COMPREHENSIVE FORECASTING OF SOFTWARE INTEGRITY IN C'I SYSTEMS

by

Edward Hirschman

A project report submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING

in

Systems Engineering

APPROVED:

[Signatures]

Benjamin S. Blanchard, Chairman

Walter J. Fabrycky

Walter R. Beam

October 1992

Blacksburg, Virginia
COMPREHENSIVE FORECASTING OF SOFTWARE INTEGRITY IN C'I SYSTEMS

by

Edward Hirschman

Committee Chairman: Benjamin S. Blanchard
Systems Engineering

(ABSTRACT)

The purpose of this project is to forecast the incidence of failures to be encountered by a software package for C'I systems over and throughout its life cycle.

It will be assumed that a data base of software previously developed for C'I systems will be used to forecast the software integrity of a software package under initial development. "Software integrity" is defined as a projection of the stream of failures that will be experienced by the new software. The failure history of the mature C'I systems software will be statistically quantified parametrically and by experimental design techniques (ANOVA) to gather information which will be used to forecast what C'I software with similar characteristics--length, language, debugging effort, etc.--will experience.

Then, as the new C'I system software matures, statistical techniques for software systems engineering will be addressed for testing appropriateness of the initial projections; and
eventually the new software will be parametrically modeled on its own merits to forecast the failures to be encountered over the remainder of its life cycle.

Lastly, the data base history of software for mature C'I systems software will be updated and amended as needed to facilitate reliable forecasting of software integrity for a new round of C'I systems software.

The attention to C'I implied by the title of the project will reflect itself in the classes of software considered and development conditions, schedules and complexities of the software.
ACKNOWLEDGEMENTS

First, I would like to thank my committee members Benjamin S. Blanchard (chairman), Wolter J. Fabrycky and Walter R. Beam for helping me with much direction and encouragement in the development of this project report. However, a special thanks is owed to Walter R. Beam for donating over one hundred hours of his time in the early stages of the development of this project report, carefully going over the numerous drafts and providing me with private instruction enabling me to nurture my skills in areas of this report with which I had little prior familiarity; for this I am greatly in debt to him and hope that someday a means to return the favor to him will surface.

Because this project report has a considerable, advanced statistical science content, a because none of my committee members are strongly familiar with statistical science, I had four statisticians referee the project report and provide my committee with guidance on how to interpret the validity and appropriateness of the statistical science content of this project report. From George Mason University, Carl M. Harris, Douglas R. Miller and Muhammad Habib reviewed the project report's statistical science content and jointly through Carl M. Harris wrote a favorable letter of their findings to
Benjamin S. Blanchard, chairman of my committee. From Va Tech, Cyrus J. Staniec also wrote a favorable letter to the committee. Cyrus J. Staniec deserves a special thanks for going over Chapters 4, 5 and 6 with a fine-toothed comb and making excellent comments and corrections which have been included in this project report, particularly with regard to the Analysis of Variance presentation herein. I thank Carl M. Harris here for giving me my first training in the field of software integrity, without which instruction the completion of this project report would have been considerably more difficult.

I greatly appreciate the time and effort expended by Andrea Newson in the professional preparation of the final copy of this project report.

I accept sole responsibility for any errors that remain in this project report.
# TABLE OF CONTENTS

ABSTRACT .................................................. ii

ACKNOWLEDGEMENTS ....................................... iv

LIST OF TABLES ............................................ viii

LIST OF FIGURES .......................................... ix

1. INTRODUCTION .......................................... 1
   1.1 Objectives ........................................ 1
   1.2 System Environment ................................ 2
   1.3 Some Key Definitions .............................. 6
   1.4 An Overview of the Following Chapters ........... 13

2. TAXONOMY OF SOFTWARE ERRORS ......................... 18
   2.1 Introduction ...................................... 18
   2.2 Initial and Ultimate Effects of Software Errors .................................................. 20
   2.3 Error Latency and Detection .................... 23
   2.4 Underlying Causes of Software Errors .......... 25
   2.5 Related Issues .................................. 26
   2.6 Remarks .......................................... 30

3. SUMMARY OF SOFTWARE INTEGRITY MODELS TO BE USED IN THIS PROJECT .................................. 32
   3.1 Introduction ...................................... 32
   3.2 The Basic Execution Time Model ................ 38
   3.3 The Logarithmic Poisson Execution Time Model . 40
   3.4 Selection of the Model .......................... 43

4. CREATING AND UPDATING A DATA BASE SAMPLE .......... 47
   4.1 Introduction ...................................... 47
   4.2 Selection of Mature Modules of Software for the Data Base Sample .................................. 48
   4.3 Updating the Data Base Sample .................. 58
   4.4 Analysis of Stakeholder Profiles ............... 58

5. THE PHASE OF ALPHA TESTING .......................... 62
   5.1 Introduction ...................................... 62
   5.2 Prediction for Alpha Testing .................... 63
   5.3 A Supplementary Prediction Technique .......... 67
   5.4 Estimation for Alpha Testing .................... 69
5.5 Determining Calendar Time from Execution Time Objective ........................................... 74

6. ESTIMATION FOR BETA TESTING AND RELEASE TO THE FIELD ................................................................. 76
   6.1 Introduction ................................................................................. 76
   6.2 Generating an Analyzable Beta Test Execution Time Stream of Failures ......................................................... 79
   6.3 Estimation of Failure Decay Parameter \( \theta \) .................................................. 83
   6.4 Estimation for Beta Testing ...................................................... 86
   6.5 Release to the Field .................................................................. 88

7. CONCLUDING REMARKS ................................................................. 90

SELECTED REFERENCES .................................................................. 93

VITA .............................................................................................. 95


LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Illustrations of Ways to Characterize Temporal Properties</td>
<td>12</td>
</tr>
<tr>
<td>1(a)</td>
<td>Time-based Failure Specification</td>
<td>12</td>
</tr>
<tr>
<td>1(b)</td>
<td>Failure-based Failure Specification</td>
<td>12</td>
</tr>
<tr>
<td>2.</td>
<td>Choice Between Models</td>
<td>45</td>
</tr>
<tr>
<td>3.</td>
<td>Untitled</td>
<td>52</td>
</tr>
<tr>
<td>4.</td>
<td>Table for $Q_{x,n}$</td>
<td>56</td>
</tr>
<tr>
<td>5.</td>
<td>Five Digit Pseudo Random Numbers</td>
<td>82</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>An Abbreviated Diagram of a Simple Civilian Air Traffic System</td>
<td>4</td>
</tr>
<tr>
<td>2.</td>
<td>Gross Phases for Life Cycle of C'I Software Flowchart</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>Flow Diagram for Software Testing</td>
<td>15</td>
</tr>
<tr>
<td>4.</td>
<td>Untitled</td>
<td>50</td>
</tr>
<tr>
<td>5.</td>
<td>Untitled</td>
<td>61</td>
</tr>
<tr>
<td>6.</td>
<td>Conceptual View of Parameter Estimation</td>
<td>70</td>
</tr>
<tr>
<td>7.</td>
<td>Untitled</td>
<td>81</td>
</tr>
<tr>
<td>8.</td>
<td>The Illustrative Single Stream of Failure Intervals and End Bits</td>
<td>84</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

Section 1.1: Objectives

Software reliability refers to absence of unexpected and unpredictable failures of a programmed process during normal operation. Unlike mechanical or electronic failures in hardware, the causes of software failures are built into software during its design. Thus, systems which exhibit low rates of software-related failures can be properly characterized as having high software integrity. We shall employ this term.

Many of these failure sources are removed during development and testing by the developer. This initial testing is commonly referred to by software developers as alpha testing. Most software is subjected to an additional regimen of beta testing, in which a representative sample of users employs the software in normal operations. Similar development processes are employed for developing software in many types of applications. Software to be used for non-critical applications and in only a few organizations is often developed using ad hoc processes. In such instances, software

\[\text{Software failure will be specifically defined in Section 1.3.}\]
integrity is handled in a largely qualitative manner. In systems such as Command, Control, Communications and Intelligence systems (for which an example follows in Section 1.2), unreliability of the software could result in large loss of life or materiel. Accordingly, expending greater resources is justified to ensure high operational integrity. Our study deals with software of sufficient importance that much attention will be paid to its integrity.

Earlier studies of software integrity have emphasized ad hoc applications of some statistical model of failure occurrences. They did not deal comprehensively with the phases of the development process. The principal aim of our study is to apply modern statistical methods systematically to the software development process. A specific objective is ability to set criteria for integrity of software at completion of its development, within constraints on development schedule and cost. In addition, we will study how these methods can be applied to forecasting integrity beyond the development phase of the life cycle.

Section 1.2: System Environment

Command, Control, Communications and Intelligence (C'I) software is the keystone of modern C'I systems. Such software is used, for example, for military purposes, in law
enforcement and in civilian air traffic control. A C'I failure could cause the loss of a battle or even the war; the continued unidentification of a serial killer while he continues to torture and kill people; or the crash of a jumbo jet with the loss of hundreds of lives. Hence, a high level of integrity is necessary and sought for C'I software.

Figure 1 presents an abbreviated diagram for a simple civilian air traffic system, involving control of aircraft traffic at an airport, as an example of a C'I system. Regarding this figure, Beam (1989) writes:

Arriving aircraft announce their presence by radio, are located by radar with the aid of transponders in each aircraft, and are assigned nonconflicting landing instructions. A short-range radar may be used to locate aircraft on runways and taxiways, where they are controlled by additional controller personnel. Departures are handled by other personnel, with aircraft being directed in paths which do not conflict with those of arrivals.

Beam (1989) also writes:

The objectives of civil systems of this type are to ensure orderly traffic flow and absence of collisions in the vicinity of busy airfields and on long-distance city-to-city routes. Military air traffic systems, not much different in form, may control the operations of fighter aircraft guarding against incursion by enemy fighters and bombers.

With regard to Figure 1, this report is especially concerned with the "Computers/communications" block appearing in that figure. From the interrelation of the pieces of that figure it can be appreciated by inspection that much of the
Figure 1\textsuperscript{2}

An Abbreviated Diagram of a Simple Civilian Air Traffic System

\textsuperscript{2}This figure is excerpted from Beam (1989).
software in use in the system performs (1) message handling (similar to electronic mail), and (2) data handling (mostly searches which rely on comparisons and/or sorting). This relatively narrow focus lends itself very well to forecasting.

A narrow domain of message handling and data handling is typical of C²I software. Therefore, this report shall be mainly concerned with software that is largely dedicated to performing these functions—though it will be alright if a minor part of other functions is addressed in the software. This will accommodate most software modules¹ for C²I systems.

The term "module" is used on occasion because much of the C²I software undergoes evolutionary development over eight to twelve years so as to avoid an unacceptably long period before usable modules are released to the field for actual utilization. Usually every year or two, new or modified modules are released to the field. These modules may be designed to correct deficiencies in previous releases and also introduce new ability to the software configuration then in use.

In the evolutionary development, users' needs are met incrementally. Each increment of development produces a

¹In this report, the terms "program" and "module" are often used interchangeably. However, modules are separately created though often not independently testable software elements of an overall functioning system.
usable result.

Section 1.3: Some Key Definitions

**Alpha testing** is testing carried out by the software developer. It begins with what are considered to be fully tested individual modules and ends with the integrated software product believed suitable for putting it into the hands of users for further testing.

**Beta testing** is a transitional test following alpha test. It typically lasts for three or four months and is done by users in a collection of test sites (actual field locations) who will put the program through rigorous paces and document failures so that, at some point before release of the software to the field, the failures can be remedied. Perhaps ten percent of the sites to which the program will go to after release to the field will be involved in beta testing. A beta test site is a location at which beta testing is performed.

Figure 2 puts alpha testing and beta testing in perspective by showing how these two tests fit into the life cycle of C'I software development and use.

In Figure 2, note the dashed arrows. If, for example, the particular alpha test or beta test is aborted due to a

*The structure of this section comprises clusters of closely related terms which are defined together.*
Figure 2^5

Gross Phases for Life Cycle of C^I Software Flowchart

^5Alpha and beta testing is with what the bulk of this project report deals. Through the testing, forecasting the C^I software's integrity and identifying failures which will be corrected prior to the software's release to the field to achieve an acceptable level of integrity is the core of this project report.
program that is egregiously bad, there may be feedback loops to prior steps to resolve the situation. Illustrated here is aborting the beta test to go back to the "conceptual-preliminary design" or "detail design and development" step.

* * * * *

Software failure is a departure of software performance from required or expected performance. The program must be executing for there to be an occurrence of a failure. The term failure relates to the behavior of the program; i.e., a failure is not a "bug" or "fault."

A bug or fault is a defect in the program such that, when executed under certain conditions, causes failures. More specifically, a bug or fault is incorrect code that yields an incorrect response of the program which, especially when detected, results in a "failure" (defective output) when this section of code is activated in the overall computer program.̊

* * * * *

Herein, prediction is the use of statistics to infer, from mature pre-existing C'I software programs, the expected software performance characteristics of the program currently being tested. The narrow focus for C'I software of message handling and data handling renders the use of prior software

̊For further elaboration on the definitions of "failure," "fault" and "bug," the reader is referred to Musa et al. (1987).
to be especially relevant and informative.

Estimation is extrapolation of information from the software program being tested to forecast characteristics of the program being tested.

It is important that the reader of this report note the difference between "prediction" and "estimation." Prediction uses mature software that has been previously developed to evaluate software currently under development. Following initial to advanced testing of the software currently under development, information gathered from the testing is used by way of estimation to evaluate the same software currently under development.

* * * * *

There are three types of time with which we are concerned:

Execution time is the time that the program actually spends on a processor in executing program instructions.

Calendar time is the usual type of time that we normally experience and is continuous.

Clock time represents elapsed time from beginning to end of program execution on a running computer. Wait time and other programs' execution time are included. However, not counted in clock time are periods during which the computer is shut down or otherwise not running.
These definitions of time are widespread in the area of software systems engineering and are not limited to any particular software application. 

* * * * *

The data base (prior to updating) and initial data base are used interchangeably in this report. This constitutes all software previously developed for C'I systems from which a sample is to be drawn for prediction purposes.

The data base sample is that subset of the initial data base that is actually used for prediction purposes for a particular C'I software program.

* * * * *

In a nutshell, we desire a set of machinery--testing methods--to achieve a state of availability for the C'I software modules. "Availability" qualitatively means that the software will be useful in the field and can be utilized without further changes until the next general release of the software a year or two later. To achieve this, alternative execution routes through the software should be incorporated into the software to protect against failures that might arise in crucial steps after release of the software to the field.

7For further elaboration on these definitions of time, the reader is referred to Musa et al. (1987).
Maintenance of the software is implicitly defined as achieving the desired level of availability for the software and is usually manifested in the routine, periodic upgrade of the software at intervals of a year or two. However, at times a critical or perhaps even a life-threatening failure will surface in the software that merits immediate attention. If there are alternative steps in the software to compensate for the failure, the various sites using the software will be alerted of the problem and be advised on how to utilize the software in a way or ways to avoid encountering the failure. But sometimes there are no steps through the software to compensate for the failure. In this event, an intermediate upgrade of the software prior to the regular upgrading will be necessary. Other than the encountering of a highly significant failure, software "availability" and software "maintenance" can be viewed upon as being synonymous.

* * * * *

Musa et al. (1987) gives four basic ways to characterize temporal properties of software failure:

(1) time of failure,
(2) time interval between failures,
(3) cumulative failures experienced up to a given time,
(4) failures experienced in a particular time interval.

In Table 1 illustrates the above failure occurrences in
Table 1a

Illustrations of Ways to Characterize Temporal Properties of Software Failure

Table 1(a)

Time-based Failure Specification

<table>
<thead>
<tr>
<th>Failure number</th>
<th>Failure time (sec)</th>
<th>Failure interval (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>43</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>58</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>88</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>103</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>123</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>150</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>169</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>199</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>231</td>
<td>32</td>
</tr>
<tr>
<td>14</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>15</td>
<td>296</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 1(b)

Failure-based Failure Specification

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Cumulative failures</th>
<th>Failures in interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>90</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>120</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>150</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>180</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>210</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>240</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>270</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

*Tables 1(a) and 1(b) are excerpted from Musa et al. (1987).*
time.

The measurement of the above temporal properties of software failure differs by what type of time is being used in a particular analysis. For example, if calendar time is used instead of clock time for "time interval between failures," the time during which the computer is shut down or otherwise not running would be included in the calculation of the time involved and hence the measurement for calendar time would be greater than the corresponding measurement for clock time. As the analysis for this project report unfolds, the nature and usefulness of the different time and temporal property concepts will become clearer to the reader. In varying contexts, one temporal property is more useful than another, and one type of time is more useful than another—-with not the same temporal property or type of time being the most useful at all stages of the unfolding analysis.

Section 1.4: An Overview of the Following Chapters

First, some general comments are in order.

According to Musa et al. (1987), two principal factors affect failure behavior:

(1) the quantity of faults in the software being executed,

(2) the executing environment or operational profile of
execution.

Code inherited from other applications usually does not introduce an appreciable quantity of failures, with the possible exception of the interfaces--because of extensive prior debugging and correction for the pre-existing code. This largely assumes that the application for the inherited code in terms of functional characteristics remains the same or is closely similar, which in practice turns out to likely not be a notably restrictive requirement.

The flow process exhibited in Figure 3 gives the specific stages of testing--in conjunction with forecasting--that is developed in the unfolding of this report to the reader.

Now, an overview of the individual chapters will be given.

Chapter 2 directs us into reaching an understanding of how software errors' arise so that they can be readily detected and corrected, but their corresponding failures appropriately modeled. However, as a principal intent and minor diversion, this chapter takes an overall system perspective for finding a route to avoid making software errors. It has been demonstrated that the incidence of software design errors, whether discovered through testing and

---

³Throughout this project report, software "errors" will mean software "bug" or "fault."
start

determination of data base sample by analysis of variance (ANOVA) for prediction of alpha test duration

is data base sample "large enough"

use regression to predict alpha test duration

conduct alpha testing

estimate additional amount of alpha testing needed, if any

is additional alpha testing needed

conduct additional alpha testing

Figure 3
Flow Diagram for Software Testing
prediction of amount of beta testing needed

conduct beta testing

estimate if additional beta testing is desirable

is additional beta testing desirable

yes

release of software to the field

no

does software "perform well"

yes

add this program to data base

no

at this stage do not add this program to data base

multi-user upgrades as conditions warrant

end
use, is smaller when good design practices and tools are employed.

Chapter 3 introduces the reader to the two main software integrity modules that are currently under study but already have achieved a notable degree of acceptance by experts in the field of software integrity.

Chapter 4 concerns the use of analysis of variance and a multiple comparison method to show how a quality, appropriate-for-analysis data base sample can be derived from data describing a collection of mature candidate C'I software, to forecast the quantity of failures that the program under study is expected to experience. Also introduced in this chapter is "stakeholder analysis"--which is analysis of the vested interests the various parties involved in the software development have that can affect its quality in various ways.

Chapter 5 addresses alpha testing in detail. Chapter 6 addresses beta testing and release of the software to the field in detail. Chapter 7 makes concluding remarks.

Overall, through this report the reader will become familiar with a new approach for measuring the integrity of software for C'I systems that results in more accurate forecasting of the software's integrity.
CHAPTER 2
TAXONOMY OF SOFTWARE ERRORS

Section 2.1: Introduction

A purpose of this chapter is to reach an understanding of how software errors arise so that they can be not only readily detected and corrected, but their corresponding failures appropriately modeled. However, as a principal intent and minor diversion, this chapter takes an overall system perspective for finding a route to avoid making software errors.

One possibility that stands out for avoiding software errors is to develop more specialized hardware that will constrain the programmer in such ways that human lapses are less likely. With software costs now outstripping hardware costs, and the fact that most C'I computer use revolves around the narrow consideration of message handling and data handling, the avoidance of software errors (e.g., for C'I systems as well as others) through the development and use of specialized hardware is likely to be cost-effective and should receive more attention by the computer industry.

In the software community, detection and correction of software errors has received more attention than avoidance of those errors. But, avoidance of errors can become feasible
and in wider use if the human lapses\textsuperscript{10} that cause software errors can be characterized, so that the software programming environment is altered to avoid those lapses to make them occur less frequently or not at all.

On top of the human lapses list are omission and commission in the context of missed intent or mis-design. In the case of omission, something important is left out of the program. For commission, the algorithm can be programmed correctly but be faulty; or the algorithm can be correct but be programmed incorrectly. Combinations of these errors can and do occur, making not only the categorizing, but the correcting, of these types of errors complex and subject to introduction of new errors as a part of the "fix."

As can be appreciated, a wide variety of software errors can arise in practice. Fortunately, the narrow focus of software for C'I systems (principally, message handling and data handling) may make comprehending the spectrum of its software errors a practical part of generating software of high integrity (for C'I systems) at only a moderate additional cost.

\textsuperscript{10}We are presuming that specification correctly describes the environments in which the software is to operate. Software developers should not be held liable for unexpected changes in the environment. A specification which projects likely changes in the environment is a powerful weapon in the war against software errors.
An example of a type of software error that has been successfully addressed and reduced at a reasonable cost is syntax errors. In most cases, these are detected by modern language translators that devote much attention to detecting and explaining syntax errors that are often labeled as "fatal," in which case program translation is halted, and "warning," in which case the translated program may execute but its performance may be suspect.

With the issue of syntax errors largely resolved, the remaining errors can be termed semantic. Here the possibility arises that programming languages that use more natural terms and expressions--relative to the application--as opposed to code that must be interpreted in terms of abstract programming objectives can greatly improve software integrity, at little if any additional net cost. Regardless, this consideration is beyond the scope of the current project.

Section 2.2: Initial and Ultimate Effects of Software Errors

Most errors can be presumed to have a "starting point," which is the point that the correct or intended path is no longer followed in program execution. Below is a representative, though probably incomplete, list of ten ways this can happen, as taken form Beam (GMU working paper,
(1) An incorrect branch/jump/call is taken, which is due to a lapse or typing error.

(2) There is an incorrect number, character or string placed in a memory location, or data is taken from an incorrect location.

(3) Memory is overwritten in own or other's data or program regions or in supervisory data or program regions.

(4) There is incorrect or spurious activation of an I/O port or device.

(5) There is incorrect conversion of input and output data.

(6) The wrong register may be named as an operand.

(7) The wrong memory variable or address may be named as an operand.

(8) The incorrect radix may be used for data or address value.

(9) An incorrect addressing mode may be invoked.

(10) Program self-modification (usually proscribed by good practice, and in any event avoidable through hardware and system software choice) may be performed incorrectly.

An error in the program may produce an end result (output or state) which is detected by a user. The nature of the detected error may be useful in determining where in the
program a correction is needed. This is not always the case, particularly if "markers" in the program--lines in the program that produce comments in program output to show what part of the program that execution is occurring at a particular point of time--are not used liberally.

According to Beam (GMU working paper, 7/3/87) the following five items (probably incomplete) are the most common ultimate effects of errors embedded in a faulty program.

(1) There is infinite looping in the program execution.

(2) The program executes, but output is observed to be incorrect.

(3) In time-sharing systems and in some multi-tasking systems, there may be a partition abort when the system hardware or software detects evidence of erroneous behavior of a program execution.

(4) For some errors such as division by zero in a single-tasked system, there may be a hardware invoked abort.

(5) There may be damage to system hardware or destruction of disk storage contents (a rare but possible event, occurring, for example, with I/O devices that lack protection against rapid and continued operation of mechanisms intended for intermittent operation).

In summary, even a small deviation at the onset of error may lead to program or system crashes. It is not only the
nature and extent of an error that is important; but where in
the program that the error occurs which determines how serious
the consequences of that error will be.

Section 2.3:  Error Latency\textsuperscript{11} and Detection

As Beam (GMU working paper, 7/3/87) writes,

Programming errors are seldom immediately observable... The difficulty of eliminating errors is closely related
to the frequency with which they become apparent. We use
the term latency here with its usual meaning: the delay
between introduction of an error and its appearing during
program test or operation.

Beam (GMU working paper, 7/3/87) notes five degrees of
latency, as follows.

(1) The error is observable (though perhaps not
observed) on each execution.

(2) The error is immediately observable with particular
combinations of input data. (Beam notes that these are the
errors for which most software test procedures are developed.)

(3) The error is observable only following execution of
the program.

(4) The error is triggered by external system operation
or incidental data (e.g. system date), rather than by or in
addition to input data.

\textsuperscript{11}"Error latency" as used here is synonymous with observation
of a "software failure."
(5) The error is triggered by unexpected hardware response or timing of the computer on which the program is operating. (Beam notes that "Hardware does play a role in errors, particularly in time-dependent code.")

Once an error latency is detected, the portion of the program(s) giving rise to it must be isolated. Beam (GMU working paper, 7/3/87) presents five ways that the source of an error may be determined.

(1) The source may be deducible from examination of normal program output.

(2) The source may be deducible by executions using available test input data.

(3) The source may be found by stepwise tracing with recording and comparison of register and memory contents.

(4) The source may be found by modification of dynamics of system or external elements, or by simulation of external elements. (Beam notes that..."Errors in time-dependent program execution are among the most difficult to locate. A typical problem is a set of communicating computers, each with its own timeouts and other time-dependencies.")

(5) The source may be deducible only by examination of the requirements and the conceptual approaches taken to satisfy them.

Beam notes that on rare occasions none of the above
approaches taken to determine the source of the error latency will succeed, and when such is the case, unorthodox heroics must be utilized.

Section 2.4: Underlying Causes of Software Errors

Understanding why an error occurs sometimes gives a clue as to how it can be avoided. Beam (GMU working paper, 7/3/87) gives an unstructured list of eleven occasions by which an error might arise. The list is as follows.

(1) The programmer’s memory may lapse (as a result of fatigue, sleeplessness, distraction or confusion).

(2) The programmer may generate typographic or spelling errors.

(3) There may be transcription errors. (Beam notes that transcription errors are common when handwritten copy is to be transcribed into computer readable form.)

(4) There may be program logical misconstruction, which particularly arises from complex algorithmic construction.

(5) The program logic may be translated incorrectly into code.

(6) The programmer may have omitted an essential step. (Beam notes that this is a common error, often arising when working through code of even simple logical constructs.)

(7) The programmer may have added a spurious or
incorrect step.

(8) The programmer may misunderstand parameters or operation of calling, particularly when calling routines developed by others.

(9) The programmer may misinterpret operation of language translator or be unaware of a defect in it. (Beam notes that this source of error would seem to be unlikely, yet it continues to happen.)

(10) The programmer may have received incomplete or incorrect information on calling or called routines.

(11) The programmer may be unaware of correct scope or other attributes of that data which would be provided to the program for input or control.

Section 2.5: Related Issues

Sometimes, when "fixing" a software error, new errors are introduced. Generally, the more complex and/or extensive the program changes required for the correction, the more likely new errors are introduced.

A correction may involve (1) only one module, (2) several closely related modules, (3) modest global changes, (4) application redesign, or (5) repair of a software "error" in one program such that operation of other programs or hardware is changed. Each of these changes involves different
opportunities for introduction of new errors.

Part of the problem with software errors is that they are of eclectic variety. Rarely does one have access to "a complete, archetypical, 'handbook' or 'ideal' structure for any given operation." (Beam--GMU working paper, 7/3/67).

A programmer is ever treading on virgin or near-virgin ground--with respect to new developments or software changes--and past mistakes made by one programmer are frequently repeated by another programmer; this can be so, even if both programmers are involved in creating different portions of the same software package.

An attempt to reduce the incidence of software errors may be likened to the creation of standardized algorithmic modules that can be taken "off the shelf" and plugged into the program under development, as the need arises. This is already being done in the realm of numerical methods and many other general purpose software applications. Perhaps the message handling and data handling focus of C'I systems, which are usually custom-programmed in large part, can benefit from such treatments. Doing such, with the repetitiveness and overkill concerning unrequired elements that would be involved, could notably increase program size. However, the ever increasing capacity of computers makes that consideration less important and may be offset by the greater integrity that would be
realized.

Constraining program structures by defining those particularly well-suited to the application has much likelihood of success in the area of C'I software programming and should receive more attention than it appears to have at the present time.

A serious problem is in the area of dealing with and controlling external interfaces to the program. As Beam (GMU working paper, 7/3/87) writes: "Errors associated with external interfaces are becoming more prevalent, as more applications become communications-based." Such is indeed applicable to the message-handling and data handling orientation of C'I software programs.

Software errors, by examination of their variety, leads one to a realization that they have few if any common cause(s). Hence, tracking one or a few types of errors, while fruitful at times, is not likely to lead to a universal cure-all. Instead, one should look for general principles upon which to debug either updates to C'I software packages or new C'I software packages. At best, broad guidelines will benefit the debugging programmers in the C'I environment.

To an extent, this means that a database of previously developed C'I systems software can be meaningfully tracked to statistically predict the integrity of new software programs
to enhance understanding of what will be necessary to bring a C'I software addition up to desired integrity. In essence, there will be only a slight learning curve involved in the programming effort. Statistical methods based on prior experiences will have theoretical significance in using prior software.

To be specially noted, there are, in fact, few recommended programming methods which are used explicitly to address particular areas of application, except for programming languages which have been adopted (e.g., COBOL for "business" applications). New techniques such as object oriented design are applied with zeal in any application by progressive programmers, while conservative programmers stick to techniques with which they are familiar.

A related issue which will be addressed in Chapter 4 is stakeholder analysis—or the consideration of the vested interests (stakes) the parties involved in the software development have. One expects that the greater the interests certain parties have in a particular software development effort, greater care probably will be exercised in developing the software, resulting in somewhat greater integrity of the software program.
Section 2.6: Remarks

Improvements in C'I software integrity may come about by three avenues suggested by Beam (GMU working paper, 7/3/87). They are as follows.

(1) The improvement may come about via very-high-level languages and program generators. General purpose programming languages are popular, but Beam notes:

very-high-level and probably application-limited languages permit less opportunity for the programmer to stray. The same is true of "program generators" which are similar but often non-procedural in part.

(2) The improvement may come about by increasing redundancy in program description.

(3) The improvement may come about by hardware assistance for quality assurance.

(Since Beam's paper, object-oriented design has been shown to be an effective way to partition programs at detailed levels and clearly should reduce programming errors, as well.)

The relevancy of points (2) and (3) above have been alluded to earlier in this chapter under the auspices of slightly different objectives.

Through this chapter, a greater understanding of software errors occurring for C'I systems has been fostered. And, of primary importance, a way for meaningful statistical analysis of the occurrence of C'I software errors has been introduced,
which is not by directly analyzing only a particular program itself, but by also analyzing mature C²I software programs of a similar nature that precede the program for which a software error integrity prediction is being sought.
Chapter 3

Summary of Software Integrity Models
To Be Used in This Project

Section 3.1: Introduction

This chapter introduces the reader to the two main software integrity models that currently are under study but already have achieved a notable degree of acceptance by experts in the field.

Execution time, which was defined in Section 1.3 as the time that a processor actually spends in executing the instructions for a program under study, is the metric cornerstone of our analysis. The importance of execution time is that it now has been generally accepted that the models are superior when they are based on execution time as opposed to calendar or clock time (where these are also as defined in Section 1.3). Hence the primary focus of the models in this chapter is on execution time. Later in the project, a crude conversion of execution time into calendar time will be presented and discussed.

There is an implicit assumption involved that contends the executions will eventually include all conditions and combinations of data for which the software is designed to be valid. When N processors run the same programs and data, the
results may be no more meaningful than if they are run one
time or N times by one processor.\textsuperscript{12} Although this non-
stochastic environment defies rigorous proofs, in engineering
one must make progress, rigor or no. We shall subscribe to
execution time models with the understanding that their
validity depends on steps being taken to execute under a full
range of conditions for which it is intended to be used.

Because this paper will deal \textit{indirectly} with determining
calendar time, we need not be concerned with the component
parameters for calendar time. The crude and somewhat
imprecise method of converting execution time to calendar time
that we will use will be adequate to put reasonable bounds on
resource expenditures or costs.

Our primary interest in model selection is to determine
when to graduate a program from alpha test to beta test; and
from beta test to distribution to the field where its intended
actual use will take place.

The parameters of the two models upon which we will focus
need to be predicted before failure data becomes available and
when systems engineering studies are needed concerning project
phases. The basic execution time model (see Section 3.2)

\textsuperscript{12}The exception would be if each processor were concurrently
executing a different set of programs. Even in this case only
those aspects of the software under test which are environment-
sensitive would be more thoroughly exercised.
already has procedures for predicting the execution time's component parameter. The logarithmic Poisson model (see Section 3.3) does not yet have such procedures developed for it. Currently, prediction accuracy is limited because there is not much data that has been collected regarding this and little work has been undertaken in this area. A practical limitation is that software technology is in rapid evolution and this has raised doubts about the meaning of such predictions when the projections exceed a few months or years.

A point to keep in mind is that system engineering studies can realize substantial cost savings on a wide variety of projects; hence, even imprecise forecasting of software integrity can have an enormous positive return obtained from applying what is currently known about software integrity models. Limited accuracy should not be labeled a handicap when studies solely use relative values for quantities that are needed; limited accuracy should be regarded as preferable to not making a study at all, unless its conclusions are less correct than estimates made by experts--on the back of an envelope.

Estimation generally is accomplished for either a

---

13 A deterrent to vigorous application is that few software developments are assigned quantitative integrity requirements. People are still very seat-of-the-pants, qualitatively oriented.
subsystem or system test or during the operational phase. Statisticians believe that, typically, estimation is more accurate than predication.

The operational phase of software for C'I systems usually consists of a series of releases. That is because the software is modified to make it more comprehensive and useful. What is added in subsequent releases may bear little resemblance to what is already there.

The use of evolutionary releases avoids holding up a finished module of code that has evolved to an acceptable level of integrity for immediate use. Without immediate release of completed modules, ten years or more might pass before any part of code, from development of the first module, will be released to the field—a totally unacceptable lapse of time.

Fortunately, for the modules inherited without change from earlier release times, software faults are not normally introduced to the upgraded software with the possible exception of changes in the data they receive from new or altered modules. The pre-existing code has been largely debugged in the previous application. However, there may also be a problem with the interfaces.

Given the high integrity usually achieved for software for mature pre-existing modules of C'I software, the quantity
and type of faults in mature inherited code are generally ignored and are rarely repaired. It is much simpler to change new or revised modules so as to avoid evoking incorrect responses from mature modules.

Nevertheless, a purpose of this study is to be able to determine (via methods proposed for use in beta testing) if pre-existing modules for C'I software contain a sufficient number of faults such that debugging of earlier modules should be undertaken as a part of the new release(s).

In the system test phase, resource usage parameters concern the resources that are required for failure identification along with correction. These parameters concern—for example—required resources per execution time hour and per failure.

At present, the best estimation recourse is to establish the resource usage parameter values by data collection for a particular project and generate a least squares fitting of the data.

The resource usage parameter values may be related to:

1. batch debugging as opposed to interactive debugging,
2. the availability of debugging aids,
3. the particular computer used,
4. the particular language and language translator used,
(5) the overhead for administrative contributions and documentation which are associated with corrections.

Notable uncertainties occur for resource usage parameters to a much greater extent than for the other typically used parameter values. According to Musa et al. (1987):

The range of uncertainty depends on a number of factors. The effects of these uncertainties on the prediction of the data for reaching the failure intensity objective can be established by calculating this data for different sets of values.

In practice, failures may be broken down into categories comprising severity groups. The classifications for severity of the failures are usually made by more than a single person. Hence, it is desirable to have objective rather than subjective classification rules. Quantitative criteria, if possible, are preferred; and regardless of whether judgment—even by quantitative criteria—is subject to error, the existence of such rules likely enhances understanding and reduces differences.

According to Musa et al. (1987):

There appear to be three principal approaches to handling severity classifications in estimating reliability and related quantities:
1. Classify the failures and estimate failure intensity and other quantities separately for each class.
2. Classify the failures, but lump the data together, weighting the time intervals between failures of different classes according to the severity of the failure class.
3. Classify the failures, but ignore severity in
estimating the overall failure intensity. Develop failure intensity for each failure class by multiplying the overall failures occurring in each class.

The first is the most precise, but suffers from working with small samples of failures, and there is already a short supply of data.

On occasion, working with categories of failures may be useful. The categorization of failures enhances the possibility that software integrity goals can be achieved at economic costs and with reasonable schedules.

Section 3.2 will present the basic execution time model. This model is an example of finite failures category models, and is the most useful known example of that category.

Section 3.3 will present the logarithmic Poisson execution time model. This model is an example of infinite failures category models, and is the most useful known example of that category. (As we shall see, this model is most useful for C'I software undergoing beta test and when the software has been released to the field for actual use—if, during each phase, there will be little if any maintenance performed.)

Section 3.2: The Basic Execution Time Model

Musa et al. (1987) gives five reasons for selecting the basic execution time model:
1. It generally predicts satisfactorily.
2. It is simple and easy to understand.
3. It is the model that is most thoroughly developed and has been most widely applied to actual projects. Several programs have been developed to aid in implementing it.
4. Its parameters have a clear interpretation and can be related to information that exists before the program has been executed. The information includes characteristics of the development environment. The foregoing property is important for system engineering studies.
5. It can handle evolving systems (systems that change in size) by adjusting the failure times to estimates of what they would have been if all the code had been present...

The basic execution time model is associated with an operational profile that is uniform, as has been demonstrated by Downs (1985) and Trachtenberg (1985).

However, this model is notably tolerant of a substantial amount of nonuniformity, as has been demonstrated also by Downs (1985) and by Trachtenberg (1985).

Denoting execution time by $\tau$, the basic execution time model can be written:

$$\mu(\tau) = \nu_0 \left[1 - \exp\left(-\frac{\lambda_0}{\nu_0}\right)\right],$$

where $\mu(\tau)$ is cumulative number of failures, $\nu_0$ is total failures to be experienced in infinite time, and $\lambda_0$ is the initial failure intensity denoting number of failures per CPH-hr. at the start of execution.

The basic execution time model denotes an arithmetic change for failure intensity with regard to the experienced
number of mean failures.

For the basic execution time model, the failure intensity, denoted by $\lambda$, as a function of number of failures observed, is:

$$\lambda(\mu) = \lambda_o \left[ 1 - \frac{\mu}{\nu_o} \right].$$

Section 3.3: The Logarithmic Poisson Execution Time Model

An operational profile that is highly nonuniform generates a failure intensity history for which the logarithmic Poisson execution time model is more suitable than the basic execution time model. The word "highly" is used because the basic execution time model can tolerate a substantial degree of nonuniformity.

The logarithmic Poisson execution time model is especially useful in the case where some parts of the software are executed with frequency much greater than others.

The failure intensity for the logarithmic Poisson execution time model eventually decreases very slowly. This is a result of notably infrequent execution of input states that have faults remaining in them. However, in system tests the operational profile may be deliberately altered by test planners to reduce test execution time of the system in order
to expose problems via reducing execution time needed to observe rare failures; that is, failures that would occur rarely during actual operation of the software.

Again denoting execution time by \( \tau \), the logarithmic Poisson execution time model can be written:

\[
\mu(\tau) = \frac{1}{\theta} \ln (\lambda_0 \theta \tau + 1)
\]

where \( \mu(\tau) \) is number of failures, \( \theta \) is the failure decay parameter, and \( \lambda_0 \) is the initial failure intensity of number of failures per CPU-hr. at the start of the program.

In general,

\[
\mu(\tau) = \sum_{n} \frac{1}{\theta_n} \ln (\lambda_0 \theta_n \tau + 1)
\]

where \( n \) is over disjoint subsets of the software whose integrity is over different classes. A similar general expression exists for the basic execution time model.

To determine \( \theta \), which is the failure decay parameter, plot the failure intensities' natural logarithm against mean failures experienced. Then, by transforming

\[
\lambda(\mu) = \lambda_0 \exp(-\theta \mu)
\]

we see that \( \theta \) is the magnitude of the slope of the line we have plotted.

The logarithmic Poisson execution time model denotes a geometric change for failure intensity with regard to the
experienced number of mean failures.

When software has been released and becomes operational where no features are added and no repairs are made between releases, the failure intensity will be constant.\textsuperscript{14} Then both the execution time model and the logarithmic Poisson execution time model, as nonhomogeneous Poisson processes, simplify to become homogeneous Poisson processes with the failure intensity for its parameter. Here, failure intervals are described as an exponential distribution. The integrity \( R \) in conjunction with \( \lambda \), the failure intensity parameter, are related by

\[
R(\tau) = \exp(-\lambda \tau).
\]

The reader should note that the integrity is dependent on failure intensity and the execution time's period.

It is to be noted that most of the above and the immediately preceding section holds true for the basic calendar time model vs. the logarithmic Poisson calendar time model, where "calendar time" has been substituted for "execution time" as a concept and with regard to the calendar time parameter.

\textsuperscript{14}Of course, if the error rate is unacceptable to the user, the result may be that the program is modified—if possible—or abandoned if "work around" is not possible. "Work around" is taking actions to overcome the existence of the errors.
Section 3.4: Selection of the Model

There are two categories of probabilistic models used for application of determining software integrity, as mentioned earlier in this chapter.

One is the finite failures category, which includes the basic execution time model.

The other is the infinite failures category, which includes the logarithmic Poisson execution time model.

The number of failures in infinite time is unbounded in the infinite failures category models, but is bounded in the finite failures category models.

According to Musa et al. (1987), the capability and applicability of the basic execution time model are generally superior to those of other published models. Both the basic execution time model and the logarithmic Poisson execution time model are preferred in terms of simplicity, with the logarithmic Poisson execution time model being second-best to the basic execution time model with regard to capability. But, the logarithmic Poisson execution time model is superior to other models with regard to the characteristic of estimation capability. Musa et a. (1987) write:

The logarithmic Poisson is superior in predictive validity. Although the basic model is not as good as the

15"Estimative" is the correct term for here.
logarithmic Poisson model in this respect, the difference is not significant after about 60 percent of the way through the test period. Thus these two models appear to be the two models of choice. One possible approach is to use the basic model for pretest studies and estimates and for periods of phased integration. Switch to the logarithmic Poisson model when integration is complete and the program is stable.

The optimum choice selection between the basic execution time model and logarithmic Poisson execution time model is presented in Table 2.

Finite failures models are credible from the viewpoint of situations where there is a vigorous repair program resulting in removal of errors well before major additions or revisions, which introduce new errors. Infinite failures models are well suited for the occasion of slow or no repair occurring. It is important to note that infinite failures need not imply infinite faults. A situation of infinite faults can occur; for example, with the case of continual introduction of new faults with repair. If a program were executed for infinite time, the infinite failures case is credible; but in practice software has a limited useful lifetime.

Practically speaking, a related circumstance has to do with whether people think they can locate and remove all errors. Since program errors depend on operating system and other environments—factors which also change—these may be
<table>
<thead>
<tr>
<th>Purpose of application or existence of condition</th>
<th>Basic</th>
<th>Logarithmic Poisson</th>
</tr>
</thead>
<tbody>
<tr>
<td>Studies or predictions before execution</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Studying effects of software engineering technology (through study of faults)</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Program size changing continually and substantially</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Highly nonuniform operational profile</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Early estimation validity important</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

\[\textsuperscript{16}\text{The above table is adapted from Musa et al. (1987).}\]
continual sources of failure. However, the errors will follow the onset of changes in program usage, environment, or data values.
CHAPTER 4
CREATING AND UPDATING A DATA BASE SAMPLE

Section 4.1: Introduction

In Section 4.2, analysis of variance (ANOVA) and a multiple comparison method show how a high quality, appropriate-for-analysis data base sample can be selected from data captured in a list of mature candidate C'TI software. The number of distinct failures, by category, from completed alpha and beta testing is the key for doing this. This yields a data base sample that is very appropriate for predicting the execution and calendar time needed for the alpha test of new C'TI software.

This data base sample is applicable to predicting the execution and calendar time necessary for beta testing, as well. However, the length and quantity of beta testing will not be determined solely from this data base sample, so lapses that may occur in forecasting for amount of beta testing that is necessary to achieve a given level of integrity will not be critical.

Maintenance of the data base is covered in Section 4.3. Section 4.4 contains analysis of stakeholder profiles, which is useful for making qualitative adjustments to the data base sample for fine tuning of the prediction(s).
Throughout this chapter—in the text, figures and tables—"m" denotes the number of programs in the starting list (consisting of programs having five thousand program lines of code or more, the reason for which will soon be explained in detail), and "n" denotes the number of failure categories being considered.

Section 4.2: Selection of Mature Modules of Software for the Data Base Sample

The purpose of this section is to show how a high quality, appropriate-for-analysis data base sample can be selected from a list of mature candidate software by application of two-factor analysis of variance. Only total, distinct numbers of failures—by category—experienced during alpha and beta test will be used.

First, take the collection of m programs—written in the same programming environment and language as the program being tested—and count the number of program lines of code for each program (see Section 5.1 for more on this). This number of program lines of code for each program is denoted as "g."

Then, for the categories of failures under consideration, each quantity of failures by category is scaled such that the lines of code for the mature programs are comparable to the program having G lines of code—where "G" is the quantity of
program lines of code for the program for which we are making the prediction(s). (Hence, all programs are to be considered as though they have G program lines of code.) The number of actual failures h for the respective program actually having g program lines of code will have the following scaled number of failures H—by category—found by the following equation.

\[ H = h \times G / g. \]

In Figure 4, it is seen how the number of failures h is related to unscaled program line size g. Because complexity increases with program line size, larger programs tend to have a larger amount of failures per program line of code—in total and by category. In Figure 4 is also seen how the scaled programs relate to a program of program line size G. Note that very small programs tend to be a problem for this method. Therefore, since the models used in this report are appropriate mainly for programs having more than five thousand program lines of code, programs of less than five thousand lines of code are deleted from the starting list.

After we note the collections of scaled programs for which there will be no significant statistical difference in number of failures—which is a goal of this analysis and which will be made clearer later in this section—we are going to want to restrict the data base sample to programs that are actually close in size to the program for which prediction(s)
The solid line is unscaled number of failures $h$, by number of program lines of code $g$. The dashed line is scaled number of failures $H$ by number of program lines $g$ related to program of number of program lines of code $G$ which is the program for which the predictions are to be made. The formula relating $h$ to $H$ is: $H = h \times G/g$. 

Figure 4
in number of failures are to be made, to prevent or reduce the impact of outliers that are present in the data base starting list. We use all of the programs for the first step of the analysis--except for programs with less than five thousand program lines of code--because we thus get a better estimate of the variance to generate a more accurate collection of programs for which analysis of variance's multiple comparison methods show no significant statistical difference in number of failures.

The scaling of failures is to be done over the roster of n categories of failures for each of the mature software programs. Then the collection of H’s (x_{ij}’s)--by respective category--are to be collected into a table (see Table 3).

Following Devore (1982), the two-factor analysis of variance table on the next page is assigned double subscripts--i and j--identifying the random variables and the observed values. The notation below is typical for analysis of variance studies, and specifically comes from Devore (1982).

X_{ij} denotes the scaled random variable for number of failures by a specific category of failures. Note that X_{ij} denotes the random variable measurements for factor A held at level i and factor B held at level j.

x_{ij} is the observed value for X_{ij}.
Table 3

Factor B

Adjusted failures, by category

<table>
<thead>
<tr>
<th>Category #1</th>
<th>Category #2</th>
<th>Category #(n-3)</th>
<th>Category #(n-2)</th>
<th>Category #(n-1)</th>
<th>Category #n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program #1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C31 Software</td>
<td></td>
<td>X_{ij}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #(M-2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #(m-1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program #m</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\( \bar{X}_i \) denotes the average measurements realized when factor A is held at level \( i \); or, more specifically

\[
\bar{X}_i = \frac{1}{n} \sum_{j=1}^{n} X_{ij}.
\]

Similarly, \( \bar{X}_j \) denotes the average measurements realized when factor B is held at level \( j \); or

\[
\bar{X}_j = \frac{1}{m} \sum_{i=1}^{m} X_{ij}.
\]

\( \bar{X} \) is the grand mean; or,

\[
\bar{X} = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} X_{ij}.
\]

To proceed, we need to specify a model. And a standard, often used simplifying assumption is

\[
X_{ij} = \alpha_i + \beta_j + \mu + \epsilon_{ij}, \quad i=1, \ldots, m; \quad j=1, \ldots, n
\]

where for the above we have assumed the existence of \( m \) parameters \( \alpha_1, \alpha_2, \ldots, \alpha_m \) and \( n \) parameters \( \beta_1, \beta_2, \ldots, \beta_n \); where \( \hat{\alpha}_i = \bar{X}_i - \bar{X} \), \( \hat{\beta}_j = \bar{X}_j - \bar{X} \) and \( \hat{\mu} = \bar{X} \).

Now, we assume we have \( \mu_{ij} \) which can be written as \( \mu_{ij} = \alpha_i + \beta_j \).

The above model is called the "additive model" and is widely used in the field of psychology. It is robust and when this assumed model is correct, the difference of mean responses for factor A (program) at levels \( i \) and \( i' \) with factor B (failure category) held at level \( j \) is

\[
\mu_{ij} - \mu_{i'j} = (\alpha_i + \beta_j) - (\alpha_{i'} + \beta_j) = \alpha_i - \alpha_{i'}.
\]
for which we note this is independent of the jth level of the second factor (factor B). A similar result holds for \( \mu_{ij} - \mu_{ij} \).

The null hypothesis we want to test is:

\[
H_{0A}: \quad \alpha_1 = \alpha_2 = \ldots = \alpha_m = 0
\]

versus \( H_{1A} \): at least one \( \alpha_i \neq 0 \); again where \( \hat{\alpha}_i = \bar{x}_{i.} - \bar{x}_{..} \). If the null hypothesis is true, we include all the involved programs being considered to be in the data base sample (before we delete programs from the data base sample because their line quantity is unacceptably greater or lesser than the program whose failures we desire to predict). If the null hypothesis is rejected, we want to know which programs we are to exclude from the data base sample.

For testing \( H_{0A} \) versus \( H_{1A} \) we use the F test statistic \( F_A \)

\[
F_{A} = \frac{MSA}{MSE}
\]

where the rejection region is \( F_A \geq F_{\alpha, m-1, (n-1)(m-1)} \).

Here we note that MSA is defined to be mean square of factor A; MSE is defined to be mean square of error; SSA is defined to be sum of squares of factor A; and SSE is defined to be sum of squares of error. In terms of mathematics notation and formulation, these are observed to be:

\[
SSA = \sum_{i=1}^{m} \sum_{j=1}^{n} (\bar{x}_{i.} - \bar{x}_{..}) = \frac{1}{n} \sum_{i=1}^{m} \bar{x}_{i.}^2 - \frac{1}{m \times n} \bar{x}_{..}^2
\]
\[ \text{SSE} = \sum_{i=1}^{m} \sum_{j=1}^{n} (x_{ij} - \bar{x}_i - \bar{x}_j + \bar{x})^2 \]

where we then have

\[ \text{MSA} = \frac{\text{SSA}}{(m-1)} \]
\[ \text{MSE} = \frac{\text{SSE}}{(m-1)(n-1)} \]

When \( H_{\alpha} \) is rejected, Tukey's procedure may be used to identify the significant differences for the levels of factor A, which is our focus of interest. (This can also be done for factor B, but this is not currently of interest to us.)

For comparing factor A levels, obtain \( Q_{\alpha, m, (m-1)(n-1)} \). (Table 4 is a sample table for \( Q \). The "\( n \)" in the table is \((m - 1) \ast (n - 1)\); which is a different definition from that of as used elsewhere in this report. And, "\( \alpha \)" is the level of significance of the test; which is a different definition from that of \( \alpha \) as used in some other portions in this chapter.)

Then we are to arrange the sample means of \( x_{i.} \) in increasing order, and underscore the pairs which differ by less than \( w \), where \( w = Q_{\alpha, m, (m-1)(n-1)} \ast \sqrt{\text{MSE}/n} \).

An appropriate size for \( \alpha \) in the determination of \( Q \) above is .05 or .1.

For greater applicability to this particular situation, we delete all programs from the sample where the actual program size is greater or lesser than ten or fifteen percent of the difference from the size of the program whose failures
### Table 4

**Table A.8 Critical Values $Q_{a,n,v}$ for the Studentized Range Distribution**

<table>
<thead>
<tr>
<th>$a$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.05</td>
<td>3.04</td>
<td>4.60</td>
<td>5.22</td>
<td>5.67</td>
<td>6.03</td>
<td>6.33</td>
<td>6.68</td>
<td>6.90</td>
<td>7.17</td>
<td></td>
</tr>
<tr>
<td>0.01</td>
<td>3.70</td>
<td>4.98</td>
<td>5.90</td>
<td>6.42</td>
<td>6.81</td>
<td>7.12</td>
<td>7.43</td>
<td>7.71</td>
<td>7.91</td>
<td>8.12</td>
</tr>
<tr>
<td>0.001</td>
<td>4.40</td>
<td>5.34</td>
<td>6.20</td>
<td>6.80</td>
<td>7.33</td>
<td>7.81</td>
<td>8.28</td>
<td>8.74</td>
<td>9.19</td>
<td>9.58</td>
</tr>
<tr>
<td>0.0001</td>
<td>5.32</td>
<td>6.04</td>
<td>6.70</td>
<td>7.40</td>
<td>7.99</td>
<td>8.54</td>
<td>9.05</td>
<td>9.55</td>
<td>9.99</td>
<td>10.40</td>
</tr>
<tr>
<td>0.00001</td>
<td>6.38</td>
<td>7.02</td>
<td>7.62</td>
<td>8.26</td>
<td>8.83</td>
<td>9.36</td>
<td>9.86</td>
<td>10.35</td>
<td>10.80</td>
<td>11.23</td>
</tr>
<tr>
<td>0.000001</td>
<td>7.54</td>
<td>8.07</td>
<td>8.62</td>
<td>9.18</td>
<td>9.71</td>
<td>10.23</td>
<td>10.73</td>
<td>11.22</td>
<td>11.69</td>
<td>12.14</td>
</tr>
<tr>
<td>0.0000001</td>
<td>8.83</td>
<td>9.32</td>
<td>9.83</td>
<td>10.37</td>
<td>10.89</td>
<td>11.39</td>
<td>11.90</td>
<td>12.38</td>
<td>12.85</td>
<td>13.31</td>
</tr>
</tbody>
</table>

*Source:* This table is adapted from Table 29 in *Biometrika Tables for Statisticians*, vol. 1, 3rd ed. by E. S. Pearson and H. O. Hartley (eds.). Reproduced with the kind permission of the Trustees of Biometrika, 1966.

18 This table was excerpted from Devore (1982).
Table 4 (continued)

<table>
<thead>
<tr>
<th>m</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>α</th>
<th>ν</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.32</td>
<td>7.47</td>
<td>7.60</td>
<td>7.72</td>
<td>7.83</td>
<td>7.93</td>
<td>8.03</td>
<td>8.12</td>
<td>8.21</td>
<td>0.05</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10.70</td>
<td>10.89</td>
<td>11.08</td>
<td>11.24</td>
<td>11.40</td>
<td>11.55</td>
<td>11.68</td>
<td>11.81</td>
<td>11.93</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.79</td>
<td>6.92</td>
<td>7.03</td>
<td>7.14</td>
<td>7.24</td>
<td>7.34</td>
<td>7.43</td>
<td>7.51</td>
<td>7.59</td>
<td>0.05</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9.48</td>
<td>9.65</td>
<td>9.81</td>
<td>9.95</td>
<td>10.08</td>
<td>10.21</td>
<td>10.32</td>
<td>10.43</td>
<td>10.54</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.43</td>
<td>6.55</td>
<td>6.66</td>
<td>6.76</td>
<td>6.83</td>
<td>6.94</td>
<td>7.02</td>
<td>7.10</td>
<td>7.17</td>
<td>0.05</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8.71</td>
<td>8.86</td>
<td>9.00</td>
<td>9.12</td>
<td>9.24</td>
<td>9.35</td>
<td>9.46</td>
<td>9.55</td>
<td>9.65</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.18</td>
<td>6.29</td>
<td>6.39</td>
<td>6.48</td>
<td>6.57</td>
<td>6.65</td>
<td>6.73</td>
<td>6.80</td>
<td>6.87</td>
<td>0.05</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8.18</td>
<td>8.31</td>
<td>8.44</td>
<td>8.55</td>
<td>8.66</td>
<td>8.76</td>
<td>8.85</td>
<td>8.94</td>
<td>9.03</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.98</td>
<td>6.09</td>
<td>6.19</td>
<td>6.28</td>
<td>6.36</td>
<td>6.44</td>
<td>6.51</td>
<td>6.58</td>
<td>6.64</td>
<td>0.05</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>7.76</td>
<td>7.91</td>
<td>8.03</td>
<td>8.13</td>
<td>8.23</td>
<td>8.33</td>
<td>8.41</td>
<td>8.49</td>
<td>8.57</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.83</td>
<td>5.93</td>
<td>6.03</td>
<td>6.11</td>
<td>6.19</td>
<td>6.27</td>
<td>6.34</td>
<td>6.40</td>
<td>6.47</td>
<td>0.05</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7.40</td>
<td>7.60</td>
<td>7.71</td>
<td>7.81</td>
<td>7.91</td>
<td>7.99</td>
<td>8.08</td>
<td>8.15</td>
<td>8.22</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.71</td>
<td>5.81</td>
<td>5.90</td>
<td>5.98</td>
<td>6.06</td>
<td>6.13</td>
<td>6.20</td>
<td>6.27</td>
<td>6.33</td>
<td>0.05</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7.25</td>
<td>7.36</td>
<td>7.46</td>
<td>7.56</td>
<td>7.65</td>
<td>7.73</td>
<td>7.81</td>
<td>7.88</td>
<td>7.95</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.61</td>
<td>5.71</td>
<td>5.80</td>
<td>5.88</td>
<td>5.95</td>
<td>6.02</td>
<td>6.09</td>
<td>6.15</td>
<td>6.21</td>
<td>0.05</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7.06</td>
<td>7.17</td>
<td>7.26</td>
<td>7.36</td>
<td>7.44</td>
<td>7.52</td>
<td>7.59</td>
<td>7.66</td>
<td>7.73</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.33</td>
<td>5.43</td>
<td>5.51</td>
<td>5.59</td>
<td>5.66</td>
<td>5.73</td>
<td>5.80</td>
<td>5.85</td>
<td>5.91</td>
<td>0.05</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>6.90</td>
<td>7.01</td>
<td>7.10</td>
<td>7.19</td>
<td>7.27</td>
<td>7.35</td>
<td>7.42</td>
<td>7.48</td>
<td>7.55</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.46</td>
<td>5.55</td>
<td>5.64</td>
<td>5.71</td>
<td>5.79</td>
<td>5.85</td>
<td>5.91</td>
<td>5.97</td>
<td>6.03</td>
<td>0.05</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>6.77</td>
<td>6.87</td>
<td>6.96</td>
<td>7.05</td>
<td>7.13</td>
<td>7.20</td>
<td>7.27</td>
<td>7.33</td>
<td>7.39</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.40</td>
<td>5.49</td>
<td>5.57</td>
<td>5.65</td>
<td>5.72</td>
<td>5.78</td>
<td>5.85</td>
<td>5.90</td>
<td>5.96</td>
<td>0.05</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6.66</td>
<td>6.76</td>
<td>6.84</td>
<td>6.93</td>
<td>7.00</td>
<td>7.07</td>
<td>7.14</td>
<td>7.20</td>
<td>7.26</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.35</td>
<td>5.44</td>
<td>5.52</td>
<td>5.59</td>
<td>5.66</td>
<td>5.73</td>
<td>5.79</td>
<td>5.84</td>
<td>5.90</td>
<td>0.05</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>6.56</td>
<td>6.66</td>
<td>6.74</td>
<td>6.82</td>
<td>6.90</td>
<td>6.97</td>
<td>7.03</td>
<td>7.09</td>
<td>7.15</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.31</td>
<td>5.39</td>
<td>5.47</td>
<td>5.54</td>
<td>5.61</td>
<td>5.67</td>
<td>5.73</td>
<td>5.79</td>
<td>5.84</td>
<td>0.05</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>6.47</td>
<td>6.57</td>
<td>6.66</td>
<td>6.73</td>
<td>6.81</td>
<td>6.87</td>
<td>6.94</td>
<td>7.00</td>
<td>7.05</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.27</td>
<td>5.35</td>
<td>5.42</td>
<td>5.50</td>
<td>5.57</td>
<td>5.63</td>
<td>5.69</td>
<td>5.74</td>
<td>5.79</td>
<td>0.05</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>6.41</td>
<td>6.50</td>
<td>6.58</td>
<td>6.65</td>
<td>6.73</td>
<td>6.79</td>
<td>6.85</td>
<td>6.91</td>
<td>6.97</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.23</td>
<td>5.31</td>
<td>5.39</td>
<td>5.46</td>
<td>5.53</td>
<td>5.59</td>
<td>5.65</td>
<td>5.70</td>
<td>5.75</td>
<td>0.05</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>6.34</td>
<td>6.43</td>
<td>6.51</td>
<td>6.58</td>
<td>6.65</td>
<td>6.72</td>
<td>6.78</td>
<td>6.84</td>
<td>6.89</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.20</td>
<td>5.28</td>
<td>5.36</td>
<td>5.43</td>
<td>5.49</td>
<td>5.55</td>
<td>5.61</td>
<td>5.66</td>
<td>5.71</td>
<td>0.05</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>6.28</td>
<td>6.37</td>
<td>6.45</td>
<td>6.52</td>
<td>6.59</td>
<td>6.65</td>
<td>6.71</td>
<td>6.77</td>
<td>6.82</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>5.18</td>
<td>5.23</td>
<td>5.32</td>
<td>5.38</td>
<td>5.44</td>
<td>5.49</td>
<td>5.55</td>
<td>5.59</td>
<td>0.05</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>6.19</td>
<td>6.26</td>
<td>6.33</td>
<td>6.39</td>
<td>6.45</td>
<td>6.51</td>
<td>6.56</td>
<td>6.61</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.00</td>
<td>5.08</td>
<td>5.15</td>
<td>5.21</td>
<td>5.27</td>
<td>5.33</td>
<td>5.38</td>
<td>5.43</td>
<td>5.47</td>
<td>0.05</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>5.93</td>
<td>6.01</td>
<td>6.08</td>
<td>6.14</td>
<td>6.20</td>
<td>6.26</td>
<td>6.31</td>
<td>6.36</td>
<td>6.41</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.90</td>
<td>4.98</td>
<td>5.04</td>
<td>5.11</td>
<td>5.16</td>
<td>5.22</td>
<td>5.27</td>
<td>5.31</td>
<td>5.36</td>
<td>0.05</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>5.76</td>
<td>5.83</td>
<td>5.90</td>
<td>5.96</td>
<td>6.02</td>
<td>6.07</td>
<td>6.12</td>
<td>6.16</td>
<td>6.21</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.81</td>
<td>4.88</td>
<td>4.94</td>
<td>5.00</td>
<td>5.06</td>
<td>5.11</td>
<td>5.15</td>
<td>5.20</td>
<td>5.24</td>
<td>0.05</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>5.66</td>
<td>5.73</td>
<td>5.79</td>
<td>5.84</td>
<td>5.89</td>
<td>5.93</td>
<td>5.97</td>
<td>6.01</td>
<td>6.05</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.71</td>
<td>4.78</td>
<td>4.84</td>
<td>4.90</td>
<td>4.95</td>
<td>5.00</td>
<td>5.04</td>
<td>5.09</td>
<td>5.13</td>
<td>0.05</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>5.44</td>
<td>5.50</td>
<td>5.56</td>
<td>5.61</td>
<td>5.66</td>
<td>5.71</td>
<td>5.75</td>
<td>5.79</td>
<td>5.83</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.62</td>
<td>4.68</td>
<td>4.74</td>
<td>4.80</td>
<td>4.85</td>
<td>4.89</td>
<td>4.93</td>
<td>4.97</td>
<td>5.01</td>
<td>0.05</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.29</td>
<td>5.35</td>
<td>5.40</td>
<td>5.45</td>
<td>5.49</td>
<td>5.54</td>
<td>5.57</td>
<td>5.61</td>
<td>5.65</td>
<td>0.01</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
we are trying to estimate. Then, of the remaining programs, we select the data base sample that has the most programs underscored as having no significant statistical difference among them.

Section 4.3: Updating the Data Base Sample

As more C3I software programs are completed and mature, the analysis in Section 4.2 is repeated with the new program(s) which are to be appended to the initial total data base (the collection of all programs) to create a new sample for study of the program currently under testing. (See Section 6.5 for a limitation on adding new programs to the initial data base.)

As more programs are added to the initial data base, the estimate of the variance will likely become more precise (accurate) so that the generated data base sample—the portion of the data base used for prediction—will be, as an expectation, more able to give quality, accurate predictions.

Section 4.4: Analysis of Stakeholder Profiles

The stakeholders include (1) the software developers, (2) the users, and perhaps (3) the procurement organization.

Stakeholders in the second and third groups may influence the number of failures experienced, in alpha and beta tests,
by overspecifying or underspecifying the content of the code. Overspecifying the code ties the hands of code developers and may lead them down wrong or inappropriate paths. Underspecifying what the code is supposed to accomplish could lead the programmers, for example, to make undesirable simplifying assumptions that will register as more discrepancies farther down the line, but have less failures show up in the alpha test period.

The stakeholders that have an interest and part in development of the code include:

(1) managers,
(2) systems engineers,
(3) quality assurance engineers,
(4) programmers,
(5) test teams,
(6) debuggers.

The calendar time and effort spent by each stakeholder above should be recorded for use in post-alpha and post-beta test analysis. This can lead to learning what is an appropriate contribution by each by life cycle phase—here regarded to concentrate on alpha test, beta test, and release to the field.

In analysis, it is unfortunate that we cannot just prorate the efforts of those responsible for the code
development to the size of the coding task itself. For instance, the optimal number of programmers for any size of code is typically given as around six programmers. (However, since clients cannot wait years for a viable program, a large program is usually developed by several, or many, teams of this small size.) The figure on the next page is a heuristic relationship between calendar time to complete a coding job and number of programmers involved. Over a significant range of job size, such a curve would have nearly the same shape. With too many programers, the required interaction communications will overwhelm development.

A comprehensive analysis of stakeholder profile is absolutely necessary to determine if a new program is an outlier in terms of number of failures experienced. This is a missing link in analysis and the gap in which it fits should be closed.

Fortunately, most C^I software development is for the U.S. and other governments and these procurers and users of the C^I software may be motivated to maintain records necessary for a valid and useful stakeholder profile analysis.
This section is roughly a rectangular hyperbola.

This section rises slowly.

Figure 5
CHAPTER 5
THE PHASE OF ALPHA TESTING

Section 5.1: Introduction

Alpha testing is testing carried out by the software developer. It begins with what are considered to be fully tested individual modules, and ends with the integrated software product believed suitable for putting into the hands of users for further testing. Alpha testing constitutes the first testing other than that done by the programmer(s) who actually wrote the program.

Section 5.2 concerns prediction for alpha testing and Section 5.4 concerns estimation for alpha testing in terms of execution time to achieve required levels of integrity. The terms "prediction" and "estimation" are defined in Chapter 1.

Section 5.3 concerns the occasion where the method to find a data base sample, as laid out in Chapter 4, fails to find even a single program on which to use the technique in Section 5.2.

To determine calendar time needed for the alpha testing, execution time can be roughly converted to calendar time as discussed in Section 5.5.

For definition of program lines to be counted to determine the size of the program, Musa et al. (1987) write:
Experience has indicated that inherent fault quantities are more closely correlated with source than object instructions. Hence we count deliverable executable source instructions developed during the project. Thus, we do not count drivers or support programs. Data statements, declarations, and comments are not counted. When the project involves program changes rather than program creation, we do not count instructions/statements that are left alone or deleted. The incidence of inherent faults associated with such statements is much less than that associated with those that are created or modified...In practice, the number of instructions is highly correlated with the number of lines of code. As the latter is easier to count, it is most often used.

Musa et al. (1987) leads us to believe this is representative of typical practice. The reader should be aware that the source line count for machine (assembly) language source code is from about 5 to 40 times larger than that for high-level language instructions. Hence, the particular language used must always be a parameter.

For the data sample, the scaled value(s) for number of failures in the individual programs are used. Also to be equivalently scaled are \( \nu_o \), \( \lambda_o \), and \( \omega_o \) for each program in the data base sample. Not to be scaled are \( K \), \( B \), \( r \) or \( I \). These variables are defined in the next section if they have not been previously defined.

Section 5.2: Prediction for Alpha Testing

We must predict two parameters for the basic execution time model, which is the model of choice for analysis of alpha
testing: (1) $\nu_0$, which is total failures to experienced in infinite time, and (2) $\lambda_0$, which is the initial failure intensity.

In this section, the analysis need not be done for failures by category. Greater accuracy is obtained by not diluting the data base sample at this time. In Section 5.4, some of the analysis will be done on failures by category; since Section 5.4 constitutes the safety net, little is lost by deferring failure by category analysis until then.

According to Musa et al. (1987), the fault reduction factor B is defined and determined as follows:

The fault reduction factor B is the ratio of net fault reduction to failures experienced as time of operation approaches infinity. Expected values are taken for both the faults and the failures. If this weren't done, the result would depend on the particular realization of input states executed rather than the operational profile. The total number of faults corrected is frequently larger than the net number. As a simple example, 100 failures may occur for a system. Let us assume that insufficient information is available to permit the faults that caused 2 of them to be found. However, 98 faults are removed. In the correction process, assume that 5 new faults are spawned. The total number of faults corrected is 98 but the net number is 93. The fault reduction factor B would be 0.93.

The conditions to be considered for estimating B are (from Musa et al. (1987)):

1. the degree to which relationships between failures that are the product of the same underlying fault are recognized,
2. detectability (the proportion of failures whose faults can be found, which depends on the amount of
information recorded about the failure),
3. the extent to which code inspection independent of
test is performed,
4. the degree to which relationships between faults
are discerned, and
5. the extent to which new faults are spawned during
fault correction.

The following procedure to determine \( \nu_0 \) and \( \lambda_0 \) is
recommended by Musa et al. (1987)\(^{19} \):

We let \( \nu_0 = \frac{\omega_0}{B} \)

where \( \omega_0 \) is the number of inherent faults, as determined by
the data base sample.

To get an expected value estimation for \( \nu_0 \), determine \( \frac{\omega_0}{B} \)
for each program in the data base sample directly used for
this analysis and calculate the mean of \( \frac{\omega_0}{B} \). Through this, we
will have found an unbiased estimator of the mean for \( \nu_0 \).

To find the expected value for \( \lambda_0 \), we let

\[ \lambda_0 = fK\omega_0, \]

where \( f = \frac{r}{I} \) and where \( f \) is the linear execution frequency of
the program in the data base sample; \( r \) is the average

\(^{19}\text{Musa et al. (1987) sometimes flies "fast and loose" with its}
sources, as is done here. But because this approach looks}
reasonable to this student, it--the technique--is presented here as}
correct without question.
instruction rate divided by the quantity of object instructions I in the program being predicted; and K is the fault exposure ratio relating failure intensity to the "fault velocity"—that is, K denotes the fraction of time the program execution code results in a failure.

K must be calculated from similar programs, which in our case are the programs in the data base sample—generated by the method in Chapter 4—used for prediction of the program characteristics being predicted.

According to Musa et al. (1987), to evaluate K from other projects which have the available data needed, we require four items of information:

1. a set of failure data sufficient to permit B to be estimated,
2. the fault reduction factor B,
3. program size in object instructions, and
4. average instruction execution rate.

If the failure data are not true execution time data, appropriate adjustments are performed to render them comparable to execution time data.

For each program in the data base sample, K is found to be

\[ K = \frac{1}{B_f \frac{\lambda_o}{\nu_o}}. \]

For each program in the data base sample, \( \omega_o \) has already been found. Hence, to get the expected value for \( \lambda_o \),
determine $K\omega_o$ for each element in the data base sample and take the mean of $K\omega_o$. That gives us an appropriate unbiased prediction of $\lambda_o$, according to the criteria we have set forth, which is to find the expected value of $\lambda_o$.

After predicting $\nu_o$ and $\lambda_o$, the basic execution time model can be used now to predict the amount of execution time expected to be necessary to reach a given failure intensity objective for alpha testing.

Section 5.3: A Supplementary Prediction Technique

This section concerns the occasion where the method to find a data base sample in Chapter 4 fails to find even a single program on which to use the technique in Section 5.2.

In such an event (or perhaps when only--say--one or two programs are in the data base sample) the method of regression can be used to come up with a prediction of the amount of execution time that should be initially forwarded for application of alpha testing.

It is beyond the scope of this project to delve into (statistical) regression techniques in depth. However, as only a crude estimate for the necessary amount of execution time to be initially used is needed here, the use of least squares can almost--but not quite--be forwarded for "mindless" application.
From the initial sample of m programs used in Chapter 4, form a scatter plot of true program line count (according to the criteria set forth in Section 5.1 of this chapter) against total alpha test execution time actually used in successful development of the programs in the data base. This scatter plot will suggest a model, if one is not already known. (For example: if we assume that as number of failures increases, then the amount of execution time needed for alpha test increasingly increases, Figure 4 in Chapter 4 suggests our model should be something like $Y = aX^{**2}$, where $Y$ is the amount of execution time needed for alpha test and $X$ is line count according to the criteria in Section 5.1.)

If the predicted execution time needed for alpha testing is accurate, then the estimated additional execution time needed for testing (see Section 5.4) will be reasonable, and probably sufficient for our purposes.

For greater accuracy, more robust techniques of regression can be looked into to see if a better initial estimate of needed execution time is obtainable for alpha testing. However, these techniques tend to be more

---

This possible model is only a suggestion. A curve of the form $aX^{**2}$ and $\exp(ax) - 1$ may look similar but behave differently. I believe that complexity and error-proneness increase roughly as $(\text{count})^{**2}$, but there is a process method-dependent factor as well.
computationally intensive and few statistical software packages currently implement them.

The method of Section 5.2 of this chapter is considered to be significantly superior to use of regression, hence this technique should be used only as a last resort.

Section 5.4: Estimation for Alpha Testing

When the length of execution time predicted as necessary to achieve failure level objective (call this length of time "S") has been completed, the failure data for the particular program in testing should be used to see if the failure objective has actually been realized, or that more execution time in alpha testing should be expended (see Figure 6, where we have arrived at the stage "Execution time to date").

The technique of choice here is maximum likelihood estimation,\(^{21}\) although other methods are available and feasible.

The basic execution time model yields the dashed line of failure intensity anticipated against execution time as exemplified in Figure 6. This line is determined from the two parameters defined in Section 5.2—\(\lambda_0\) (the initial failure intensity) and \(\nu_0\) (the total expected failures). The

---

\(^{21}\)See Hogg-Craig, fourth edition (1970) for a description of this method.
Figure 6\textsuperscript{22}

Conceptual View of Parameter Estimation

\textsuperscript{22}This figure is excerpted from Musa et al. (1987).
parameter values are selected such that the likelihood of occurrence is maximized for the failure intensity set experienced by the program under testing. In an approximate sense, the indicated data determines the curve fitting outcome.

The method of maximum likelihood is the most widely selected estimation technique of choice. It is particularly applicable when the data's underlying distribution is specified (as is done here), or is known.

For the maximum likelihood method, we must solve the simultaneous equations, where "L" is the likelihood function,\(^{23}\)

\[
\frac{\partial \ln L(\lambda_0, \nu_0; Y_o)}{\partial \lambda_0} = 0 \quad \text{(eq. 1)}
\]

and

\[
\frac{\partial \ln L(\lambda_0, \nu_0; Y_o)}{\partial \nu_0} = 0 \quad \text{(eq. 2)}
\]

\(Y_o\) constitutes the set of observations; where "D_1" failures have been observed at execution times of \(\tau_1, \tau_2, \ldots, \tau_{D_1}\).

Now, we proceed to determine \(L\).

---

From Section 3.2, we note that the basic execution time model can be written as:

\[ \mu(\tau) = \nu_0 \left[ 1 - \exp \left( -\frac{\lambda_0}{\nu_0} \tau \right) \right]. \]

Dividing both sides of the immediately above equation by \( \nu_0 \) yields the cumulative probability mass function as approximated by the continuous cumulative distribution function:

\[ \frac{\mu(\tau)}{\nu_0} = \left[ 1 - \exp \left( -\frac{\lambda_0}{\nu_0} \tau \right) \right]. \]

Now, we note:

\[ \frac{d(\mu(\tau)/\nu_0)}{d\tau} = \frac{\lambda_0}{\nu_0} \exp \left[ -\frac{\lambda_0}{\nu_0} \tau \right] \]

is the continuous probability density approximation to our basic execution time model.

From Musa et al. (1987) and Hogg-Craig (fourth edition, 1970), we therefore know:

\[ L(\lambda_0, \nu_0; Y_n) = \frac{\lambda_0}{\nu_0} \exp \left( -\frac{\lambda_0}{\nu_0} \tau_1 \right) \prod_{i=2}^{D_1} \left( \frac{\lambda_0}{\nu_0-i+1} \right) \left( \frac{\lambda_0}{\nu_0-D_1} \right) \]

\[ \exp \left( \frac{\lambda_0}{\nu_0-D_1} (\tau_1 - \tau_{i-1}) \right) \exp \left( \frac{\lambda_0}{\nu_0-D_1} (S - \tau_{D_1}) \right) \]

where "S" is execution time to date.

To solve the simultaneous set of equations--equations 1 and 2 above--the Newton-Raphson root-finding procedure, for
example, could be used.

Musa et al. (1987) discusses confidence intervals as a step that one can take here in order to get a handle on the accuracy of the estimate for \( \lambda_0 \) and \( \nu_0 \). However, as the safety net of beta testing (and continued testing after the program is released to the field) will be undertaken, and the step of prediction has already been applied, such would be overkill. Also, there may be too little data at hand at this stage to get a sufficiently narrow confidence interval for practical usage.

The purpose of this estimate is to establish whether additional alpha testing should be done. Hence a point estimate is all we require to alert us to whether there was a deficiency in the prediction of execution time desirable for the alpha phase of testing.

For purposes of making a complete analysis, failures by category can be analyzed by the preceding method in this section to see whether there is a particular category of error that would by itself merit allocation of additional execution time to alpha testing.

Also, here is where stakeholder analysis could be used to determine whether enough alpha testing has been done (or will be done at the end of the extension), by noting whether we expect a disproportionately different outcome in determining
\( \lambda \) and \( \nu \) to have occurred because of special interests that are involved and could have influenced the outcome of those parameters in positive or negative ways.

Should additional execution time be desirable, the quantity of additional execution time determined from the point estimate—with a slight adjustment possibly being made as a result of the stakeholder analysis—should be undertaken and then the program will move on to beta testing without further exploration of the "optimal" amount of execution time needed for alpha testing.\(^2\)

It may be that the estimated basic execution time model will indicate that no further testing in alpha testing phase is necessary. In that case, the amount of time allocated from the prediction phase analysis was satisfactory.

Section 5.5: Determining Calendar Time from Execution Time Objective

Musa et al. (1987) gives a lengthy and complicated procedure by which the rate at which calendar time passes in terms of execution time can be obtained from a collection of limiting resources, such as computer time available, number of

\(^2\)In practice, a decision to extend alpha test may involve stakeholder issues. For example, if alpha-test history includes a marked number of errors which would have affected an important segment of program users, additional testing (with delay in product release) may be carried out via executive order from a manager.
testers, etc.

However, here the situation is more straightforward and simple, because in alpha test there is usually only one tester.

The single tester simply needs to make an estimate of how much execution time is accumulated per unit of calendar time --for example, a week-- and use this ratio to determine how much calendar time is needed.
CHAPTER 6

ESTIMATION FOR BETA TESTING AND RELEASE TO THE FIELD

Section 6.1: Introduction

Beta testing is a transitional test following alpha test. It typically lasts for a few months. It is done by a carefully selected sample of users who will put the program through rigorous paces and will document failures so that, before release of the software to the full set of users, the failures can be remedied. Up to ten percent of the sites to which a C'I program will go after release to the field (or a minimum of one site) may be involved in beta testing.

There is little point in forecasting an optimal or near optimal period for initial beta testing because the three to four month period is typical for C'I software regardless of size. The beta test interval is kept as short as is thought reasonable—long enough for each user to fully exercise the software in its own usage environment.

The speed and other operating characteristics of the various computers used may differ from tester to tester. A standard rating system exists for comparing the power of one computer to another, and such a rating should be used to scale the execution time of the various computers to put them on an equivalent basis as far as execution time data is concerned.
As a practical point, data is to be accumulated by individual testers, of which there may be many testers per site. However, data of execution time consumed by testers who do not find any failures are to be discarded from the sample on the assumption that they were really nonrespondents.

The data accumulated by the individual testers are to be kept in terms of execution time expended by the respective testers, not calendar or clock time. Unfortunately, the only data which may be available are failure time estimates in terms of calendar time. To remedy this, the daily times at which testing starts and ends for the day should be solicited from each tester, and the time of when the failure occurs noted. Then, with average daily execution time noted for each tester, the software analyst can compute execution time data from what is essentially calendar time data, at least as a reasonable approximation.

Each tester should be encouraged to report duplicate failures especially when these failures arise from different input values or other different circumstances.

Usually, there is to be no repair of faults underlying the failures planned at any time during beta test. Given this, the model of choice is the logarithmic Poisson model. (On occasion, however, egregious failures may be corrected during beta testing; otherwise, a notable number of the testers
may discontinue using the software before end of the beta testing period.)

Section 6.2 discusses and presents a method for generating an analyzable beta test execution time stream of failures from the data accumulated by the individual testers. This section handles the matter in terms of approximations rather than equalities.

Two parameters have to be estimated for the logarithmic Poisson model: the failure decay parameter $\theta$, and $\lambda_0$ which is the initial failure intensity parameter. Estimation of these two parameters is addressed in Section 6.3 (for $\theta$) and the beginning of Section 6.4 (for $\lambda_0$).

The remainder of Section 6.4 deals with the actual estimation for beta testing and what the objective of the estimation procedure is (which is to help determine when the process of discovering failures is becoming or already is stable, suggesting that most of the new failures that are being discovered are failures that have already been observed by others).

The estimation of when sufficient beta testing has occurred is discussed at length in Section 6.5. Section 6.5 presents criteria of when to extend beta testing; or when to proceed along the path for general release of the software to the field, which is to detect and correct the underlying
faults for failures observed during beta testing.

Section 6.2: Generating an Analyzable Beta Test Execution Time Stream of Failures

The ensemble of failure observation streams by individual testers can be looked at as a queuing theory "arrival stream" of failures (by individual tester and by execution time); it may be assumed memoryless, as a reasonable approximation. According to Ross (third edition, 1985), a random variable \( X \) is called memoryless if

\[
P(X>s+t|X>t) = P(X>s) \text{ for all } s, t \geq 0.
\]

We are interested in the formation of a single stream of failures by which we can approximately determine the "measures" \( \theta \) and \( \lambda(t) \), which are the failure decay parameter and failure intensity parameter, respectively.

Adapting methodology in Gross-Harris (second edition, 1985), and assuming each tester has approximately the same propensity to discover failures, the individual streams of failures can be combined into a single failure stream as illustrated through the following example.

First, the analyst should order the individual failure streams any way he likes (particular order--systematic or nonsystematic--is not important at this step). See Figure 7 on the next page for an illustrative ordering of four failure
these marks indicate observation of a failure at indicated execution time

Beta tester #1

A1  B1  C1  D1  E1
53479 59526 43211 58116 44643

Beta tester #2

A2  B2  C2
81115 40238 69255

Beta tester #3

A3  B3  C3  D3  E3  F3  G3
98036 40577 97344 91964 83287

Beta tester #4

A4  B4  C4  D4  E4
12217 39351 70328 26240 97391

evaluation of execution time

Figure 7
Table 5

Five Digit Pseudo Random Numbers\textsuperscript{25}

<table>
<thead>
<tr>
<th>Order number of the pseudo random number</th>
<th>pseudo random number</th>
</tr>
</thead>
<tbody>
<tr>
<td># 1</td>
<td>53479</td>
</tr>
<tr>
<td># 2</td>
<td>81115</td>
</tr>
<tr>
<td># 3</td>
<td>98036</td>
</tr>
<tr>
<td># 4</td>
<td>12217</td>
</tr>
<tr>
<td># 5</td>
<td>59526</td>
</tr>
<tr>
<td># 6</td>
<td>40238</td>
</tr>
<tr>
<td># 7</td>
<td>40577</td>
</tr>
<tr>
<td># 8</td>
<td>39351</td>
</tr>
<tr>
<td># 9</td>
<td>43211</td>
</tr>
<tr>
<td>#10</td>
<td>69255</td>
</tr>
<tr>
<td>#11</td>
<td>97344</td>
</tr>
<tr>
<td>#12</td>
<td>70328</td>
</tr>
<tr>
<td>#13</td>
<td>58116</td>
</tr>
<tr>
<td>#14</td>
<td>91964</td>
</tr>
<tr>
<td>#15</td>
<td>26240</td>
</tr>
<tr>
<td>#16</td>
<td>44643</td>
</tr>
<tr>
<td>#17</td>
<td>83287</td>
</tr>
<tr>
<td>#18</td>
<td>97391</td>
</tr>
<tr>
<td>#19</td>
<td>92823</td>
</tr>
</tbody>
</table>

\textsuperscript{25}The above list is excerpted from Handbook of Mathematical, Scientific, and Engineering Formulas, Tables, Functions, Graphs, Transforms, Staff of Research and Education Association, Dr. M. Fogiel, Director. Published by: Research and Education Association, 505 Eighth Avenue, New York, N.Y. 10018, Copyright 1980, 1984 by Research and Education Association.
streams for four individual beta testers.

"End bits" are the length of execution time, by tester, in which testing took place after the tester observes the last failure he discovers. This contains information about length of time until a failure is observed that we do not want to discard.

On the assumption that time between which a given tester observes a failure tends to increase—-and we have already assumed each tester has approximately the same propensity to discover failures—we want all the failures discovered first by respective testers to come before all the failures that come second; and so on. Subject to this requirement, the randomization scheme selected here is to remove all systematic components from entering the ensemble of the failure observation streams.

To introduce the necessary randomness, we use a uniform zero-to-1 pseudo random number generator to assign pseudo random numbers to the failure intervals and end bits of the streams as illustrated in Figure 7 by way of pseudo random numbers given in Table 5.

---

26A pseudo random number generator generates a stream of pseudo random numbers that look and behave like a stream of random numbers, but really are systematically related.
Once the pseudo random numbers have been assigned, form the single stream as indicated in Figure 8 by ordering A1 through A4 in terms of increasing magnitude of the pseudo random numbers, then B1 through B4 in terms of the increasing magnitude of the pseudo random numbers, and so forth; and then connect them (the failure intervals and end bits) as illustrated in Figure 8—first the A’s, then the B’s, and so forth.

The illustrative single, failure data stream generated by this process for the ensemble collection of failure data is hence seen in Figure 8. The necessary randomization of the failure intervals and the end bits that have occurred in the individual streams is acceptably accomplished.

It should be emphasized that the conditions necessary to justify the above methodology are assumed to only approximately hold, and the "measures" $\theta$ and $\lambda(t)$ are assumed to be only approximately determined through the above single, illustrative stream and by the methods contained in Sections 6.3 and 6.4.

Section 6.3: Estimation of Failure Decay Parameter $\theta$

To solve for a confidence interval for $\theta$, we first should estimate the maximum likelihood estimate for variable $\beta$, which according to Musa et al. (1987)—for the geometric class of
these marks indicate observation of a failure

not a failure, but an end bit

execution time

Figure 8

The Illustrative Single Stream of Failure Intervals and End Bits
models of which the logarithmic Poisson execution time model is a member—is given as \( \hat{\beta}_i \) where \( \hat{\beta}_i \) is the solution to the equation

\[
\frac{1}{\hat{\beta}_i} \left( \sum_{i=1}^{m_*} \frac{1}{1 + \hat{\beta}_i t_e} - \frac{m_* t_e}{(1 + \hat{\beta}_i t_e) \ln(1 + \hat{\beta}_i t_e)} \right) = 0
\]

where \( t_e \) is execution time for end of cumulative failures from \( t = 0 \) up to the number of failures \( m_* \) for which the particular confidence interval for \( \theta \) is being determined. (We desire to construct a sequence of confidence intervals for \( \theta \), increasing \( m_* \) by ten to twenty failures at a time until the total number of failures that arose in beta testing is accounted for.)

Now, Musa et al. (1987) note the function \( I(\hat{\beta}_i) \) is determined by the geometric family as follows.

\[
I(\hat{\beta}_i) = m_* \left\{ \frac{2 t_e}{(1 + \hat{\beta}_i t_e) \ln(1 + \hat{\beta}_i t_e)} - \frac{1}{2\hat{\beta}_i^2 \ln(1 + \ln(1 + \hat{\beta}_i t_e))} \right\}^* \left\{ 1 - \frac{1}{(1 + \hat{\beta}_i t_e)^2} \right\} - \frac{t_e^2 \ln(1 + \hat{\beta}_i t_e) + 1}{[\ln(1 + \hat{\beta}_i t_e) \ln(1 + \hat{\beta}_i t_e)]^2}
\]

Now, letting \( U = \ln(1 + \hat{\beta}_1 t_e) \)

and

\[
\text{var}[U] = \frac{1}{I(\hat{\beta}_i)} \left( \frac{t_e}{1 + \hat{\beta}_1 t_e} \right)^2
\]
we next note that
\[
\frac{\hat{U} - U}{\sqrt{\text{var}[U]}} \sim N(0,1)
\]

The latter approximation can yield confidence intervals for \(U\), and thus determine \((U_{\text{low}}, U_{\text{high}})\).

That in turn, according to Musa et al. (1987) yields a confidence interval for \(\theta\) given by:
\[
\left( \frac{U_{\text{low}}}{m_a}, \frac{U_{\text{high}}}{m_a} \right)
\]

unless \(U_{\text{low}}\) is less than zero, in which case the confidence interval for \(\theta\) is given by
\[
\left( 0, \frac{U_{\text{high}}}{m_a} \right)
\]
because \(U_{\text{high}}\) must be greater than zero.

**Section 6.4: Estimation for Beta Testing**

Now that a confidence interval for failure decay parameter \(\theta = \theta\), has been estimated, only \(\lambda_o\)--the initial failure intensity parameter--needs to be estimated and then we will be able to resort to the logarithmic Poisson execution time model to determine
\[
\lambda(\tau) = \frac{\lambda_o}{\lambda_o \theta + 1}.
\]

But an estimate for this readily available. Looking at
Figure 7 for an illustrative example, use failure intervals A1 through A4 to determine the total failures interval length for the four failures--call this \( \tau_A \)--and determine

\[
\lambda_o \sim \frac{4}{\tau_A}
\]

Therefore, for the single, failure stream as illustrated by Figure 8, increase \( m_s \) by ten to twenty failures at a time to determine both the \( m_s \) and \( t_s \) to compute the confidence interval for \( \theta \).

Then compute \((\lambda(\tau)_{\text{high}}, \lambda(\tau)_{\text{low}})\) using \((\frac{U_{\text{low}}}{m_s}, \frac{U_{\text{high}}}{m_s})\)

respectively by the functional relationship

\[
\lambda(\tau) = \frac{\lambda_o}{\lambda_o \theta_{\tau} + 1}
\]

What the analyst will be looking for is a sequence that will converge to approximately a constant--a narrow confidence interval for \((\lambda(\tau)_{\text{high}}, \lambda(\tau)_{\text{low}})\) and the pattern neither falling nor rising "fast" and falling or rising decreasingly so if either of these is present--which will be indicative that the process is becoming or already is stable, suggesting that most of the new failures that are being discovered are failures that have already been observed (most of the time by others).
Section 6.5: Release to the Field

Now that we have gotten this far, there are four considerations for whether to continue beta testing:

(1) Indication that the process is becoming stable (see Section 6.4).

(2) Evidence by inspection that the beta testing is tending to rediscover mainly failures found earlier in beta testing.

(3) The number of distinct failures found should be roughly in agreement with the estimate of distinct failures for beta testing that the estimation phase of alpha testing predicted we should find in beta test.

(4) There is a low quantity of "fatal" errors, which is defined as serious errors that users cannot "work around" or correct in some unheroic and acceptable way.

If application of the four points above is unsettling, another four weeks or so of beta testing can be conducted and then all the previous steps in Sections 6.2 through the above are to be repeated. As a basic rule, no fault corrections from beta test are assumed to be made until the end of beta testing is reached, regardless of the number of times beta testing is extended. However, if problems are bad enough, this rule is broken because if beta testers cannot live with the test software they will stop using it.
is completed. Then, the C³I software is released to the field.

Before the program failure data is added to the developer's data base, the program should be used in the field for three to six months after beta testing fault corrections have been made and it is to perform in a satisfactory manner during this time. By "satisfactory manner" is meant that no fatal errors arise and only a "small"--i.e., comparable to other programs in the data base--number of failures arise. If faults for these failures are never to be corrected, it is important that these failures be considered a minor nuisance rather than a problem.

Nevertheless, most multi-user software--including C³I software--is subjected to re-release at intervals which range from a few months to several years. Residual faults which are acceptable to users (i.e., which users can "live with") are corrected in subsequent releases for which additional changes are made.
CHAPTER 7

CONCLUDING REMARKS

This report describes a set of methods and processes which can be used in measuring and managing software integrity in the typical two-phase (alpha, beta) testing environment.

Chapter 3 describes two software integrity models. The basic execution time model is shown in Chapter 5 to be most applicable to the alpha testing phase. The logarithmic Poisson execution time model is shown in Chapter 6 to be most applicable to the beta testing phase.

Prediction and estimation techniques for alpha testing are developed in Chapter 5. The prediction technique depends on a prior experience data base. A method, based on analysis of variance, for evaluating and selecting an appropriate sample from this data base is given in Chapter 4.

In Chapter 6, an analytic estimation technique is presented for use in beta testing. Using this technique allows a software developer to apply analytical processes to determine if the beta testing process has achieved a desired level of integrity in the software.

This report provides a set of tools for management of software integrity during and after the development process. We appreciate that in the real world of software development,
traditional procedures are largely heuristic. Data bases such as that described here seldom exist in usable form. However, software integrity is of steadily growing importance, not only because of increased reliance on software controlled systems, but also because of increasing demands on integrity of systems dealing with personal and financial data.

Existing books and papers dealing with software reliability or integrity have emphasized mathematical models of software integrity. This report takes a system view of the application of models and complementary analytical techniques to the development process.

In practice, there are many variations within the two-phase (alpha, beta) testing process on which our study was based. Alpha testing, for example, includes testing done at subprogram or subsystem levels as well as integration testing and full-system testing. Each of these is different in its approach and metrics. Software schedule and cost, as well as system reliability, are influenced heavily by software development at the lower levels of integration. New models and methods are probably needed to deal with this more detailed view of software development.

There are several areas where improvement in the procedures contained in this project report can be realized. First, the analysis of variance technique can be made more
powerful in the undesirable event that columns in the matrix used for the analysis of variance technique (see Table 3) have variances that significantly differ from one another. Here, a transformation of the data is usually called for, and perhaps a transformation can be found that a layman might be able to apply.

Second, the stakeholder analysis needs development, and perhaps quantitative measures can be developed to make its application (and impact) more objective and straightforward.

Third, the beta testing procedures should be made more objective with the identification of quantitative measures to use in the beta testing analysis. Furthermore, the prediction phase for beta testing could be improved with the identification of a software integrity model that can be used in the beta testing prediction phase such that the parameters of the model can be quantitatively determined by making use of prior data samples. This would help serve to properly determine what length the prediction phase for beta testing should be as well as what objectives should be expected to be achieved from the prediction phase of beta testing.
SELECTED REFERENCES


Dr. Walter R. Beam served as a Domain Expert on C'I software systems engineering requirements and C'I systems engineering principles in general.
VITA

Edward Hirschman was born on August 1, 1950, in Agana, Guam, and is an American Citizen.

He graduated from W. T. Woodson High School in Fairfax County in Northern Virginia, in 1968. From George Mason University, Fairfax, Virginia, he has received the following earned degrees: Bachelor of Science in Economics (1973), Bachelor of Science in Mathematics (1974), Master of Arts in Economics (1986), Master of Science in Operations Research and Management Science (1987), Master of Science in Systems Engineering (1988), Master of Science in Mathematics (1989) and Master of Science in Statistical Science (1991). From Virginia Polytechnic Institute and State University, he has received the following earned degree: Master of Arts in Economics (1989).

In 1974, he worked for the World Bank as a research assistant. From 1975 to 1978, he was employed by or for the Department of the Navy as an operations research analyst, specializing in life cycle costing of developmental aircraft and avionic equipment. From 1978 to the present, he has worked as a freelance professional writer and wordsmith.