1-1-1983

The measurement and estimation of the reliability of computer software.

Larry S. Musolino

THE MEASUREMENT AND ESTIMATION OF THE RELIABILITY OF COMPUTER SOFTWARE

by

Larry S. Musolino

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

1983

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

August 22, 1983
(date)

Professor in Charge

Chairman of Department

ABSTRACT

This thesis will deal with the measurement and estimation of the reliability of computer software. The software portion of a computer system is the instruction or code used to program the hardware portion. The separation between hardware and software in a computer system with respect to reliability is a distinct one. Much theory and methodology has been developed and applied in the area of hardware reliability, however the basic differences between hardware and software reliability requires the development of models specifically geared for the measurement of software reliability.

TABLE OF CONTENTS

INTRODUCTION

Until recently, advances in hardware capabilities and reliability have not been matched by corresponding advances in the software area. To make matters worse, software is now being applied to solve larger and more complicated problems. Also, the use of computers is finding more widening applications in almost every aspect of daily life. With respect to large computer systems, the cost of computer software as part of total system cost is increasing faster than the associated hardware. Currently, it is estimated that U.S. users spend over 10 billion dollars for software every year [1]. The ratio of software expenditures to hardware expenditures is currently estimated at four to one. This ratio is predicted to rise to nine to one by 1985 [1]. These exorbitant software costs can be mainly attributed to software maintenance and testing (See Figure 1).

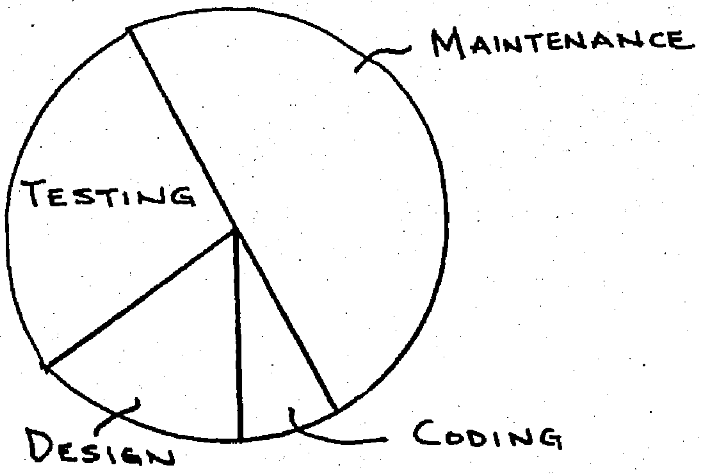

Figure 1. Typical breakdown of software costs

The costs of maintenance (fixing errors and adding changes after software is installed) and testing account for approximately 75 % of the cost of software. For example, the SAGE system, a military defense system, had an average software maintenance cost of approximately 20 million dollars per year after ten years of operation, compared to an initial development cost of 250 million dollars [2]. In typical releases of the IBM OS/360 operating system, approximately 60 % (and as high as 75 %) of the software costs were incurred after system installation. For both these examples, the costs mentioned are for maintenance only. The maintenance and testing costs probably exceeded 80 % of the total costs. It is obvious that the high cost of software is mainly due to reliability problems. In fact, in many instances the situation exists where software reliability is the limiting factor in the total reliability of a computer system. Thus, the need for a formalized method for measuring and estimating the reliability of computer software is apparent.

If the study of hardware reliability is taken as a starting point because a large amount of theory has been developed, then one would consider the possibilities of applying these results to the analysis of software reliability. Several attempts at this course of action have led to only limited success[3]. The primary reason for this limited success is the significant differences between hardware and software reliability, especially with respect to failure mechanisms.

A hardware component (e.g. integrated circuit) is assumed to have failed if a certain parameter or characteristic is found to have fallen out of a specified range or interval. This can occur either through catastrophic failure or a gradual drift out of specification. Software, however does not actually fail. Rather, it may contain one or more errors. The error has been present from the outset and when that section of the program containing the error is executed the error becomes apparent. The error(s) may or may not cause system failure. Thus, software remains the same as it was before an error was discovered, whereas hardware undergoes a change at the instant of failure.
Another inherent difference between hardware and software is with respect to testing. If software could be tested exhaustively for every possible input (an impractical task in most instances), then that particular software could be considered error free, i.e. never causing system failure. Hardware, however, could fail following extensive and even exhaustive testing.

A third difference is redundancy. Hardware reliability can be significantly improved through redundancy (i.e. parallel connection of identical components). Redundancy in software is meaningless since the same error or errors would be present in identical software.

If we formulate a reliability model where it is assumed that hardware reliability is independent of software reliability, then the total system reliability is the product of the hardware and software reliabilities. In practice, computer systems will utilize time-tested and established hardware whereas the software is specialized and still developing. Because of this, software problems may manifest themselves during all but possibly the last stages of a system's life. Software reliability measurement is crucial during the debugging and testing stages of system development. As an example, the question of the amount of money and time to be spent on debugging and testing can be answered only if reasonable measurements and accurate predictions of the reliability of the software can be made. Also, software reliability measurement is necessary to provide information and protection to current and future users of computer systems.

In an attempt to satisfy these goals, the development of techniques and testing procedures to prove that a computer program is error-free would be very desirable. However, for large programs or groups of programs this methodology may not be practical. For example, Dijkstra [4] establishes the correctness of a six-line program through a proof which is two pages long. The complete testing of a program may also not be feasible since a large program can be subjected to only a small percentage of all possible inputs. There is a very high probability that a large software system will have one or more errors during the debugging and testing stages and it is also likely that one or more bugs will remain even following several years of field operation. It is in recognition of these difficulties that computer programs which are not perfect are accepted and more attention is addressed to the problem of measuring the reliability of computer programs.

Since computer software does not age with time, it is reasonable to assume that the failure rate is constant between points in time where changes are made. Each time an error is detected, an attempt is made to eliminate the error, with the goal of reducing the overall failure rate. In practice, however, an attempt to eliminate an observed error may in fact introduce new errors causing the failure rate to actually increase. If we assume that t1, t2, ... are points in time at which errors are detected then a possible model for the failure rate of the software would be as shown in Figure 2.

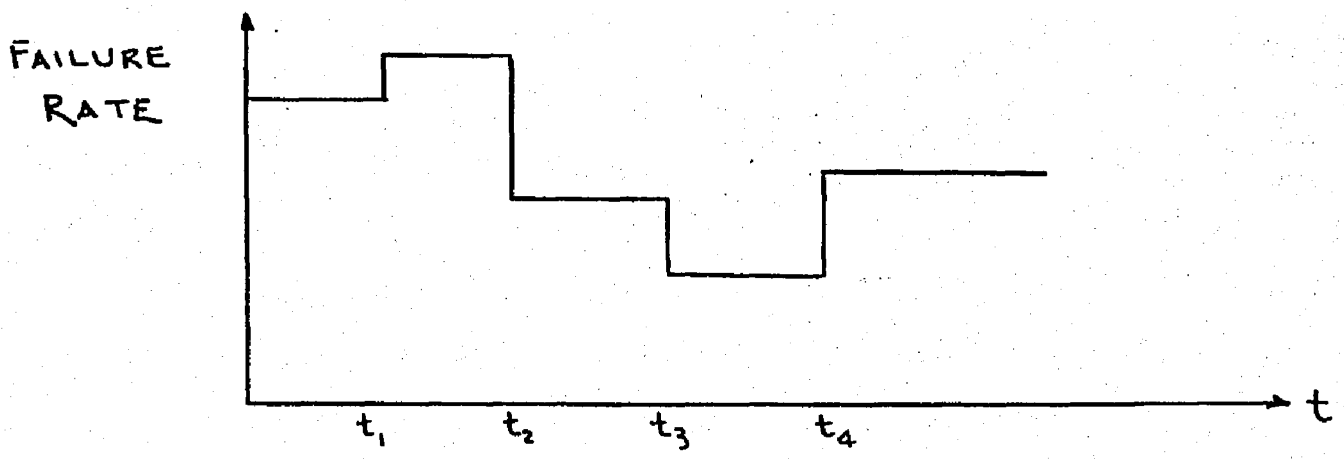

Figure 2. Failure rate as bugs are detected

Reliability models of the type above are difficult to work with and are not discussed extensively in the literature. Instead, the above model is simplified by assuming that no new errors are introduced when software is modified. This assumption causes the failure rate to resemble a step function, as shown in Figure 3.

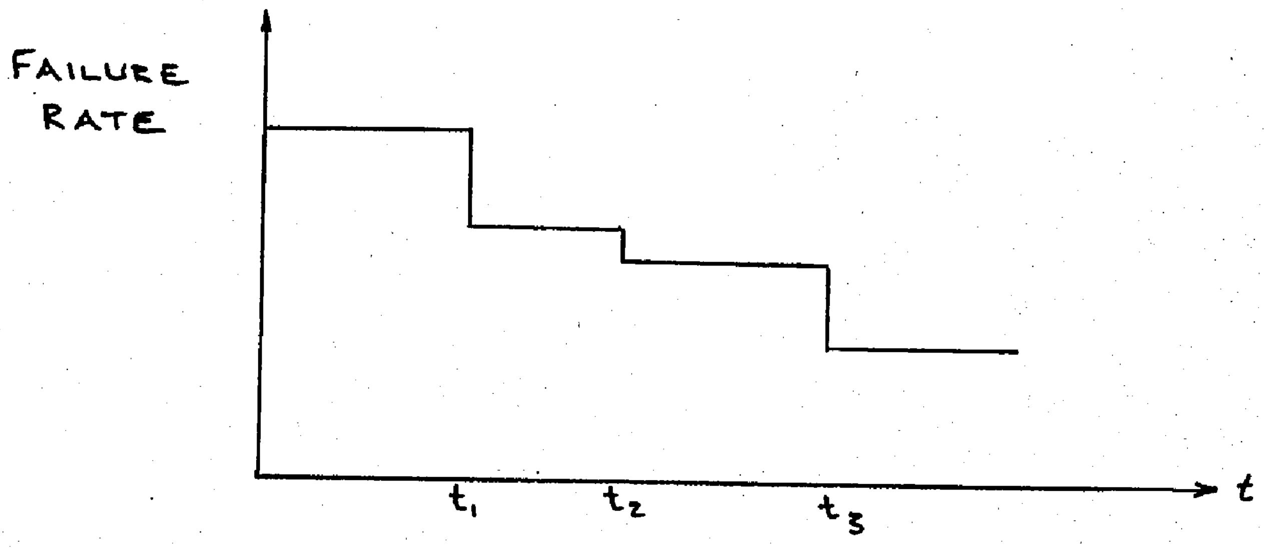

Figure 3. Failure rate as bugs are removed

This failure rate forms the basis for the reliability models proposed by Jelinski-Moranda and Shooman. These, as well as several other models are discussed in the next section.

STATISTICAL MODELS AND WORK TO DATE

Several authors have formulated models for the reliability of software and a few are described here:

A. Jelinski-Moranda Model

This model [5,6,7] has received widespread attention and use. This model also forms the basis for the methodologies proposed herein.

It assumes an exponential probability density function (pdf) for software bugs. The software failure rate for the i th bug, lambda(i), is assumed to be proportional to the number of bugs remaining in software, thus

$$\text{lambda}(i)=C[E(0)-(i-1)] \qquad (1)$$

where lambda(i) is the software failure rate for the i th bug,

C is a proportionality constant,

E(0) is the number of initial errors, i.e. the number of errors present at time t=0.

Using the exponential model and a failure rate given by equation (1), the relationship between reliability function R(t) and failure rate lambda(t), namely:

$$R(t) = \exp[-\int_0^t \text{lambda}(x)dx]$$

gives:

$$R(t) = \exp[-C(E(0)-i+1)t] \qquad (2)$$

The mean time to failure (MTTF) can then be derived as:

$$\begin{aligned} MTTF &= \int_0^\infty R(t)dt \\ &= \int_0^\infty \exp[-C(E(0)-i+1)t]dt \\ &= \{\frac{-1}{C(E(0)-i+1)} \exp[-C(E(0)-i+1] \quad \} \\ &= \frac{1}{C(E(0)-i+1)} \qquad (3) \end{aligned}$$

Thus the MTTF is seen to be inversely proportional to the number of initial errors and number of remaining errors.

B. Shooman Model

This model [8,9] assumes that the total number of machine language instructions is a constant, the number of errors at the start of integration is a constant and decreases as errors are corrected. No new errors are introduced during the process of testing. The difference between the errors initially present and cumulative errors corrected represent the residual errors. The failure rate is then assumed proportional to the number of these residual errors.

Thus: $\text{eres}(x) = e(0) - \text{ecum}(x)$ (4)

where $E(0)$ = the number of initial errors

$e(0)$ = the number of errors present at time $x=0$ normalized to the total number of machine language instructions, I.
= $E(0) / I$.

$eres(x)$ = number of residual errors present at time x, normalized by I.

$ecum(x)$ = Number of cumulative errors corrected by time x, normalized by I.

x = the debugging time since the start of system integration.

Since the failure rate is assumed proportional to the number of residual errors:

$$lambda(t) = D\ eres(x)$$

where D = constant of proportionality

Thus the reliability function can be found by:

$$R(t) = \exp[-\int_0^t lambda(x) dx]$$
$$= \exp[-\int_0^t D\ eres(x) dx] \qquad (6)$$

And, as assumed, since the hazard rate is assumed independent of time, a constant failure rate is obtained:

$$MTTF = \frac{1}{lambda(t)} = \frac{1}{D\ eres(t)} \qquad (7)$$

In order to estimate the mean time to failure, equation (4) is substituted in equation (7) producing:

$$MTTF = \frac{1}{D\ eres(t)} = \frac{1}{D[e(0) - ecum(x)]}$$

$$= \frac{1}{D[(E(0)/I) - ecum(x)]} \qquad (8)$$

There are two unknowns in Equation (8), namely D and E(0). There are various methods for estimating these parameters, two of which are the moment matching method [10] and the maximum likelihood method [8]. The moment matching method is discussed here:

Consider two debugging intervals x1 and x2 such that x1 < x2. Then:

$$\frac{T1}{n1} = \frac{1}{D\ [e(0) - ecum(x1)]} \qquad (9)$$

and:

$$\frac{T2}{n2} = \frac{1}{D\ [e(0) - ecum(x2)]} \qquad (10)$$

where T1, T2 = system operating time corresponding to x1 and x2.

n1, n2 = number of software errors during x1 and x2.

Dividing equation (9) by (10), recalling that E(0) = I e(0) and letting alpha = (T1 n2)/(T2 n1) = MTTF1/MTTF2 yields:

$$alpha = \frac{T1\ n2}{T2\ n1} = \frac{e(0) - ecum(x2)}{e(0) - ecum(x1)}$$

$$e(0) = \frac{e(0) - ecum(x2)}{alpha} + ecum(x1)$$

$$I\ e(0) = I\left\{\frac{e(0) - ecum(x2)}{alpha} + \frac{alpha\ ecum(x1)}{alpha}\right\}$$

$$E(0) = \frac{I\ e(0)}{alpha} + I\left\{\frac{alpha * ecum(x1) - ecum(x2)}{alpha}\right\}$$

$$E(0) = I\left\{\frac{alpha * ecum(x1) - ecum(x2)}{alpha - 1}\right\} \qquad (11)$$

This provides an estimate for E(0). To estimate D, we can combine this result, Equation (11) with Equation (9) to produce:

$$D = \frac{n1}{T1\ \{[E(0)/I] - ecum(x1)\}}$$

This provides an estimate for D.

Both the Shooman and Jelinski-Moranda models have the same basic structure, each representing the failure rate as a decreasing step function. In the Shooman model, however, the number of errors corrected instead of the number of errors found to have occurred is used to estimate the reliability (number of remaining errors). The distinction between the number of errors which have occurred and the number of errors removed is needed when the errors are not corrected at the time they are discovered.

C. SCHICK MODEL

The Schick Model [11] assumes a failure rate proportional to the remaining errors and debugging time.

Let t(i) be the time interval between the (i-1) and i th error.

Then

$$\text{lambda}(t) = F[E(0) - (i-1)] * x(i) \qquad (12)$$

where F is a proportionality constant

The reliability function can be obtained by:

$$R(t) = \exp\left[\int_0^t \text{lambda}(x)dx\right]$$

$$= \frac{\exp[-F * (E(0) - i + 1] * t * t}{2} \qquad (13)$$

$$\text{MTTF} = \int_0^\infty R(t)dt$$

$$= \sqrt{\frac{\text{pi}}{2 * F[E(0) - i + 1]}} \qquad (14)$$

There are advantages and disadvantages in assuming the hazard rate proportional to the debugging time. Application to existing data is probably the optimal way to decide if this method is suited for a particular application.

Several more models have been proposed in the literature by various authors [7]. In addition, Bayesian models have also been proposed [12]. The true value of any particular model lies in its ability to predict with a desired accuracy. Since data is relatively scarce, software error documentation is sparse (and there is a lack of consistency in what data is available), experimental validation of these models is limited. One such attempt has been reported [7] wherein nine models were compared. The error data used by Sukert came from the Software Problem Reports (SPRs) during the software development of a large command and control system. The software was written in Jovial J4 code and consisted of approximately 250 routines and over 100,000 lines of code. The data was restructured so that each entry corresponded to a single error and entries due to non-software errors were deleted. The data was then sorted according to the date of the SPR in order to provide a time-wise input to the models. Data on CPU time was not available and a day was considered the basic unit of debugging time.

Several models were then compared and the following conclusions were drawn (The Shooman model could not be compared due to the unavailability of CPU data) by Sukert:

1) The Jelinski-Moranda and Schick models provided higher predictions for the number of remaining errors than was actually the case.
2) The Jelinski-Moranda and Schick models appeared to provide fairly accurate results for the number of remaining errors where the testing phase was short or program length relatively short.

3) For programs where the testing phase was long or program length was large, a slightly modified version of the Jelinski-Moranda model provided the best prediction of all the models studied for the number of remaining errors in the software.

It should be noted that the study was limited in scope and further research of this kind is required before any concrete assessments can be made.

THE SOFTWARE DEVELOPMENT PROCESS

The software development process can be organized into the following six phases:

1) System Requirement Analysis
2) Software Specifications
3) Software Design
4) Software Implementation
5) Software Validation
6) Software Operation and Maintenance

The system requirements explicitly state the performance requirements, rules, and possibly criteria for evaluating the final product. The requirements are used as the standard against which the acceptance of a product is based. The system requirements are analyzed to determine if they are to be included in hardware or software subsystems. Software subsystem requirements are formulated at this point. These requirements now become the software specifications and serve as the basis for software design, implementation, validation and documentation. Research has shown [13,14] that most software errors (up to 70 %) are introduced due to incomplete specifications. Most of these errors are not detected until well into the development process. Of course, the cost to find and correct these errors increases as development time increases.

Following software specification, software design begins. This design consists mainly of the algorithms, data structures and formulations of specific functions on a particular computer system. A recently developed software design technique emphasizes top down design to minimize logical errors through rigid structuring and a sequence of steps designed to break each task into a number of smaller tasks, thus affecting "modular" design [15,16].

Software implementation consists of coding programs according to the software design. It is in this step that a structured program may improve reliability and productivity. Also, a high level language is chosen, as well as programming standards. The errors resulting from software implementation are, in general, easier and relatively inexpensive to discover and correct.

Software validation is an unnecessary step if the previous four phases have been executed error - free. However, this is very often not the case. Validation consists of demonstrating that the software meets the previously established requirements. This will consist of either code analysis or testing, or both. Code analysis or testing refers to whether or not program execution is required. Code analysis will typically consist of program and flow analysis, i.e. statement never reached, variable never initialized, etc. Code testing will consist of verifying that the program actually executes to implement the established requirements (i.e. input and output specifications). Much research has recently been devoted to "proving" a program is correct [17], however this approach has been used only in small programs.

The testing of a program consists of executing the program with certain inputs and checking the respective outputs for validity. In order for this to guarantee correctness, it implies that all "necessary" test cases are included and the number of test cases is reasonable. Testing of a program for every possible input combination is not always a feasible alternative. Finding a small subset of possible inputs for use in program testing is a current area of active research [18]. Software testing and validation is discussed in more detail in the following section.

Software operation and maintenance refers to the time period following the "installation" of software. Errors may be detected by the customer; other modifications may be due to changing or mis-interpreted requirements. Changes in the software, once installed, may introduce additional errors, and after a certain point it may be cost effective to develop an entirely new system.

CURRENT METHODS OF SOFTWARE TESTING

Software, in general, has had a radical increase in complexity in the last 10 years. Software testing has undergone a corresponding increase. Previously, software testing was an informal process where the programmer would exercise his code against a small set of arbitrary test cases. As the volume and complexity of software has increased, it has become clear that a formal and thorough procedure for software testing is increasingly important.
Software testing now has more formal procedures:

1) Software testing is now performed over as much of the development cycle as possible.
2) Testing is now more formalized with specifically identified activities.
3) Some aspects of software testing are being performed by organizations independent of the software designer or programmer.

This section will discuss the current software validation techniques which are being employed. Software validation is one approach taken towards the goal of achieving reliable software. Of course, the approach of improved design and implementation to achieve reliable software is preferred.

Software validation consists simply of ensuring that the particular software module being tested meets its specific requirements. Since a major problem in dealing with large software systems is their size and complexity, automation and the ability to automate validation techniques is discussed. In dealing with software validation, it is important to realize the types of errors encountered and how they might be introduced. A software error is some mechanism which causes the software to deviate from its intended program behavior. These errors can be broadly sub-divided into performance and logical errors. The former are errors which lead to failures where results are not produced within specified limits (e.g. time or space). The latter are errors which may be introduced through implementation. Logical errors can be further divided as:

a) Control flow errors: these errors may result from a failure to test for a certain condition, and may result in the execution of erroneous programming.
b) Path selection errors: these errors may occur due to a condition being incorrectly expressed, thus, an action is sometimes performed (or not performed) under erroneous conditions.
c) Incorrect action: these errors may result when a required computation is either not performed or performed

incorrectly.

d) Interface errors: these errors may occur if a calling and called module (e.g. subroutines) are inconsistent with each other.

A similar classification of errors is proposed in Reference 19.

PROGRAM TESTING

Program testing is the process of exercising a program with a set of inputs and checking the corresponding outputs. Software program testing is currently an area of very active research, especially with respect to the selection of input test sets. A set of test data is optimal if it detects errors in a program whenever it is incorrect. Using the notation given by Howden [20], let P be a program to implement a function F with domain D. Then T, a certain test set which is a subset of D, is a reliable test iff $\forall$ d $\in$ T, P(d) = F(d) implies $\forall$ d $\in$ D that P(d) = F(d).

An optimal test criterion is one that generates reliable tests. Goodenough [21] has proposed that a test criterion C is reliable if it can be shown that every set of test input T is executed successfully by the program or every set is executed unsuccessfully. C is a confirmed criterion if and omly if it can be shown that for every error contained in the program, there is a set of test data that satisfies the criterion and is capable of showing the error. An optimal test criterion thus is one which is both reliable and confirmed. Exhaustive testing is both a reliable and confirmed test criterion. In practice, showing a particular criterion, which is not exhaustive, to be both reliable and confirmed may be difficult. However, the theory explains why testing all program statements, branches, loops, etc. may not be optimal test criteria.

Howden [20] showed there to be no general procedure for obtaining a reliable test from a program. Thus, the best possible may be test strategies which will work only for a particular class of programs. Testing still remains a very important tool in software validation, despite these discouraging theoretical results. In many applications there may be no substitute for testing, especially for very large programs. In addition, where software is used in critical applications the testing may have to be performed in the operating environment.

EXHAUSTIVE TESTING

Exhaustive testing requires that all possible inputs belonging to the input domain be used. Such a set of tests will also be reliable. Exhaustive testing, in theory, can guarantee software validity. Obviously both excess validation time and excess expense can become a problem. Thus, due to the large number of possible inputs, exhaustive testing may not always be feasible. Also, it may be impossible to test for certain program behavior. In fact, some programs may have infinite input domain so that exhaustive testing is a certain impossibility.

An alternative, proposed by Boehm [1] suggests that every executable path in a program be exercised at least once. This test criterion however, may not expose errors due to control flow or path selection. Also, the number of test cases required tends to be large and testing all executable paths at least once may be infeasible. Another alternative is to test a program with a large number of inputs which are randomly distributed over the input space. The intention is that the results of a random sample will give a reliable indication of program reliability. Statistical methods can then be used to derive an estimate of reliability with corresponding confidence intervals.

FUNCTIONAL TESTING

Functional testing, the most widely used testing method consists of selecting an appropriate set of test inputs, executing the program and examining the outputs. This selection of test inputs is based on a review of the software requirements, design, etc.
The tests are selected to show that the software contains certain desired capabilities and characteristics. In actual practice, the selection of these inputs is usually made by an experienced programmer, who has some idea of the sources of common errors. In Reference 21, Goodenough suggests a procedure to select input test data via a decision table. All possible combinations of conditions that can occur are tabulated. As software development proceeds, the table is expanded Subsequently, test cases are selected such that all entries in the table are tested.
Hetzel [22] has shown functional testing to be more attractive than criterion dependent testing, to be discussed below. The attractiveness of a testing procedure depends on the selection of test

cases. A certain programmer may choose test inputs based on his prior experience or the intended operation of the software. Thus his test inputs may test only those parts of the program with which he is familiar, possibly leaving some errors undetected. It is seen that a methodological approach to the selection of input test data is required.

CRITERION DEPENDENT TESTING

In criterion dependent testing, test inputs are generated until a given test criterion is satisfied. The test criterion is usually based on program structure. One criterion often employed is to choose a set of inputs such that every statement in the program is executed at least once. This criterion may not exercise all branches and not detect errors in program flow control.

Possibly an improved criterion is to select the input set such that all branches are executed at least once [18]. This will guarantee also that all statements are executed at least once. Howden also proposes a boundary test criterion based on the observation that a number of errors result from the handling of boundary conditions in loops. These criteria are based on program structure, and procedures for finding paths satisfying these criteria can be automated.

A basic hypothesis for criterion dependent testing is that the program input domain can be partitioned into a number of equivalence classes with the property that a test of a representative in an equivalence class will test the entire class. Thus, testing representatives from each equivalence class will be sufficient to test the program. All the above mentioned criteria fail to have this property. In fact these criteria fail to have the reliable and confirmed requirements discussed earlier. The effectiveness of these criteria is still under debate and further research is required.

Another approach to define test criterion is based on the type of errors which can be detected by the test data [23]. The type of error is defined by modifications to a program and these modifications are usually small changes at a single point in the program. The modified programs are termed mutants. A set of test cases is said to be adequate if it identifies all mutants from the from the correct program. This concept is similar to error seeding. In error seeding, the seeded errors are planted into the program manually while the mutants are generated methodically. Both techniques attempt to find a set of test cases that identifies these artificial errors.

AUTOMATED SOFTWARE EVALUATION

Ramamoorthy [24] broadly defines the characteristics of any computing system into two categories, namely behavioral and structural characteristics. A program is typically specified by its input-output relationship, which is a behavioral property. The structure of the program however is usually a function of the software designer(s). Ramamoorhty suggests that a program can be considered as the sum of its behavioral characteristics of the components on its structural form. Thus, since the complete validation of a program may not be feasible, the strategy will be to attempt a partial validation of the program components using techniques that have the potential of automation.

Many of the partial validation techniques presently used attempt to decompose the above mentioned characteristics into classes and then validate each separate class to a specified extent. However, decomposition of behavioral characteristics is a difficult task. Fortunately, careful analysis of the structural characteristics may reveal useful information which could assist in the decomposition and validation strategies. This analysis of structural characteristics lays the groundwork for most automated software evaluation procedures.

Three underlying techniques form the basis for current software evaluation:

1) Static analysis
2) Dynamic analysis
3) Simulation

Static analysis is based on the examination of the design and program code. Dynamic analysis is based on the examination of program behavior during execution.

Static Analysis

Static analysis includes a set of program analysis procedures directed towards the indictment of certain software attributes. The presence or absence of these attributes may imply a negative or positive quality concerning the software or possible sources of error(s). Static analysis primarily consists of the attempt to detect semantic and structural errors; the removal of obvious errors from the program in order to set up a configuration of the program for further analysis; and to identify questionable areas of the software which can be candidates for dynamic analysis. These features generally require a large amount of repetitive scanning of source information to be performed. In actual practice, in order to achieve efficiency, most software evaluation systems represent the programs in an internal database. Thus, static analysis consists of two major components: (1) an input analyzer to produce the database, and (2) routines for performing the structural analysis.

Analysis procedures are typically based on program and data flow. For program flow analysis, graph theory is usually employed where the program is represented by a graph and the nodes correspond to the statements and the edges correspond to the flow of control. Thus, unreachable code and looping errors may be identified by analyzing the graph [25]. Data flow analysis is achieved by program optimization techniques [26] which attempts to discover mainly errors in variable references; for example, that all variables are properly initialized.

Static analysis systems oftentimes utilize compilers to analyze the source code for syntax errors. Also, compilers are used to generate efficient code. In order to optimize the compilation, compilers are designed to save as little information as possible.

Thus, many program details such as interface parameters are not recorded at all. Even if this information is saved during compilation, it is discarded once the compilation is completed. In order to generate the data base for diagnostic purposes, the penalty of compiling the entire program is incurred, even though only minor changes may have been made to several routines.

Static analysis systems, on the other hand, are typically designed to document as much programming data as possible. The data is usually stored in secondary storage. As the source code is modified, only those parts of the data corresponding to the updates are changed. This data also represents the current status of the program and can be used for maintenance as well as documentation purposes. Custom- made analysis routines can be developed and can complement facilities already provided by the compiler.

The philosophy then, is to search for attributes in the software which may represent common programming errors or poor programming practices. Thus, static analysis systems are limited by their nature. Other errors, possibly even trivial ones will remain undetected. Also static analysis is unable to adjust its focus based on results found earlier. In many cases this analysis will indicate only the existence of possible deviations because the feasibility of the path along which the deviation was detected cannot be easily determined. These deviations would then require further analysis by other means. In addition, most of this analysis is designed to detect program structure and syntax errors while logical errors will not be detected.

Dynamic Analysis

Dynamic analysis involves the execution of programs and corresponding observation of run-time behavior. This is intended to examine certain behavioral characteristics which are not examined in static analysis. This analysis will involve both error diagnosis and the verification that performance requirements are being met. It helps to detect and locate errors by noting the various steps that occurred during execution. Paths which are traversed are recorded by the dynamic analyzer. The amount of code not exercised by the test case is usually a good indication of test ineffectiveness. Sections of code that are most frequently executed are identified for optimization purposes. These objectives are often achieved by inserting tracking code (mostly counters) in the source code in order to observe run - time behavior. This tracking code, especially for a large program, may affect the storage requirements and execution time of a program. Thus, any interference because of this tracking code must be predictable and must not affect program output.

Simulation

This is a procedure wherein system hardware/software is modeled to study its characteristics. Simulation procedures can and should be used throughout the development of software to assure that requirements are constantly being met. During system design and analysis, simulation allows the designer to assess if system objectives are being met by the set of derived requirements, test various proposed algorithms, and identify errors early in the design stages. The structure of the simulation will obviously be dependent on application and operating environment. References 27 and 28 describe systems that provide tools to assist simulations and automated mechanisms for inserting models into simulation runs.

MEASURING AND ESTIMATING THE RELIABILITY OF COMPUTER SOFTWARE

It is very likely that a large software system will have one or more bugs during debugging and testing and it is also very likely that one or more bugs will remain even after several years of operational use. This is due, in part, to the fact that every executable branch and/or statement will probably not be exercized during software testing. As the size of the software increases, the amount of coverage afforded by testing will decrease. This section will attempt to provide an estimate for the time required to debug software to a given level of reliability.

As background, the model proposed by Jelinski and Moranda [5], and discussed in Section 2 is summarized below. It is assumed that the software contains an initial number of errors, i.e. errors occurring at time $t=0$, $E(0)$, and that the failure rate is proportional to the number of remaining errors in the program. The failure rate, as given in Eq. (1), shows a linear dependence on $E(0)$ and on the constant of proportionality, C. The model assumes that an error when encountered is removed. It is further assumed that during this error removal process, no new errors are introduced.

Equation (2), Section 2 provided the reliability function, namely:

$$R(t) = \exp[\,-C(E(0)-i+1)t]$$

From this, the failure distribution can be obtained as :

$$F(t) = 1 - \exp\,[-C(E(0)-i+1)t]$$

Applying the basis of the model, i.e. every time a software error is encountered, it is removed with probability one, and since the failure rate is assumed proportional to the number of remaining errors, it can be seen that the time between failures will tend to increase. The time interval between the (i-1)st and the ith failure, t_i, will have a distribution of the form 1 - exp { -C[E(0)-i+1]t_i}.
In order to effectively make use of this model, however, both C, the constant of proportionality and E(0), the number of initial errors must be known. In actual circumstances this is not the case, and thus it is necessary to derive estimates for both C and E(0) using the time between failures. This section will deal with statistical aspects of the reliability model proposed by Jelinski and Moranda. This model can be used to address the amount of time needed to resolve a given software to a certain level of reliability, i.e. amount of debug time necessary. Obviously, the values of E(0) and C are paramount to answer this question. We assume first that these two parameters are known. Reliability, by definition, is measured on a probability scale, and

can be treated as such. Thus, if t_obj is the objective time for the software and $t1$ the time to the first detected error, the probability that the software will perform to the objective can be expressed as:

$$P[\ t1 > t\_obj] = P\_obj \qquad (14)$$

To reach this objective level of reliability, the number of errors to be removed from the software is needed. If we call this number of errors r, then:

$$\exp\ [\ -C(E(0) - r)t\_obj = P\_obj \qquad (15)$$

or:

$$r = E(0) + (\ln P\_obj)/(C*t\_obj) \qquad (16)$$

where r is taken as the closest integer value satisfying the above equation. Thus, r is a function of the parameters E(0) and C, and also of the objective reliability P_obj and objective program operating time t_obj. Also, there are certain factors which affect the model but are difficult to include mathematically. For example, there are finite amounts of time required to run the software, discover the errors, and locate and correct these errors. The size of the software will have an impact as will the method of debugging. Other factors which will affect the time to detect and correct errors include the programming language, the type of inputs used, programming structure, expertise of the debugger and so on. These parameters do not lend themselves to the inclusion in this model and are thus omitted. The model does take into account the time needed to detect these software errors.

The probability of detecting an error in software prior to installation will be proportional to the probability of exercizing the particular segment of code containing the bug. If exhaustive testing were employed, all errors would be detected in the validation phase, however as previously mentioned, exhaustive testing may be an impractical or even impossible task. This probability of detection will be based on the criticality (i.e. frequency of execution) and failure rate of individual program execution paths. With this scenario, the "most obvious" or most probable errors will be detected first, i.e. relatively small time to failure, while "embedded" errors, e.g. seldom called routines or utility modules will be the most difficult or least probable to detect.

Equation (16) provides a method to determine the number of errors that are to be removed from the software. Let the time between the detection of the i th and (i+1) st error be expressed as X(i+1). Let this also include the time required to remove the error. Then the total time required to remove all r errors in the software , X_tot , can be expressed as:

$$X\_tot = X1 + X2 + X3 + \dots + Xr$$

If we assume that these times are independent of each other, then the distribution of the Xi's can be obtained as the convolution of the distribution of the r individual times. This density function can then be given as:

$$f(Xi) = C[E(0)-i+1] * \exp[-C(E(0)-i+1)t]$$

The moment generating function of this density function can be obtained by:

$$M\_Xi(y) = \text{Expected } [\exp (y*Xi)]$$

$$= \frac{C(E(0)-i+1)}{C(E(0)-i+1) - y}$$

Thus the moment generating function of the total time, X_tot can be formed as the product of the individual generating functions:

$$M\_X\_tot(y) = \prod_{i=1}^{r} \frac{C(E(0)-i+1)}{C(E(0)-i+1) - y}$$

In order to obtain the inverse of M_X_tot(y), partial fraction expansion is necessary. This proves to be a long and tiring procedure and only the result is given here:

$$F(X\_tot) = \prod_{i=1}^{r} C(E(0)-i+1) \sum_{i=1}^{r} \left[\prod_{j=1}^{r} \frac{1 - \exp[-C(E(0)-i+1)\,X\_tot]}{[C(E(0)-j+1)-C(E(0)-i+1)](C(E(0)-i+1)} \right]$$

where the first factor in the summation is not evaluated at i=j. Needless to say, this is a lengthy expression for the distribution of X_tot, and in many cases it may be more pragmatic to deal with an estimate of the bounds of the distribution. To discuss the bounds on the distribution, it is necessary to utilize theorems developed by Barlow [29]] concerning increasing failure rates. The distributions for the Xi's, i=1,2,...,r are all increasing failure rates. This is due to the fact that the model assumes the failure rate to be proportional to the number of remaining errors. The distribution of the total failure times, X_tot is the convolution of the individual Xi's and thus this distribution itself is an increasing failure rate, which has a mean, m_X_tot, given by:

$$m\_X\_tot = (1/C)*[(1/(E(0)-r+1) + (1/(E(0)-r+2) +...+ (1/E(0))]$$

Again, using a theorem from Reference 29, a largest bound on the distribution, F(X_tot), can be obtained as:

$$F(X\_tot) < 1 - \exp[-X\_tot/m\_X\_tot]$$

For a least bound, Theorem 2.4.5 of the same reference yields:

$$F(X\_tot) > 1 - \exp[-a*X\_tot]$$

where a is the solution of:

$$\exp[-a*X\_tot] + a*m\_X\_tot = 1$$

As a specific application of these results, consider the case when r = E(0), that is, the software is debugged until all errors are removed. Also, let C, the constant of proportionality be unity. Thus, the distribution of the time to debug becomes:

$$F(X\_tot) = \prod_{i=1}^{E(0)} (E(0)-i+1) \sum_{i=1}^{E(0)} \left[\prod_{j=1}^{E(0)} \frac{1 - EXP[(E(0)-i+1)\ X\_tot]}{[(E(0)-j+1)-(E(0)-i+1)]\,(E(0)-i+1)} \right]$$

and, correspondingly, the upper bound on the distribution can be written as:

$$F(X\_tot) < 1 - \exp[-X\_tot/m\_X\_tot]$$

or

$$F(X\_tot) < 1 - \exp[-X\_tot/(\quad (1/i)]$$

The lower bound on the distribution can be written as:

$$F(X\_tot) > 1 - \exp[-a*X\_tot]$$

where a is the solution of:

$$\exp[-a*X\_tot] + a*[\quad (1/i)] = 1$$

These equations are plotted for the specific cases of E(0) = 2, 5, and 10 in Figures 4, 5 and 6 respectively.

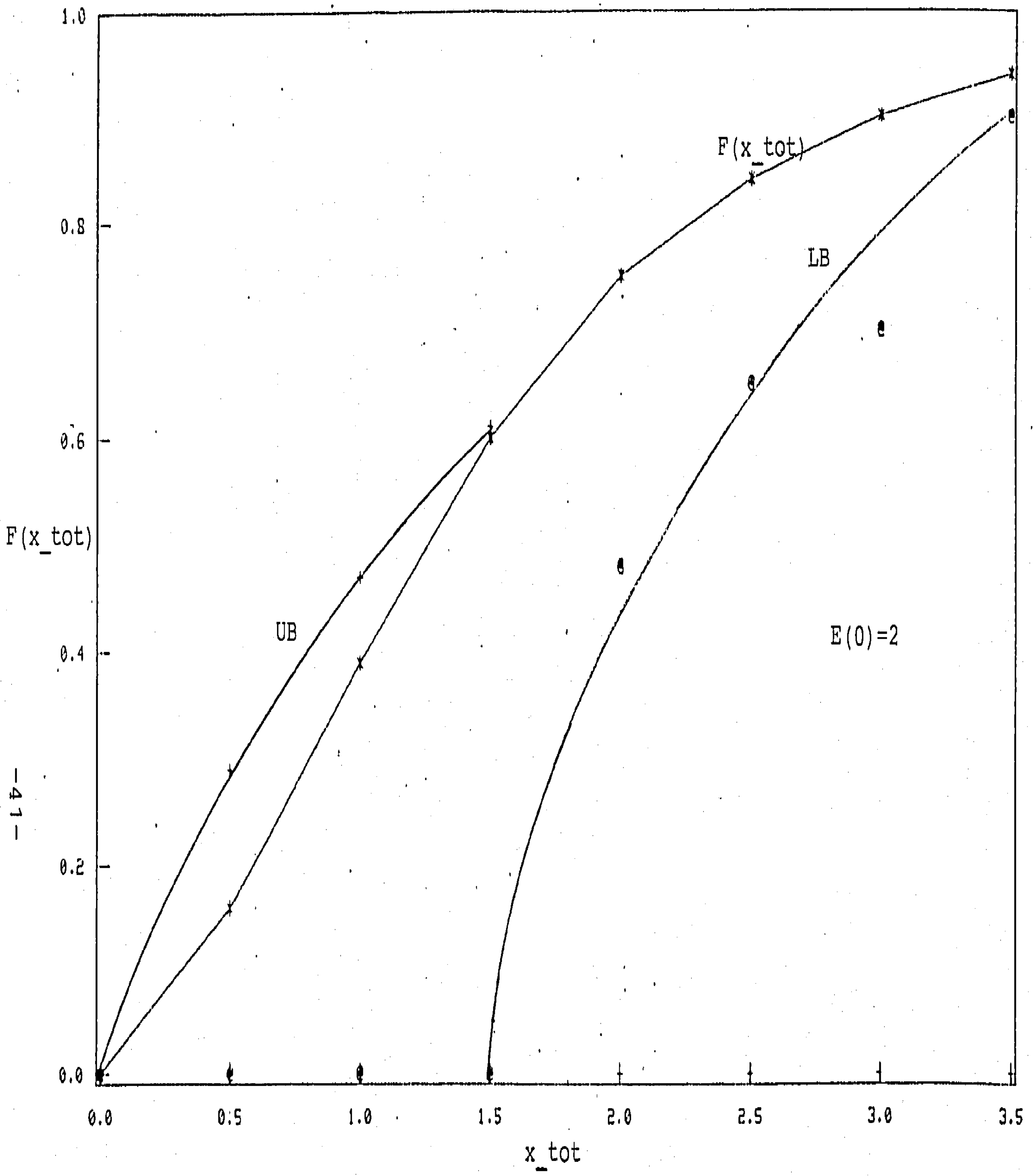

Fig. 4 Distribution of time to debug with bounds for number of of initial errors = 2.

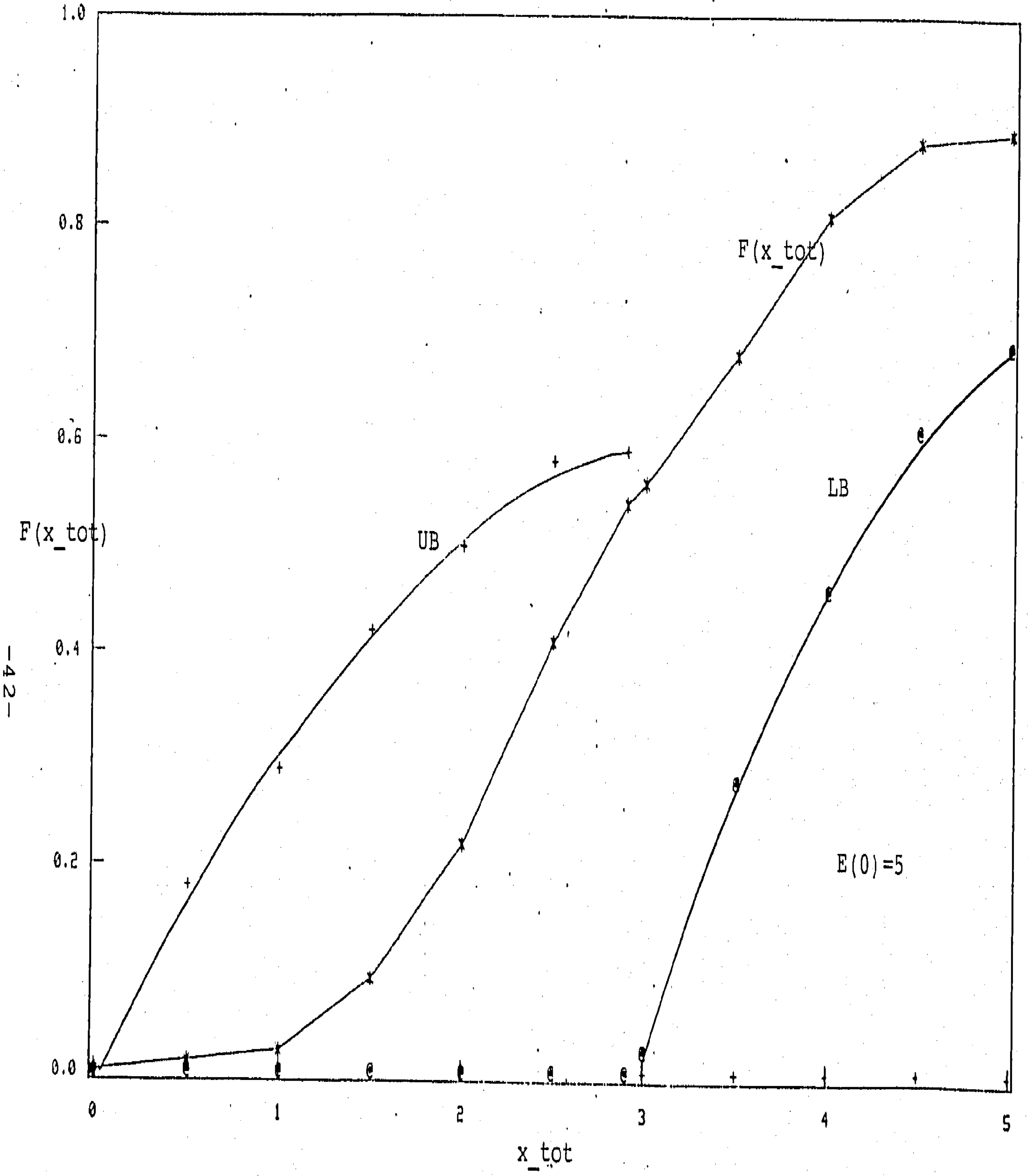

Fig. 5 Distribution of time to debug with bounds for number of initial errors = 5.

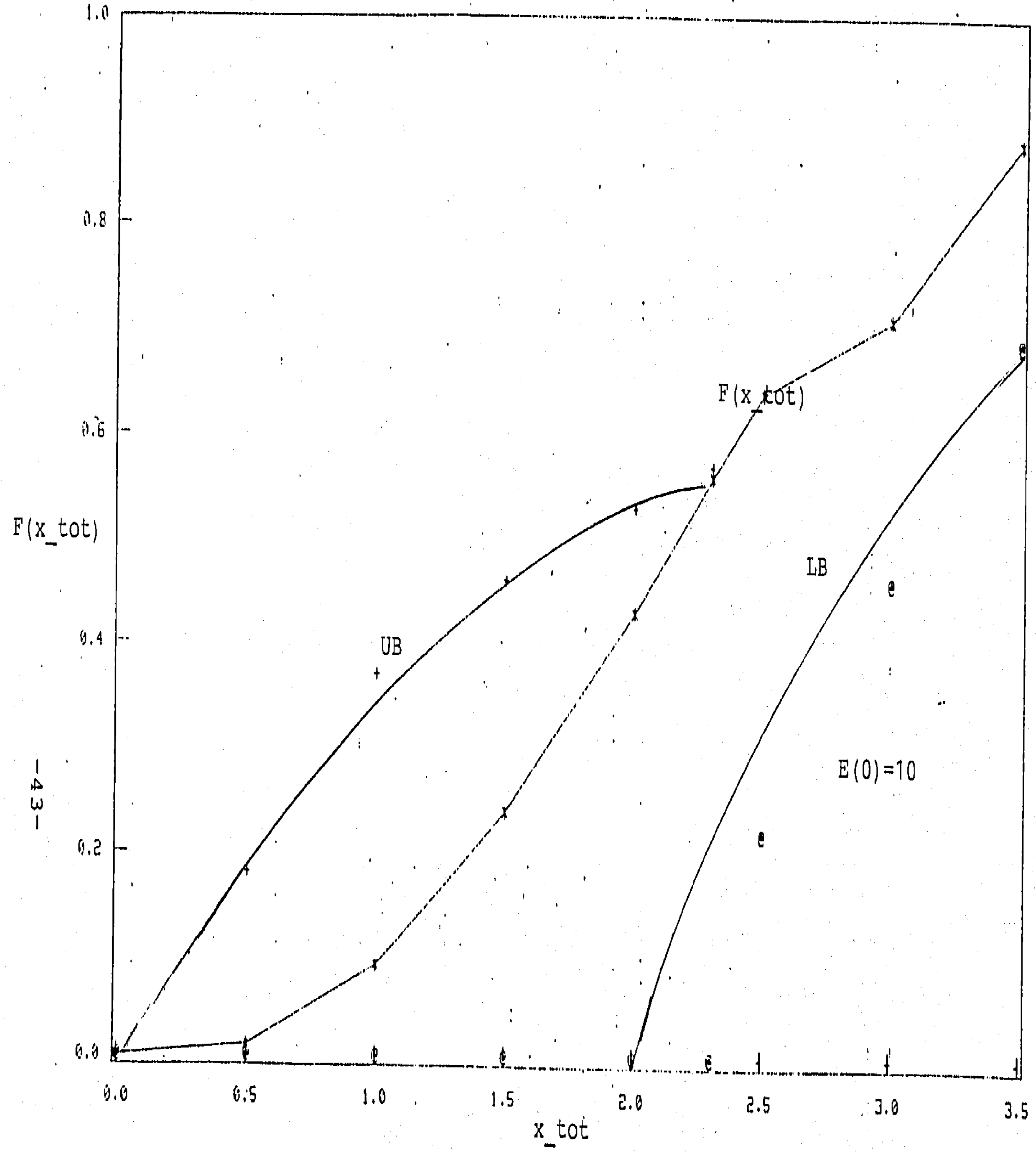

Fig. 6 Distribution of time to debug with bounds for number of initial errors = 10.

For the specific case of E(0) = 2, the distribution becomes:

$$F(X\_tot) = \prod_{i=1}^{3} (3-i) \sum_{i=1}^{2} \left[\prod_{j=1}^{2} \frac{1 - \mathrm{EXP}[(3-i) X\_tot]}{[(3-j)-(3-i)](3-i)} \right]$$

$$F(X\_tot) = 2*\{[1-\exp(-X\_tot)] - 0.5*[1-\exp(-2*X\_tot)]\}$$

and the upper and lower bounds are, respectively:

$$F(X\_tot) < 1 - \exp[-X\_tot/1.5]$$

$$F(X\_tot) > 1 - \exp[-a*X\_tot]$$

where a is the solution of:

$$\exp[-a*X\_tot] + 1.5*a = 1$$

This distribution for E(0) = 2 is tabulated below. In addition, the distribution with corresponding upper and lower bounds is plotted in Figure 4a.

X_tot	F(X_tot)	X_tot	F(X_tot)
0.0	0.000	2.5	0.843
0.5	0.155	3.0	0.903
1.0	0.399	3.5	0.940
1.5	0.604	4.0	0.960
2.0	0.748	4.5	0.987

If some arbitrary decision rule is then chosen, e.g. probability of 0.95 or greater, then an estimate of the time to debug software can be formulated. Thus, the value of the distribution will give the probability that the time to totally debug the software will take on a value equal to or less than X_tot.

Limitations of the model

Of course, the value of or an estimate of the number of initial errors present in the software is required for this model to be useful. Jelinski [6] has obtained estimates of E(0) by utilizing the times between software errors, however these estimates require caution in their use [6]. Another method to estimate the number of initial errors in a program is proposed by Halstead [30]. This interesting approach is to assume a programmer can handle on the average five "concepts" of information simultaneously. Based on this, Halstead derived n, the mean number of mental discriminations between potential errors in programming. The total number of mental discriminations required to develop a program, N, is estimated based on the total number of statements, operands, etc. in the software. The value of N, as developed by Halstead, takes into account the program volume and the level of the programming (i.e. program difficulty). Given n and N for a

program E(0), the estimated number of initial errors is given by:

$$E(0) = n/N$$

Surprisingly, this prediction agrees well with observed experimental data [30]. Measures of this type, however, are not suitable for accurate predictions of software reliability. They are based on observations with software having similar characteristics and the accuracy of such a measure on a particular program cannot be easily determined. Thus, these measures should only be used as a rough estimate of reliability.

A further limitation of the model occurs since the derivation assumes that an error, when encountered is removed. It is further assumed tha during this error removal process, no new errors are introduced.

SUMMARY

Numerous models for estimating the reliability of computer software have been proposed by various authors. Several have been discussed and one in particular, the widely-accepted Jelinski-Moranda is expanded on. This model provides an estimate of the distribution of time to debug software given that the reliability objectives are stated.

Also presented was a discussion of the software development process as well as a discussion of methods which are currently employed in the validation of software.

References and Bibliography

1. Boehm, B.W. "Software and its Impact: A Quantitative Assessment", Datamation, May, 1973
2. Thayer, R.H. "Rome Air Development Center R&D Program in Computer Language Controls and Software Engineering Techniques", RADC TR 74-80, Grifiss Air Force Base, Rome, N.Y., 1974
3. Hecht, H. "Can Software Benefit from Hardware Experience" Proceedings of the 1975 Annual Reliability and Maintainability Symposium, IEEE, N.Y., 1975
4. Dijkstra, E.W. "Notes on Structured Programming" University of Maryland Computer Sciences Center, EWD 249, 1971
5. Moranda, P. and J. Jelinski "Software Reliability Research in Statistical Computer Performance Evaluation", Edited by Walter Freiberger, Academic Press, N.Y., 1972
6. Moranda, P. and J. Jelinski "Final Report on Software Reliability Study" McDonnell Douglas Aeronautic Corp. MDC Report Number 63921, December, 1972
7. Sukert, A.N. "An investigation of Software Reliability Models" Proceedings of 1977 Annual Reliability and Maintainability Symposium, IEEE, N.Y., 1977
8. Shooman, M.L. "Operational Testing and Software

Reliability Estimation During Program Development" 1973 IEEE Symposium on Computer Software Reliability, IEEE, 1973

9. Shooman, M. L. and A. K. Trivedi "A Many State Markov Model for Computer System Performance Parameters", IEEE Transactions on Reliability, 1976

10. Singh, C. and R. Billinton "System Reliability Modeling and Evaluation" Hutchinson, London, 1977

11. Schick, G.J. "An Analysis of Competing Software Reliability Models" IEEE Transactions on Software Engineering, March, 1978

12. Littlewood, B. and J. W. Verrall "A Bayesian Reliability Growth Model for Computer Software" 1973 IEEE Symposium on Computer Software Reliability, IEEE, 1973

13. Bell, T.E. and T.A. Thayer "Software Requirements Are They Really a Problem?" Proceedings of the 2nd International Conference on Software Engineering October, 1976

14. Reifer, D.J. "Automated Aids for Reliable Software" Proceedings of the International Conference on Reliable Software, April, 1975

15. Parnas, D.L. "A Technique for Software Module Specification with Examples" Communications of the ACM, May, 1972

16. Wirth, N."Program Development by Stepwise Refinement" Communications of the ACM, April, 1977

17. Gerhart, S.L. "Observations of Fallibility in Applications of Modern Programming Methodologies" IEEE Transactions on Software Engineering, September, 1976

18. Huang, J.C. "Error Detection through Program Testing" in Current Trends in Programming Methodologies, Prentice-Hall, Englewood Cliffs, N.J., 1977

19. Howden, W.E. "Reliability of Path Analysis Testing Strategies" IEEE Transactions on Software Engineering September, 1976

20. Howden, W.E. "Methodology for the Generation of Program Test Data" IEEE Transactions on Computers May, 1975

21. Goodenough, J.B. and S.L. Gerhart "Towards a Theory of Test Data Selection" IEEE Transactions on Software Engineering, June, 1975

22. Hetzel, W. "An Experimental Analysis of Program Verification Methods" Dissertation, University of North Carolina at Chapel Hill, 1976

23. DeMillio, R.A. "Hints on Test Data Selection; Help for the Practicing Programmer" Computer, April, 1978

24. Ramamoorthy, C.V. "Design and Construction of an Automated Software Evaluation System" IEEE Transactions on Computer Software Reliability, 1973

25. Hect, M.S. "Flow Analysis of Computer Programming" North Holland Press, New York, 1977

26. Allen, F.E. and J. Cocke "A Programming Data Flow

Analysis Procedure" Communications of the ACM, March, 1976

27. Rose, C.W. "LOGOS and the Software Engineer" AFIPS, FJCC, 1977

28. Bunger, R.T. "AUTASIM: A System for Computer Assembly of Simulation Models" SIGPLAN Notices, January, 1974

29. Barlow, R.E. "Mathematical Theory of Reliability", John Wiley and Sons, New York, 1965

30. Halstead, M.H. "Elements of Software Science", Elsevier Press, New York, 1977.

VITA

Larry S. Musolino was born and raised in New York City, New York where he attended the Bronx High School of Science. He was graduated from the City College of New York in 1980 with a Bachelor of Science degree in Electrical Engineering, awarded cum laude. He is a member of the Tau Beta Pi and Eta Kappa nu national honor societies.