	.40000000E+02	.10000000E+01
11	.00000000E+00	.10000000E+01
12	.20000000E+01	.10000000E+01
13	.10000000E+01	.10000000E+01
14	.00000000E+00	.10000000E+01
15	.20000000E+01	.10000000E+01

PLEASE ENTER 1 FOR THE PROGRAM TO MAKE NEW DATA FILES; ELSE ZERO.
(YOUR RESPONSE WILL BE USED THROUGHOUT THE EXECUTION; AND A ZERO
WILL VOID THE DATA RESTORE OPTION IN DATA TRANSFORMATIONS.)
1

PLEASE ENTER OUTPUT NAME FOR INTERVAL DATA.
trytwo.dat
THE OUTPUT OF 15 INTERVAL ELEMENTS WAS PERFORMED.
1

PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.
0
THE AVAILABLE MAIN MODULE OPTIONS ARE

1 DATA INPUT	5 PLOT(S) OF THE RAW DATA
2 DATA EDIT	6 EXECUTIONS OF THE MODELS
3 DATA TRANSFORMATIONS	7 ANALYSES OF MODEL FIT
4 DATA STATISTICS	8 STOP EXECUTION OF SMERFS

PLEASE ENTER THE MAIN MODULE OPTION.
4

```
                INTERVAL DATA WITH EQUAL LENGTHS
                 WITH ERROR COUNTS TOTALING TO      210
                ********************************************
MEDIAN OF THE DATA      *           .14000000E+02              *
LOWER & UPPER HINGES    *   .20000000E+01   .20500000E+02      *
MINIMUM AND MAXIMUM     *   .00000000E+00   .38000000E+02      *
NUMBER OF ENTRIES       *              15                      *
AVERAGE OF THE DATA     *           .14000000E+02              *
STD. DEV. & VARIANCE    *   .12778330E+02   .16328571E+03      *
SKEWNESS & KURTOSIS     *   .48772605E+00   -.94226755E+00     *
                ********************************************
1

 PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.
      0
 THE AVAILABLE MAIN MODULE OPTIONS ARE
  1 DATA INPUT                    5 PLOT(S) OF THE RAW DATA
  2 DATA EDIT                     6 EXECUTIONS OF THE MODELS
  3 DATA TRANSFORMATIONS          7 ANALYSES OF MODEL FIT
  4 DATA STATISTICS               8 STOP EXECUTION OF SMERFS
 PLEASE ENTER THE MAIN MODULE OPTION.
      6

 PLEASE ENTER THE COUNT MODEL OPTION, OR ZERO FOR A LIST.
      0
 THE AVAILABLE ERROR COUNT MODELS ARE
  1 THE GENERALIZED POISSON MODEL
  2 THE NON-HOMOGENEOUS POISSON MODEL
  3 THE BROOKS AND MOTLEY MODEL
  4 THE SCHNEIDEWIND MODEL
  5 THE S-SHAPED RELIABILITY GROWTH MODEL
  6 RETURN TO THE MAIN PROGRAM
 PLEASE ENTER THE MODEL OPTION.
      4
 PLEASE ENTER 1 FOR MODEL DESCRIPTION; ELSE ZERO.
      0
 PLEASE ENTER 1 FOR DESCRIPTION OF TREATMENT TYPES; ELSE ZERO.
      1
  TREATMENT 1 -  UTILIZE ALL THE ERROR COUNTS FROM EACH OF THE
                 THE TESTING PERIODS.

  TREATMENT 2 -  IGNORE THE ERROR COUNTS FROM THE FIRST S-1 TEST-
                 ING PERIODS, AND USE ONLY THE ERROR COUNTS FROM
                 PERIOD S THROUGH THE TOTAL NUMBER OF PERIODS.

  TREATMENT 3 -  USE THE CUMULATIVE NUMBER OF ERRORS FROM PERIODS
                 1 THROUGH S-1,  AND THE INDIVIDUAL COUNTS FROM
                 PERIOD S THROUGH THE TOTAL NUMBER OF PERIODS.

 PLEASE ENTER THE DESIRED MODEL TREATMENT NUMBER, OR A 4 TO TER-
  MINATE MODEL EXECUTION.
      1

 TREATMENT 1 MODEL ESTIMATES ARE:
  BETA    .22695165E+00
  ALPHA   .49298066E+02
 AND THE WEIGHTED SUMS-OF-SQUARES BETWEEN THE PREDICTED AND OB-
  SERVED ERROR COUNTS IS  .15366929E+04

 PLEASE ENTER 1 FOR AN ESTIMATE OF THE NUMBER OF ERRORS EXPECTED
  IN THE NEXT TESTING PERIOD; ELSE ZERO.
      1
  THE EXPECTED NUMBER OF ERRORS IS  .14656216E+01
```

PLEASE ENTER 1 FOR AN ESTIMATE OF THE NUMBER OF TESTING PERIODS NEEDED TO DISCOVER THE NEXT M ERRORS; ELSE ZERO.

1

PLEASE ENTER THE VALUE FOR M.

1.000000000000000

THE EXPECTED NUMBER OF PERIODS IS .65706276E+00

PLEASE ENTER 1 TO TRY A DIFFERENT VALUE FOR M; ELSE ZERO.

0

PLEASE ENTER THE DESIRED MODEL TREATMENT NUMBER, OR A 4 TO TERMINATE MODEL EXECUTION.

4

PLEASE ENTER 1 TO COMPUTE THE PREDICTED INTERVAL ERROR COUNTS; ELSE ZERO.

1

PLEASE ENTER THE COUNT MODEL OPTION, OR ZERO FOR A LIST.

6

1

PLEASE ENTER THE MAIN MODULE OPTION, OR ZERO FOR A LIST.

7

THE ANALYSES PORTION WILL OPERATE ON YOUR LAST SELECTED PREDICTIONS FROM THE SCHNEIDEWIND (FOR ERROR COUNT) MODEL. PLEASE ENTER A ONE TO CONTINUE WITH THE ANALYSES; OTHERWISE A ZERO TO RETURN TO THE MAIN MODULE MENU.

0

PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.

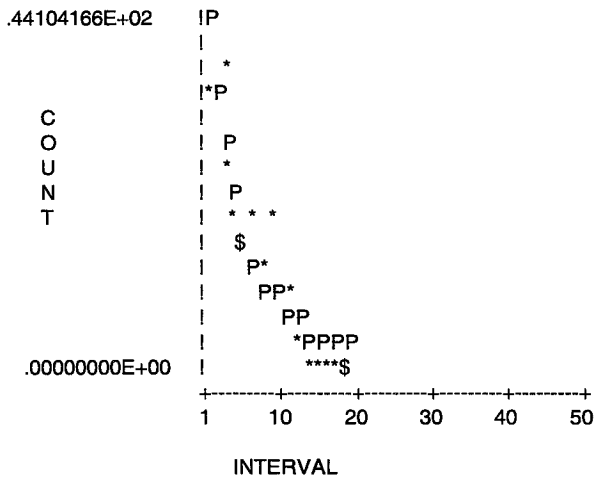
0

THE AVAILABLE ANALYSES OPTIONS ARE
1 GOODNESS-OF-FIT TESTS AND DATA LISTINGS
2 PLOT OF ORIGINAL AND PREDICTED DATA
3 PLOT OF THE RESIDUAL DATA
4 RETURN TO THE MAIN PROGRAM
PLEASE ENTER THE ANALYSES OPTION.

2

PLEASE ENTER THE PLOT TITLE (UP TO 30 CHARACTERS).

(type your title)



PLEASE ENTER THE ANALYSES OPTION; OR ZERO FOR A LIST.

4

PLEASE ENTER ONE TO PLACE THE PLOT DATA AND CHI-SQUARE STATISTIC
ON THE OPTIONAL SMERFS PLOT FILE; ELSE ZERO.

1

THE AVAILABLE MAIN MODULE OPTIONS ARE

1 DATA INPUT	5 PLOT(S) OF THE RAW DATA
2 DATA EDIT	6 EXECUTIONS OF THE MODELS
3 DATA TRANSFORMATIONS	7 ANALYSES OF MODEL FIT
4 DATA STATISTICS	8 STOP EXECUTION OF SMERFS

PLEASE ENTER THE MAIN MODULE OPTION.

8

THE SMERFS EXECUTION HAS ENDED.

REFERENCES

[1] ASQC Software Metrics Tutorial Notes.

[2] Khoshgoftaar, Taghi M. and Munson, John C. "Predicting Software Development Errors Using Software Complexity Metrics" IEEE Journal on Selected Areas in Communications. Vol. 8, No. 2. February 1990.

[3] Friedman, M. A. et. al. "Reliability Techniques for Combined Hardware and Software Systems." Final Report, Contract No. F30602-89-C-0111. Rome Laboratory. 1991.

[4] Zuse, Horst. "Measuring Factors Contributing to Software Maintenance Complexity." Minutes from the ASQC Second International Conference On Software Quality. Research Triangle Park, NC. October 1992.

[5] Montgomery, Douglas C. and Peck, Elizabeth A. Introduction To Linear Regression Analysis. Wiley and Sons. 1992.

[6] McCabe, Thomas J. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. NBS Special Publication 500-99. U. S. Department of Commerce. December 1982.

[7] Musa, John D. "Operational Profiles in Software-Reliability Engineering." IEEE Software. March 1993. pp. 14-32.

[8] Braley, Dennis "Test Case Effectiveness Measuring Using the Logiscope Dynamic Analysis Capability." JSC Software Technology Branch.

[9] McGilton and Morgan, Introducing the UNIX System. McGraw Hill Book Company. 1983.

[10] IEEE Standard Glossary of Software Engineering Terminology. ANSI/IEEE Std 729-1983, p. 32.

[11] Martin L. Shooman, Software Engineering Design, Reliability, and Management. McGraw Hill. 1983. p. 304.

[12] Musa, John D.; Iannino, Anthony; and Okumoto, Kazuhira. Software Reliability Measurement, Prediction, Application. McGraw Hill. New York.

[13] Farr, William H. A Survey of Software Reliability, Modeling, and Estimation. Naval Surface Warfare Center. NAVSWC TR 82-171. September 1983.

[14] Farr, William H., and Smith, Oliver D. Statistical Modeling and Estimation of Reliability Function for Software (SMERFS) Users Guide. Naval Surface Warfare Center. NAVSWC TR 84-373 Rev. 2. March 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Aug/94	Technical Memorandum

4. TITLE AND SUBTITLE

Software Analysis Handbook: Software Complexity Analysis and Software Reliability Estimation and Prediction

5. FUNDING NUMBERS

6. AUTHOR(S)

Alice T. Lee; Todd Gunn*; Tuan Pham*; Ron Ricaldi*

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Safety, Reliability, and Quality Assurance Office
Space Station Safety and Mission Assurance Division
Lyndon B. Johnson Space Center
Houston, Texas 77058

8. PERFORMING ORGANIZATION REPORT NUMBERS

S-774

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

TM-104799

11. SUPPLEMENTARY NOTES

*Loral Space Information Systems, Houston, Texas

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited
Available from the NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390 Subject category: 59

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This handbook documents the three software analysis processes the Space Station Software Analysis team uses to assess Space Station software, including their backgrounds, theories, tools, and analysis procedures. Potential applications of these analysis results are also presented.

The first section describes how software complexity analysis provides quantitative information on code, such as code structure and risk areas, throughout the software life cycle. Software complexity analysis allows an analyst to understand the software structure; identify critical software components; assess risk areas within a software system; identify testing deficiencies; and recommend program improvements. Performing this type of analysis during the early design phases of software development can positively affect the process, and may prevent later, much larger, difficulties.

The second section describes how software reliability estimation and prediction analysis, or software reliability, provides a quantitative means to measure the probability of failure-free operation of a computer program, and describes the two tools used by JSC to determine failure rates and design tradeoffs between reliability, costs, performance, and schedule.

14. SUBJECT TERMS

computer systems design, software engineering, program verification, computer programs, computer systems performance, design analysis, reliability analysis

15. NUMBER OF PAGES
91

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited