The Measurement and Management of Software Reliability

JOHN D. MUSA, SENIOR MEMBER, IEEE

Abstract—**The theme of this paper is the field of software reliability measurement and its applications. Needs for and potential uses of software reliability measurement are discussed. Software reliability and hardware reliability are compared, and some basic software reliability concepts are outlined. A brief summary of the major steps in the history and evolution of the field is presented. Two of the leading software reliability models are described in some detail. The topics of combinations of software (and hardware) components and availability are discussed briefly. The paper concludes with an analysis of the current state of the art and a description of further research needs.**

"Mistakes are at the very base of human thought, embedded there, feeding the structure like root nodules. If we were not provided with the knack of being wrong, we could never get anything useful done . . . we could only stay the way we are today, stuck fast. . . . Your average good computer can make calculations in an instant which would take a lifetime of slide rules for any of us. Think of what we could gain from the near infinity of precise, machine-made miscomputations which is now so easily within our grasp [1]."

ALTHOUGH the foregoing quotation may enhance the cause of creativity, it is not likely to be looked on with favor by most managers and users of computing systems. Reliability, in fact, is one of the most critical issues that software engineers are currently struggling with.

I. Why Measure Software Reliability?

Many people think of reliability as a devoutly wished for but seldom present attribute of a program. This may be fine in the abstract, but in the real world software reliability is usually achieved at the expense of some other characteristic of the *product* (program size, run time or response time, maintainability, etc.) or the *process* of producing the product (cost, resource requirements, schedule, etc.). One wishes to make tradeoffs among these software product and process characteristics. Boehm *et al.* [2] have classified some of the general software product characteristics that one may wish to consider. *Measurement* becomes very important as soon as one wishes to make a tradeoff of this sort. In some cases, reliability is one aspect of a more general characteristic such as user satisfaction or user work efficiency (e.g., in time-sharing systems) [3].

Two trends have strongly stimulated the demand for software reliability measures. The microprocessor revolution has extended the heavy dependence of institutions on computing systems down to lower and lower levels and smaller and smaller organizations. A growing proportion of the systems operate in real time and the operational and cost impacts of malfunctions are enormous. For example, consider the effects of a breakdown of an airline reservation system. The second trend, the increase in distributed processing and networking, has greatly increased the size and complexity of computing systems; many of these systems are in effect multi-multiprocessor systems with many diverse pieces of software running simultaneously and interacting. Development costs are consequently very large. The pressures for achieving a better and more finely tuned balance among product and process characteristics revolving around reliability have therefore been mounting.

Manuscript received February 1, 1980; revised April 25, 1980.
The author is with Bell Laboratories, Whippany, NJ 07981.

The discussion above of the need for software reliability metrics concentrated on tradeoffs among software product and process characteristics. Since these tradeoffs are primarily made by systems engineers, measurement is of particular importance to them. However, software reliability metrics can also be of great value to the software engineer and manager. There are three aspects of this.

1) Software reliability figures can be used to evaluate software engineering technology. The field of software engineering is in rapid ferment, but unfortunately there has been little quantitative evaluation of the continual flow of proposed new techniques. Many of these innovations have been greeted with initial enthusiasm because of the great need for them, but this attitude has often soured and turned to skepticism when many of the ideas did not turn out to be sufficiently effective to justify their cost. The difficulty of distinguishing between good and bad new technology has led to a general resistance to change on the part of software managers and software engineers that is counterproductive. A software reliability measure offers the promise of establishing at least one criterion for evaluating the new technology. For example, one might run experiments to determine the increase in mean time to failure (MTTF) at the start of system test that results from the use of design reviews.

2) A software reliability metric offers the possibility of evaluating status during the test phases of a project. A good means of evaluating testing progress has previously been rather elusive. Managers have used methods such as the intuition of the designers or the test team, the percent of tests completed, and the successful execution of critical functional tests. None of these have been really satisfactory and some have been quite unsatisfactory. On the other hand, a reliability metric can be established from actual test data which provides a much more objective means of determining status, assuming proper interpretation. It has been generally found that the increase achieved in MTTF during test is very highly correlated with the amount of testing. It will be seen later in this paper that

this fact provides the means for relating schedules to reliability requirements. Thus reliability can be closely linked with project schedules. Furthermore, the cost of testing is highly correlated with MTTF improvement. Since two of the key process attributes that a manager must control are schedule and cost, reliability can be very intimately tied in with project management.

3) Software reliability can be used as a means for monitoring the operational performance of software and controlling changes to the software. Since change usually involves a degradation of reliability, a reliability performance objective can be used as a means for determining when software changes will be allowed and perhaps even how large they can be.

The purpose of this paper is to summarize the state of the art of software reliability measurement, touching also on the historical development of the field. The focus tends to be on concepts and techniques that are to some degree supported by data or actual use. In concentrating on software reliability *measurement*, the author does not mean to imply that investigation of the *causes* and means of reducing software faults is unimportant. However, it is helpful to know how bad a situation is before one attacks it and just what economic benefits can be obtained from improving things, so that one knows how hard to attack. A discussion of the causes and cures for software faults is a paper (and perhaps a book) in itself and is related to such topics as complexity and software psychology.

The bibliography has been chosen from a very large collection of material that was examined; all references are listed either because they are used in this paper or because they present significant additional material useful to the reader. It was felt that a selective rather than an exhaustive approach would be most beneficial.

II. Software Reliability Versus Hardware Reliability

The field of hardware reliability has been established for some time, so it is natural to inquire as to what extent (if any) theoretical techniques derived for hardware reliability may be applicable to software. It turns out that the division between hardware and software reliability is somewhat artificial. Both may be defined similarly. The source of failures in software is design faults, while the source in hardware is physical deterioration. The concepts and theories developed for software reliability could really be applied to any design activity, including hardware design. Once a software (design) defect is properly fixed, it is in general fixed for all time. Failure usually occurs only when a program (design) is exposed to an environment that it was not developed or tested for. Although manufacturing can have major impact on the quality of physical components, the replication process for software (design) is essentially trivial and can be performed to very high standards of quality.

Probably the reason that the "design reliability" concept has not been applied to hardware to any extent is because the probability of failure due to wear and other physical causes has usually been much greater than the probability of failure due to an unrecognized design problem. It was possible to keep hardware design failures relatively low because hardware was generally less complex logically than software (although this may be changing nowadays with integrated circuits). Hardware design failures had to be kept low because retrofitting of manufactured items in the field was very expensive.

Despite the foregoing differences, software reliability theory can be developed in a manner that is compatible with hardware reliability theory, so that system reliability figures may be computed using standard hardware combinatorial techniques [4], [5].

Although there are many similarities between hardware and software reliability, and one must not err on the side of assuming that software always presents unique problems, one must also be careful not to carry analogies too far.

III. Basic Concepts

Software reliability will be defined in a manner very similar to hardware reliability: it is the probability of failure-free operation of a software component or system in a specified environment for a specified time. A *failure* is defined as an unacceptable departure of program operation from requirements. A *fault* is the software defect that causes a failure. The term *error* will be used to indicate the human action that results in a fault [6]. Software *availability* is usually defined as the expected fraction of time during which a software component or system is functioning acceptably.[1] Input data is not considered part of a software component or system; its reliability is associated with the functioning of an external component. Data generated by the software is also not *part* of the software. (It may be a required output which can fail.) Only data constants entered at compilation time by a programmer can be considered part of the program.

The term "acceptable" implies that the user must determine what he considers to be a failure; this usually depends on the effect of the particular behavior of the system in question on the user's operation costs, etc. In fact, the situation is often more complex than "acceptable" or "unacceptable:" the user may wish to establish several classes of failures of differing severities and define reliability requirements for each class.

The foregoing definition of software reliability is an *operational* one; it has been adopted because it offers the greatest utility to software engineers and managers, since it directly measures the impact on the user of a system. Some think more in terms of the number of faults in a program. This is a *state* definition. Although this quantity will not be used to define reliability, it will be applied in determining the time and costs for fault removal.

Reliability [denoted $R(t)$] represents the probability that failure will *not* occur in time t; let

$$F(t) = 1 - R(t) \tag{1}$$

represent the probability that it *will*. The hazard rate $z(t)$ represents an instantaneous failure rate with respect to time, or the failure rate given that a system or component has survived until now. Thus

$$z(t) = \frac{F'(t)}{R(t)}. \tag{2}$$

It is readily shown [5, p. 182] that

$$R(t) = \exp\left[-\int_0^t z(x)\,dx\right]. \tag{3}$$

The mean time to failure T is defined as the expected value of the failure interval; it can be related to the reliability as

[1] This is an "interval" definition; a "pointwise" definition is also possible [7], [8].

follows [5, p. 197]:

$$T = \int_0^\infty R(t)\,dt. \tag{4}$$

If the hazard rate is constant (the situation when the failure intervals are distributed exponentially), the MTTF is readily shown [5, p. 198] to be its reciprocal.

Hecht [9] has defined three principal functions of software reliability: measurement, estimation, and prediction. Reliability *measurement* (used here in a more specialized and precise sense than that of the title of this paper) is based on failure interval data obtained by running a program in its actual operating environment. Reliability *estimation* refers to the process of determining a metric based on failure interval data from a test environment. Note that estimation can be performed with respect to present or future reliability quantities. The term software reliability *prediction* is defined as the process of computing software reliability quantities from program characteristics (*not* failure intervals). Typically, software reliability prediction takes into account factors such as the size and complexity of a program, and it is normally performed during a program phase prior to test. Future estimation might be thought of by some as prediction, but a careful and deliberate distinction in terminology will be made here.

IV. History

The first paper on software reliability appears to have been published (although not widely circulated) in 1967. Hudson [10] viewed software development as a birth and death process (a type of Markov process) in which fault generation (through design changes, faults created in fixing other faults, etc.) was a birth, and fault correction was a death. The number of faults existing at any time defined the state of the process; the transition probabilities related to the birth and death functions. He generally confined his work to pure death processes, for reasons of mathematical tractability. As the result of his assumptions, he obtained a Weibull distribution of intervals between failures. Data from the system test phase of one program is presented. Reasonable agreement between model and data is obtained if the system test phase is split into three overlapping subphases and separate fits made for each. The rate of fault correction was assumed to be proportional to the number of remaining faults and to some positive power of time (i.e., the rate of fault correction was assumed to increase with time).

The next major step was made by Jelinski and Moranda in 1971 [11]. They assumed a hazard rate for failures that was piecewise constant and proportional to the number of faults remaining. The hazard rate changed at each fault correction by a constant amount, but was constant between corrections. They applied maximum likelihood estimation to determine the total number of faults existing in the software and the constant of proportionality between number of faults remaining and hazard rate. Moranda [12] has also proposed two variants of what he terms the "deeutrophication process." In one, the hazard rate decreases in steps that form a geometric progression (rather than being of constant amount). The second (called the "geometric Poisson" model) has a hazard rate which also decreases in a geometric progression, but the decrements occur at fixed intervals rather than at each failure correction. One of the important points made in their discussion of the original model has to do with the distinction between software and hardware failure rates. They point out that it is the detection and times between detections of software faults through program operation which correspond to the failures of hardware.

Almost simultaneously with the Jelinski and Moranda work, Shooman [13] published a similar model which introduced some new concepts. Shooman also assumed that the hazard rate was proportional to the number of remaining faults. He viewed the hazard rate as being determined by the rate at which execution of the program resulted in the remaining faults being passed. Thus the hazard rate depended on the instruction processing rate, the number of instructions in the program and the number of faults remaining in the program. He postulated a bulk constant to account for the fact that program structure (e.g., loops) could result in repetitions that did not access new instructions and hence new faults. The number of remaining faults, of course, depended on the number of faults corrected, and the profile of the latter quantity as a function of time was assumed to be related to the project personnel profile in time. Several different fault correction profiles were proposed; the choice would depend on the particular project one was working with. Shooman and Natarajan have also proposed more complex models of the fault generation and correction process [14].

Another early model was proposed by Schick and Wolverton [15], [16]. The hazard rate assumed was proportional to the product of the number of faults remaining and the time spent in debugging. The amount of debugging time between failures has a Rayleigh distribution. Thus the size of the changes in hazard rate (at fault corrections) increases with debugging time.

Schneidewind [17], [18] approached software reliability modeling from an empirical viewpoint. He recommended the investigation of different reliability functions such as the exponential, normal, gamma, and Weibull and choosing the distribution that best fit the particular project in question. In looking at data, Schneidewind found that the best distribution varied from project to project. He indicated the importance of determining confidence intervals for the parameters estimated rather than just relying on point estimates. He was the first to suggest consideration of two kinds of time, operating time of the program and cumulative test time. Schneidewind also suggested that the time lag between failure detection and correction be determined from actual data and used to correct the time scale in forecasts [19].

In early 1973, Musa commenced work on an execution time model of software reliability, leading to results that were published in 1975 [20]. This theory built on earlier contributions, but also broke new ground in a number of ways. He postulated that execution time (i.e., the actual processor time utilized in executing the program) was the best practical measure of failure-inducing stress that was being placed on the program. Hence, he concluded that software reliability theory should be based on execution time rather than calendar time. Most calendar time models do not account for varying usage of the program in either test or operation. An execution time model is superior in ability to model, in conceptual insight, and in predictive validity.

Musa considered execution time in two respects; the operating time of a product delivered to the field, and the cumulative execution time that had occurred during test phases of the development process and during post delivery maintenance. The hazard rate was assumed to be constant with respect to operating time but would change as a function of the faults remaining and hence the cumulative execution time. Use of two kinds of time separates fault repair and growth phenomena from program operation phenomena and greatly simplifies both conceptual thinking and analysis. Otherwise, the probability distributions would have to be "at least" Weibull and probably more complex. It is difficult to relate the parameters of these more complex (than time-varying exponential) distributions to meaningful software development variables.

Musa assumed that the fault correction rate was proportional to the fault detection or hazard rate, making consideration of debugging personnel profiles unnecessary.

These concepts were tested in real time on four development projects with excellent results. With this formulation, the variability in models from project to project noted by Schneidewind and Shooman did not occur. Thus the modeling approach became universal and much easier to apply. Hecht [21] in at least two cases has independently verified the simplification resulting from looking at software reliability as a function of execution time rather than calendar time.

A calendar time component was developed for the model that related execution time to calendar time, allowing execution time predictions to be converted to dates. The calendar time component is based on a model of the debugging process.

It appears that the execution time theory has been tested more thoroughly and applied more extensively than any other software reliability model. The assumptions on which it is based have been carefully stated and their validity examined with data from 16 software systems [22], [23]; the results are good. Corroborating evidence for one of the execution time model assumptions (i.e., that the hazard rate is proportional to the number of remaining faults) has been presented by Miyamoto [24]. This model has been applied for many different purposes and a great deal of experience has been gained in its use [25]-[28].

A generally different approach to software reliability measurement has been taken by Littlewood [8], [29]-[31]. Littlewood takes a Bayesian approach; that is, he views software reliability as a measure of strength of belief that a program will operate successfully rather than the outcome of an (possibly hypothetical) experiment to determine the number of times a program would operate successfully out of, say, one hundred executions. Both Littlewood and Musa assume (as do most other researchers) that failures occur randomly during the operation of the program, with the hazard rate constant with respect to the operating time of the program (in the sense defined by Musa). However, while Musa postulates simply that the value of the hazard rate is a *function* of the number of faults remaining, Littlewood models it as a *random process* in the failures experienced. (Hence, the hazard rate is really a conditional one.) Littlewood proposes various functional forms for the description of the variation of the parameters of the random process with the number of failures experienced. The constants that produce the best fit for each functional form are determined. Then the functional forms are compared (at the optimum values of the constants) and the best fitting form is selected.

Recently Littlewood has proposed a differential fault model [32] which assumes that faults make different contributions to the hazard rate of a program, based on the hypothesis that different faults are accessed with different frequencies.

Littlewood points out that in viewing the hazard rate as a random process he takes account of the uncertainty of repair involved in debugging. Musa handles imperfect debugging on an average basis, contending that the failure to failure uncertainties in the process are second order effects not worth modeling.

Goel and Okumoto [33] present an imperfect debugging model that is intermediate in complexity between the Musa and Littlewood models. It is based on a view of debugging as a Markov process, with appropriate transition probabilities between states. Several useful quantities can be derived analytically, with the mathematics remaining tractable. In another paper, Goel and Okumoto [34], reasoning from assumptions similar to those of Jelinski and Moranda, describe failure detection as a nonhomogeneous Poisson process. The cumulative number of failures detected and the distribution of the number of remaining failures are both found to be Poisson.

A theoretically appealing approach to software reliability is the input space approach [35], [36]. In this approach, all possible sets of input conditions for a program are enumerated and one determines the proportion of these that results in successful operation. Unfortunately, this approach suffers from two deficiencies

1) the large number of possible input sets for any useful program makes it impractical;
2) the proportion of input sets that execute successfully is not particularly meaningful to software engineers; MTTF is more useful since it is relatable to costs and other impacts of failure and since it is compatible with hardware reliability theory; to obtain MTTF from the input space approach requires that each input state be weighted by its execution time and frequency of operation, adding even more complication.

The author feels however, that the input space approach is valuable as a concept and may be one of the keys to a deeper understanding of the test compression factor (discussed later).

V. Software Reliability Models

We will now look at two of the leading software reliability models, Musa's execution time model and Littlewood's Bayesian model, in some detail as examples of two of the principal approaches to this subject. The principal objective of a software reliability model is to predict behavior that will be experienced when the program is operational. This expected behavior changes rapidly and can be tracked during the period in which the program is tested. Reliability or MTTF generally increases as a function of accumulated execution time. The parameters of the model can be determined to some degree of accuracy from program characteristics such as size, complexity, etc., before testing commences. Thus, during the preliminary planning, requirements generation, design, and coding phases of a project, expected behavior in the field can be predicted as a function of execution time accumulated during system test. The predicted behavior indicates what would happen if test were terminated at the prediction point and the program placed into operational use in the field. Much better estimates of parameters can be made during the test phases.

Both models have certain assumptions in common, in addition to having some peculiar to themselves. Both assume that tests represent the environment in which the program will be used, that all failures will be observed and that failure intervals are independent of each other. They also both assume that failure intervals are exponentially distributed with respect to the operating time of the program (in the case of the Littlewood model the exponential distribution is conditional on the failure rate, since the failure rate is assumed to be a random variable). Representativeness of testing and observation of failures depend on a well-planned and well-executed testing effort. Evidence gathered to date indicates that the independence and exponential distribution assumptions are soundly based [22]. The models assumed systems that have been completely integrated before failure data is collected, error being introduced when they are not. Work in progress [37] may permit incompletely integrated (partial) systems to be modeled more accurately by use of a procedure for adjustment of failure intervals.

It is best, of course, if testing is directly representative of what will be encountered in the operating environment of the program. However, for reasons of efficiency, one usually wishes in test to remove the redundancy inherent in the operational environment, thereby speeding up testing. Musa [25] has proposed the use of a "testing compression factor" to account for this removal of redundancy. If, for example, one hour of test represents twelve hours of actual operation, then the testing compression factor is 12.

Both the Musa and Littlewood models (and most others) view failures as occurring randomly in time. This randomness is primarily due to the randomness with which parts of the program are executed, due to various functions being required by the user in time, and to a small extent due to the randomness with which the faults associated with the failures are created throughout the program.

VI. Execution Time Model

The execution time model deals with two kinds of execution time

1) the cumulative execution time (denoted τ) that clocks development activity and is measured up to the reference point at which reliability is being evaluated, and
2) the program operating time (denoted τ') which is the execution time projected from the reference point into the future on the basis that no further fault correction is performed.

The model consists of two components, the execution time component and the calendar time component. The former component characterizes reliability behavior as a function of cumulative execution time τ. The latter component relates cumulative execution time τ to cumulative calendar time t.

The execution time component of the model, in addition to the assumptions common to it and the Bayesian model noted above, postulates that

1) the hazard rate is proportional to the number of faults remaining N, and
2) the execution time rate of change of the number of faults corrected is proportional to the hazard rate.

The constant of proportionality in the second assumption above is called the fault reduction factor B. It was designed to account for three effects that are each assumed to be proportional to the hazard rate

1) fault growth due to new faults that are spawned in the process of correcting old faults;
2) faults found by code inspection that was stimulated by the detection of a related fault during test, and
3) a proportion of failures whose causative faults cannot be determined and hence cannot be corrected.

Based on the foregoing assumptions, a number of relationships can be derived [20]. The net number of faults detected and corrected n is shown to be an exponential function of the cumulative execution time τ,

$$n = N_0 \left[1 - \exp\left(-\frac{C\tau}{M_0 T_0}\right)\right] \tag{5}$$

where N_0 is the number of inherent faults in the program, M_0 is the total number of failures possible during the maintained life of the software, T_0 is the MTTF at the start of test and C is the testing compression factor. "Maintained life" is the period extending from the start of test to discontinuance of program failure correction. (Once program failure correction has been discontinued, the number of failures becomes dependent on the program lifetime and (5) does not hold.)

Failures and faults are related through

$$N_0 = BM_0 \tag{6}$$

and

$$n = Bm \tag{7}$$

where m is the number of failures experienced. Thus the fault reduction factor B may be seen as an expression of the net fault removal per failure.

The number of failures experienced is also an exponential function of the cumulative execution time

$$m = M_0 \left[1 - \exp\left(-\frac{C\tau}{M_0 T_0}\right)\right]. \tag{8}$$

This relationship is illustrated in Fig. 1.

The present mean time to failure T is also shown to be related to the cumulative execution time

$$T = T_0 \exp\left(\frac{C\tau}{M_0 T_0}\right) \tag{9}$$

illustrated in Fig. 2. MTTF's are measured in execution time. Note that the present MTTF increases as testing proceeds. Reliability R for an operational period τ' is given by

$$R = \exp\left(-\frac{\tau'}{T}\right). \tag{10}$$

If a MTTF objective of T_F has been set for the program, then the number of failures that must be experienced and the additional execution time required to meet this objective can be derived as

$$\Delta m = M_0 T_0 \left[\frac{1}{T} - \frac{1}{T_F}\right] \tag{11}$$

and

$$\Delta\tau = \frac{M_0 T_0}{C} \ln\left(\frac{T_F}{T}\right). \tag{12}$$

The calendar time component of the model relates execution time and calendar time by looking at the constraints that are involved in applying resources to the project. The rate of testing is constrained by the failure identification or test team personnel, the failure correction or debugging personnel and the computer time available. The quantity of these resources available to the project may be more or less freely established in the early stages of the project but increases are generally not feasible during the system test phase because of the long lead times required for training and computer procurement. At any given value of τ, one of these resources will be limiting and

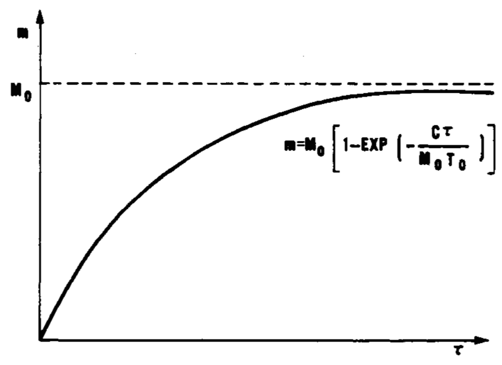

Fig. 1. Failures experienced versus execution time.

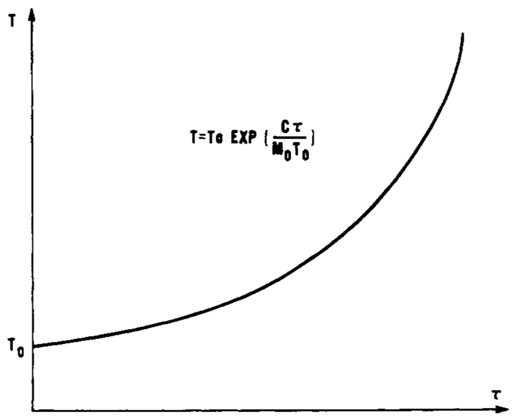

Fig. 2. Present MTTF versus execution time.

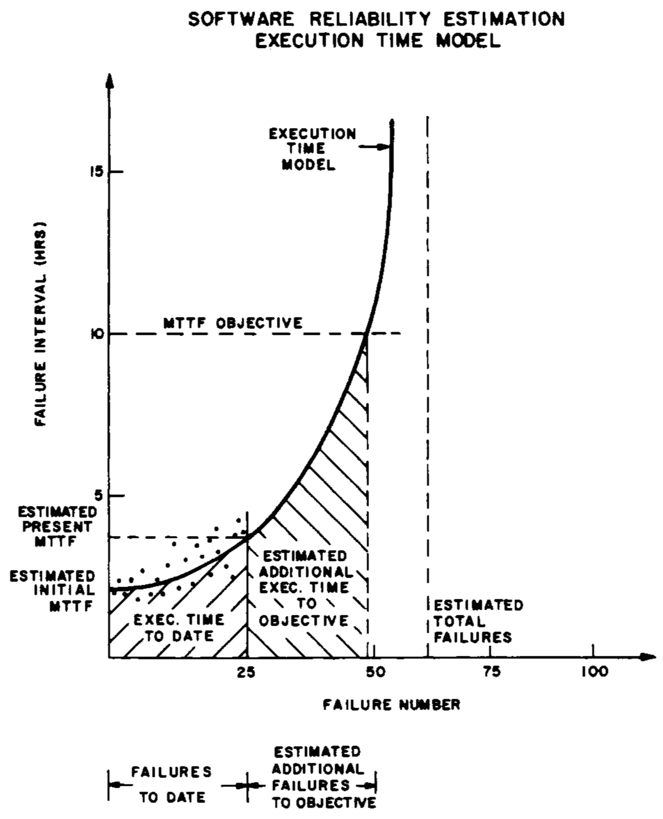

Fig. 3. Program parameter reestimation and project status monitoring.

will be determining the rate at which execution time can be spent per unit calendar time. A test phase may consist of from one to three periods, each characterized by a different limiting resource. The following is a common scenario. At the start of testing one identifies a large number of failures separated by short time intervals. Testing must be stopped from time to time in order to let the people who are fixing the faults keep up with the load. As testing progresses, the intervals between failures become longer and longer and the failure correction personnel are no longer fully loaded; the test team becomes the bottleneck. The effort required to run tests and analyze the results is occupying all their time. Finally, at even longer intervals, the capacity of the computing facilities becomes limiting. The calendar time component of the model is derived by assuming that the quantities of the resources available are constant for the remainder of the test period and that the resource expenditures $\Delta\chi_k$ associated with a change in MTTF can be approximated by

$$\Delta\chi_k = \theta_k \Delta\tau + \mu_k \Delta m \tag{13}$$

where $\Delta\tau$ is the increment of execution time, Δm is the increment of failures experienced, θ_k is an execution time coefficient of resource expenditure, and μ_k is a failure coefficient of resource expenditure. The k is an index indicating the particular resource involved. It is further assumed that failure identification personnel can be fully utilized, that computer utilization is constant and that the utilization of failure correction personnel is established by limitation of failure queue length for any debugger, assuming that failure correction is a Poisson process and that debuggers are randomly assigned in time.

The reader is referred to [20] for the formula that relates execution time and calendar time. The formula requires knowledge of parameters that relate to the resources required for failure identification and correction, the resources available and the resources that can be utilized (due to bottlenecks, as expressed by queueing theory).

A number of parameters must be evaluated in order to specify the execution time model completely, including the estimation of predicted completion dates. (If the latter are not required, only three parameters will suffice.) The parameters may be grouped into four categories: planned, debug environment, test environment, and program. The planned parameters are established by project objectives and available resources. The debug environment parameters relate to the resources required for failure identification and correction. It is hoped that ultimately values of these parameters can be determined for all software projects or for large classes of projects. The values of the debug environment parameters may be related to such factors as batch debugging versus interactive debugging, debugging aids available, computer used, language used, administrative and documentation overhead associated with corrections, etc. Herndon and Keenan [38] have investigated some of these factors. The test environment parameter is the testing compression factor that has already been discussed. There are two program parameters, M_0 and T_0. They must initially be estimated from characteristics of the program itself (i.e., by reliability prediction). However, once data is available on failure intervals, these parameters may be reestimated. The accuracy with which they are known generally increases with the size of the sample of failures.

Maximum likelihood estimation is used to reestimate M_0 and T_0 as testing progresses based on the execution time intervals between failure experienced in testing. Fig. 3 illustrates the process conceptually. Note that the execution time model is characterized by a curve whose vertical axis intercept is T_0, the

```
BASED ON SAMPLE OF     136 TEST FAILURES
EXECUTION TIME IS       25.34 HRS
MTTF OBJECTIVE IS       27.80 HOURS
CALENDAR TIME TO DATE IS     96 DAYS
PRESENT DATE:11/ 9/73

                         CONF. LIMITS           MOST           CONF. LIMITS
                    95%    90%    75%     50%  LIKELY   50%    75%    90%     95%

TOTAL FAILURES      136    136    136     138    142    148    152    163     182

INITIAL MTTF(HR)  0.522  0.617  0.701   0.744  0.847  0.949  0.992   1.08    1.17

PRESENT MTTF(HR) 999999 999999 999999    30.9   20.4   14.5   12.5   9.53    7.05

PERCENT OF OBJ    100.0  100.0  100.0   100.0   73.4   52.0   45.1   34.3    25.4

         *** ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE ***

FAILURES              0      0      0       0      2      5      7     12      23

EXEC. TIME(HR)        0      0      0       0   2.46   6.09   7.94   12.4    19.4

CAL. TIME(DAYS)       0      0      0       0  0.958   2.85   4.03   7.39    13.8

COMPLETION DATE  110973 110973 110973  110973 111273 111473 111673 112173 112973
```

Fig. 4. Sample project status report.

estimated initial MTTF, and which approaches an asymptote whose horizontal axis value is M_O, the estimated total failures. The values of these parameters are chosen to maximize the likelihood of occurrence of the set of failure intervals (plotted as dots in the figure). The failure interval is represented along the vertical axis and the sequential failure number on the horizontal axis. Note that several other quantities related to the model can be read from the figure. Since reestimation of the parameters is a statistical process, one also wishes to determine confidence intervals. The curve may now be viewed as a band, with its thickness increasing with the magnitude of the confidence interval (a 90 percent confidence interval will have a thicker band than a 75 percent one). The confidence interval for T_O will be given by the range of vertical axis intercepts and the confidence interval for M_O will be given by the range of asymptotes approached.

The parameter M_O may be estimated, prior to test, from the number of inherent faults N_O and the fault reduction factor B. It has long been assumed that the size of a program has the most impact on the number of faults it contains; this has been verified by a number of researchers [36], [39]-[41]. Some data has been taken on average fault rates per delivered executable source instruction. A range of 3.36 to 7.98 faults/1000 delivered executable source instructions for assembly language programs at the start of system test has been reported [20], [40], [42].

The question has been raised of the possibility of obtaining better predictions of the number of faults by developing and using expressions of program complexity in the prediction process [43]-[46]. Several different complexity measures have been proposed; they fall into the general categories of structural complexity and textual complexity [47]-[49]. In some cases size is subsumed by the complexity measure. This field is an active area for research at the present time. It is not clear which complexity metrics will prove to be the best ones. In fact, it may turn out that the concept of complexity is a multidimensional one. Some workers [39], [50], [51] have taken a multidimensional approach, using regression analysis. However, there will be considerable pressure for simplification, because the additional predictibility that can be added by use of a complexity measure is limited. Curtis discusses the field of software complexity in considerable detail in a companion paper in this issue. Since faults are introduced into software by a human error process, software psychology may have a great deal to contribute to unraveling this puzzle.

The parameter T_O may be predicted from

$$T_O = \frac{1}{fKN_O} \tag{14}$$

where f is the linear execution frequency of the program or the average instruction execution rate divided by the number of object instructions in the program and K is a fault exposure ratio which relates fault exposure frequency to "fault velocity." The fault velocity is the rate at which faults in the program would pass by if the program were executed linearly. The fault exposure ratio represents the fraction of time that the "passage" results in a failure.

It accounts for

1) programs are not generally executed in "straight line" fashion, but have many loops and branches, and
2) the machine state varies and hence the fault associated with an instruction may or may not be exposed at one particular execution of the instruction.

At present, K must be determined from a similar program. It may be possible in the future to relate K to program dynamic structure in some way. On six projects for which data is available, K ranges from 1.54×10^{-7} to 3.99×10^{-7}. The small range of these values may be due to program dynamic structure averaging out in some fashion for programs of any size; further investigation is required. Schneidewind [52] has suggested the use of simulation in relating program structure and failure characteristics; this may be a useful tool.

A program is available [53], [54] for the reestimation of M_O and T_O and the calculation of a number of quantities that are useful to the software manager and software engineer. A sample output from the program, taken during the system test phase of an actual project, is illustrated in Fig. 4. Maximum likelihood estimates and confidence intervals (the two bounds are placed on either side of the maximum likelihood estimate in the figure) for several useful quantities are provided in addition to those for total failures and initial MTTF (M_O and T_O, respectively). The range of the present MTTF T is printed along with a range of projected completion dates that indicate when a specified MTTF objective will be reached. Note that "999999" indicates "no upper limit." The number of additional failures that must be experienced Δm and the additional execution time $\Delta \tau$ required to reach the MTTF objective are given. The foregoing results are very useful to managers for planning schedules, estimating status and progress and determining when to terminate testing.

The resource parameters defined in the derivation of the calendar time component of the model pave the way for relating reliability requirements and cost, since each of the resources has an associated cost that is readily determined. An expression for system test cost (including failure identification and failure correction) as a function of reliability objective may be written. If other development costs can either be considered constant with respect to reliability or their variation is known, an expression for the total development cost as a function of reliability can be obtained. It is usually possible to determine the impact of reliability on operating costs for the system. Thus total life-cycle system cost can be determined as a function of reliability and optimization used to find the desired reliability objective. For a further discussion of the application of this and other techniques to system engineering see [28].

The effects of various program development techniques such as structured programming, better documentation, code reading and better test design on reliability can be determined by first finding the effects of the techniques on the parameters of the execution time model and then investigating the impact of these parameters on reliability. This two stage approach is desirable because the parameters can be more easily related to the techniques than reliability can. The effect on cost of the techniques can be computed by holding the reliability goal constant and subtracting the incremental cost of the techniques (such as additional program design, test planning or documentation cost) from the incremental savings achieved in failure identification and correction.

Fig. 5 illustrates the variation of present MTTF over the system test period for a project. Note the generally steady upward progress (more testing results in an increase in MTTF). The center line indicates the maximum likelihood estimate and the two other lines, the boundaries of the 75 percent confidence interval. The present MTTF is very sensitive to remaining faults when only a few remain; hence, its upper confidence limit can vary appreciably as each additional data point is added. The execution time model has been applied to the monitoring of project status during system test on a number of projects [23]; this figure illustrates its utility.

The execution time theory can assist a manager through simulation in reaching tradeoff decisions among schedules, costs, resources and reliability in a continuing fashion. It can also assist in determining resource allocations. One selects the parameters to be varied and then picks several values of each, looking at the effects on MTTF, project completion date, etc. These techniques are further discussed in [26].

Software measurement can presently be achieved with excellent accuracy. Fig. 6 presents data from an operational software system, indicating the maximum likelihood and 75 percent confidence bounds for the present MTTF. It has been found that variations in the MTTF and the size of the confidence interval are highly correlated with periods of fault correction or addition of new capabilities. The MTTF tends to increase and exhibit less dispersion during periods of fault correction and decrease and show more dispersion during periods of addition of new capabilities.

This fact can be usefully applied by a manager responsible for the maintenance of operational software in determining when to schedule the addition of new capabilities and when not to (i.e., as a guide for imposing change control) [27].

User comments indicate that the execution time model provides a good conceptual framework for understanding the software failure process. It is simple, its parameters are closely related to the physical world and it is compatible with hardware reliability theory. Most users feel that the benefits of applying the model already exceed the costs (which are basically data collection and computation). There have been two interesting side benefits. The process of defining just what constitutes a failure and the process of setting an MTTF objective have both been salutary in opening up communication between customer and developer.

Most of the assumptions that were made in deriving the execution time model have been validated [22] and experience has been gained with the model on a wide variety of software systems (more than 20 as of this date). There is no evidence to indicate any class of software to which the model would not apply. Thus this model has been tested and its validity examined much more thoroughly than any other proposed software reliability model.

VII. Bayesian Model

Although the Bayesian model shares many of the assumptions made for the execution time model, it also departs from it in significant aspects. Although both models assume that failures are distributed exponentially with respect to operating time of the program, the Bayesian model assumes that the failure rate parameter is a random process with respect to the failures experienced, while the execution time model views it as a *function* of the faults remaining. The execution time model defines a fault reduction ratio to account for imperfect correction on an *average* basis; the possibility of imperfect correction is included in the Bayesian model's definition of the random process.

Littlewood does not use MTTF as a concept in describing his model. In fact, he objects to its use, because he feels that there is a chance that a program might be perfect and therefore have an undefined or infinite MTTF [55]. Musa, on the other hand, never having met a perfect program in some 20 years of experience, has argued that the simplicity promoted by the use of the MTTF concept is worth the very infrequent possibility of handling an exceptional case in a different way. Littlewood's approach is to estimate percentiles of the distribution of the time to the next (or kth from now) failure.

The Littlewood model is based on a subjective or Bayesian interpretation of probability [8]. Littlewood argues that there is no sense in which one can envisage an ensemble of repetitions of the reliability measuring operation upon which a "frequentist" interpretation would depend. Since the Bayesian model is subjective, periods of failure-free operation cause the reliability to improve. The Musa model allows for reliability improvement during periods of failure-free operation by using a continuous approximation (i.e., an interpolation/extrapolation) to the failure detection and correction process. The Jelinski–Moranda model places a floor under the reliability improvement by assuming that a pseudofailure occurs at the end of the failure-free period.

Littlewood assumes two sources of randomness in software: randomness in the input data, which causes the intervals between failures to be random, and randomness in the mapping of input space to output space for the program. The mapping is the product of the program development process. Some

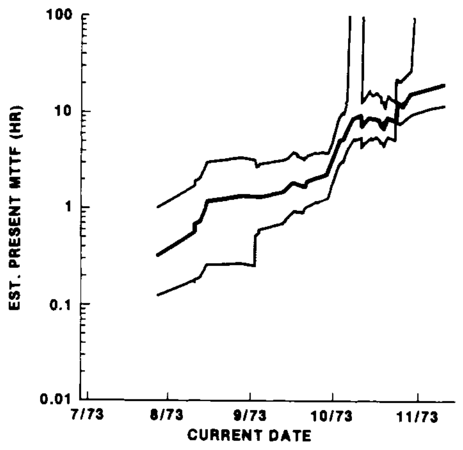

Fig. 5. Present MTTF history for system test period of typical project.

of the mappings are faulty. In the process of correcting the faults, new mappings are created which may or may not be faulty, i.e., the fault correction process introduces additional uncertainty. By assuming that the failure rate itself is a random process, Littlewood allows for this possibility. Thus two sources of uncertainty are compounded, resulting in a point process which is not Poisson. The Musa model incorporates the randomness in input data but does not view the failure rate as being random; any modeling improvement is not considered worth the extra complexity.

The Littlewood model usually leads more quickly to analytical difficulties than does the Musa model because of its more complex formulation.

The Littlewood model assumes that the failure intervals τ_i' have a conditional density function given by

$$f(\tau_i'|\lambda_i) = \lambda_i e^{-\lambda_i \tau_i'}. \tag{15}$$

The failure rate is assumed to be a random process with a conditional density function that is gamma

$$g(\lambda_i|\alpha, \psi(i)) = \frac{\psi(i)\{\psi(i)\lambda_i\}^{\alpha-1}\exp[-\psi(i)\lambda_i]}{\Gamma(\alpha)}. \tag{16}$$

The unconditional distribution of τ_i' is Pareto. The unknown quantities in the model are α and $\psi(i)$. Reliability growth is reflected in the $\psi(i)$ parameter.

Littlewood recommends that various families of growth functions be investigated for any given project and the one that provides the best fit to the data be chosen. The $\psi(i)$ function reflects the repair activities (including their uncertainty). There are many possible families of functions; two that have been suggested by Littlewood are

$$\psi(i) = \beta_1 + \beta_2 i \tag{17}$$

and

$$\psi(i) = \beta_1 + \beta_2 i^2. \tag{18}$$

Musa [56] has suggested the use of a rational function for $\psi(i)$, based on the concept that the parameter should be inversely related to the number of failures remaining. The suggested function was

$$\psi(i) = \frac{M_o T_o \alpha}{M_o - i} \tag{19}$$

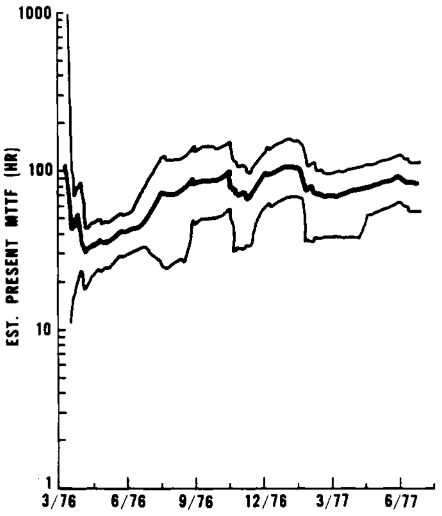

Fig. 6. Present MTTF history for typical operational software under maintenance.

where M_o is the number of failures expected during the maintained life of the software, T_o is the initial MTTF, α is the parameter of the gamma distribution and i is the index number of the failure.

Values of parameters for a given growth function and comparisons determining which growth function is best are established by testing goodness of fit to the data by means of the Cramér-von Mises statistic. This computation requires a sort of the data and minimization of the statistic over a multidimensional surface. These calculations are unfortunately very expensive in computer time, several orders of magnitude greater than those for the Musa model. Recent work by Iannino indicates that maximum likelihood estimation of parameters rather than a goodness of fit approach may reduce the difference in computation somewhat [57].

The Bayesian analysis assumes a uniform prior probability density function for α.

Littlewood's analysis leads to a hazard rate of the form

$$\lambda(\tau) = \frac{n}{\psi(n)}\left[\log \prod_{i=1}^{n} \frac{\psi(i) + \tau_i}{\psi(i)}\right]^{-1} \tag{20}$$

where $n - 1$ failures have previously occurred and τ_i represents the execution time since the $(i - 1)$th failure [29]. Note that the hazard rate changes discontinuously at each failure detection and correction and varies continuously with the cumulative execution time. Thus the hazard rate changes in accordance with one's subjective ideas of how reliability should change in a failure-free environment.

Littlewood also derives percentiles of the failure interval distribution for the failure interval between the $(n + k - 1)$th and the $(n + k)$th failures, where $k \geqq 0$. Note that n represents the next failure. The percentile $y_{q,n+k}$ is given by

$$y_{q,n+k} = \psi(n + k)\{\theta^{[(1-q)^{-1/(n+1)}-1]} - 1\} \tag{21}$$

for the 100q percentile [31], where θ is given by

$$\theta = \prod_{i=1}^{n} \frac{\tau_i + \psi(i)}{\psi(i)}. \tag{22}$$

The general Littlewood model has been verified against one set of project data with essentially good results [58].

The Littlewood approach results in somewhat greater mathematical elegance than the Musa model at the expense of greater complexity, which makes the model more difficult to understand and increases the computations required to apply it very substantially, as previously noted. It permits characterization of reliability decay as well as growth. The Musa model recognizes a situation of no growth or decay but does not predict its future course. This is not a material disadvantage, because prediction of decay is probably substantially poorer in quality than prediction of growth, the mechanisms being much less well known.

Littlewood has not developed (and probably cannot develop, due to analytical complexity) relationships for execution time required to reach a reliability objective in the general case (he has done this for the special case of the differential fault model, to be described). Musa considers the reliability or MTTF objective as a concept that is important and useful for software engineers and managers and determines the execution time required to reach it. The foregoing situation is an illustration of how the Littlewood model leads more quickly to analytical difficulties than does the Musa model because of its more complex formulation.

The Littlewood model does not include a calendar time component as does the Musa model. Hence, the attainment of reliability objectives can not be related to calendar dates, an item of great interest to software managers and engineers.

The Musa model has parameters which have readily understood physical interpretations but the Littlewood model does not. (Although the growth function [56] suggested by Musa for the Littlewood model remedies this situation somewhat.) Hence, it is not possible to apply current findings in software complexity research to the function of reliability prediction if the Littlewood model is used.

The differential fault model proposed by Littlewood may be viewed as an approximate particularization of the general Littlewood model, although it was not developed in this fashion. We say "approximate" because reliability growth is modeled through *both* parameters of the gamma distribution of the failure rate in the differential fault model. Littlewood hypothesizes that failures occur with different frequencies due to the variation in frequency with which different input states of the program are executed. The differential fault model thus postulates two sources of reliability growth; one is represented by each of the parameters of the gamma distribution. The first source is the detection and correction of faults and the second source follows from the hypothesis that failures occur with different frequencies: the most frequently occurring faults are detected and corrected first [30]. Littlewood considers that uncertainties in reliability growth probably result more from uncertainties in the relative frequencies of execution of different input states than uncertainties in fault correction.

Littlewood assumes that the random process representing the failure rate λ is a sum of random processes ϕ_i, where each ϕ_i is associated with a particular fault

$$\lambda = \sum_{i=1}^{N-n} \phi_i \tag{23}$$

where N represents the total number of inherent faults in the software and n represents the number of faults detected and corrected up to the present. The probability distributions associated with the individual faults prior to any debugging are assumed to be identical gamma distributions. The density function is given by

$$f(\phi) = \frac{\beta^\alpha \phi^{\alpha-1} e^{-\beta\phi}}{\Gamma(\alpha)}, \qquad \phi > 0 . \tag{24}$$

The distribution of each of the ϕ_i's in (23) must be found from Bayes' theorem; each will be a conditional distribution for ϕ_i given that fault i has not been corrected up until the present.

The overall hazard rate is given by

$$\lambda = \frac{(N-n)\alpha}{\beta + \tau} \tag{25}$$

where τ represents the cumulative execution time at the point where the hazard rate is measured. The density function of the execution time to the next failure τ' is Pareto

$$f(\tau') = \frac{(N-n)\alpha(\beta+\tau)^{(N-n)\alpha}}{(\beta+\tau+\tau')^{(N-n)\alpha+1}} . \tag{26}$$

It will be seen from (25) that

1) the failure rate drops by an amount $\alpha/(\beta+\tau)$ for each failure that is detected and corrected; corrections in the early part of testing (small τ) therefore result in greater reductions in failure rate than later ones;
2) the failure rate decreases during periods of failure-free operation due to the presence of τ in the denominator.

If $\alpha \to \infty$, $\beta \to \infty$ such that α/β is constant, equation (26) reduces to an exponential distribution.

Littlewood's differential fault model is new and has not yet been tested against actual data. However, it deserves consideration because the issue of differential fault frequencies it raises is an important one. It is a more complicated model than a model in which all faults are assumed to be uncovered with equal frequency. Whether or not the model will provide enough additional accuracy to justify its added complexity is not known at the present time. The model appears plausible if one assumes that programmers generate faults at a constant rate with respect to the possible input states in the program. This is based on the concept that programmers devote approximately equal time to thinking about each input state and the actions to be taken and implemented in the program when that state occurs. (A more naive viewpoint is that faults are distributed at a constant rate with respect to instructions; this model may represent the way programmers behave with respect to typographical errors, but the vast majority of errors are not typographical.) However, another hypothesis is possible. It may be that faults are distributed at a constant rate with respect to execution time. This would happen if programmers spent more time in designing the responses to input states that occur frequently, so that the fault rate for a particular input state would be (at least approximately) inversely proportional to its frequency of execution. This postulate is also plausible. The determination of which of the two postulates just discussed is in closer accord with reality (or is in close enough accord) will have to await evaluation with real data.

VIII. Software Reliability Combinatorics

The manner in which most software reliability models (including both the Musa and Littlewood models) have been developed results in a compatibility with hardware reliability theory that permits combination of hardware and software components in determining overall reliability of complete systems. The techniques of reliability budgeting or allocation commonly used in system engineering can therefore be applied to systems involving hardware and software components provided that these components are *concurrently functioning* [25]. A system is considered to be composed of concurrently functioning components if the satisfactory operation of the system is dependent on continuous satisfactory operation of some combination of these components. This situation is analogous to the analysis of reliability for a hardware system. The system is analyzed by drawing a "failure logic" diagram of the system and applying the combinatoric rules for AND and OR combinations of components developed in hardware reliability theory [4, ch. 9] [5, ch. 3].

If a system is composed of *sequentially functioning* components, i.e., if only one component functions at a time and the satisfactory operation of the system is dependent on the satisfactory functioning of each component when it is active then an approach developed by Littlewood may be used [59]. The system is assumed to consist of k components among which control is switched randomly according to a semi-Markov process; i.e., the transition probabilities between components are dependent only on the identity of the immediately preceeding component. The probability distributions of the sojourn times of the components in the active state are not restricted in any way. Failures for the ith component are assumed to occur in accordance with a Poisson process with rate λ_i. If the λ_i are small then the system process is Poisson with rate given by

$$\lambda = \frac{\sum_{i=1}^{k} \sum_{j=1}^{k} \rho_i p_{ij} \mu_{ij} \lambda_i}{\sum_{i=1}^{k} \sum_{j=1}^{k} \rho_i p_{ij} \mu_{ij}} \tag{27}$$

where the ρ_i are the equilibrium probabilities given by

$$\rho_i = \sum_{j=1}^{k} \rho_j p_{ji}. \tag{28}$$

The transition probability from component i to j is given by p_{ij} and μ_{ij} is the mean duration spent in component i before switching to component j. The rate λ_i is the reciprocal of the MTTF T_i.

IX. Availability

Availability may be computed for software as it is for hardware. Recall that availability was defined as the expected fraction of time during which a system will operate satisfactorily. If we let the time interval over which the measurement is made approach infinity, then the availability is given by the familiar ratio

$$A = \frac{T}{T+F} \tag{29}$$

where A is the availability, T is the MTTF and F is the mean time to repair (MTTR). Usually the MTTF applied here is a figure computed for serious failures and not those that involve only minor degradation of the system. Since it is generally not practical to hold up operation of the system while performing fault determination and correction in the field, MTTR is ordinarily determined as the average time required to restore the data base for a program, to reload the program, and resume execution. Markov process theory has often proved to be useful in looking at the details of availability behavior [33], [60], [61].

X. Current State of the Art and Research Needs

Software reliability measurement in the operational environment can at present be achieved with excellent accuracy. Hence, use of MTTF as a means for controlling change in operational software and as a basis for evaluating different software engineering technologies is very feasible [62].

The quality of software reliability estimation based on testing is most dependent on the representativeness of testing; it is essential to do a good job of test planning. Knowledge of the test compression factor is necessary if one wishes to know the absolute value of the MTTF or to make calculations dependent on it; it is not necessary when making relative comparisons (e.g., tracking progress on a project). At present one must estimate the test compression factor from measurement of a similar project in a similar test environment or be conservative and set it equal to 1. Research into the factors that influence the test compression factor would definitely be beneficial, particularly if this leads to ways of predicting it. The current quality of software reliability estimation could be characterized as good for present estimation and fair for future estimation. Present estimation (discussed in terms of the execution time model) requires the values of the test compression factor and the program parameters. Future estimation, in addition, necessitates knowledge of the planned and debug environment parameters. Data collection is needed on a number of projects to determine the values of the debug environment parameters and the extent to which they vary between different projects or different classes of projects. If they do vary, a study of the factors that influence them should be undertaken. Given the current state of software reliability estimation, status monitoring and tracking of projects can be accomplished with a relatively good level of quality. Estimation of project completion dates can presently be characterized as fair [62].

The function of software reliability prediction needs the most work [62]. However, it also offers great promise in terms of ultimate potential benefits, since it bears on how well system engineering and early planning decisions can be made. Such decisions often have the greatest impact on schedules and cost. All of the quantities required for software estimation (again, discussed in terms of the execution time model) are needed for software prediction as well. In addition, since one must predict the program parameters rather than estimate them from failure interval data, one requires the number of faults inherent in the software N_0, the fault reduction factor B, the fault exposure ratio K, and the linear execution frequency f.

One approach to determining the number of faults (already discussed) is based on size and complexity of the program. Data on the relationship between faults and size and complexity is just beginning to accumulate but much more is needed. Also, better ways of estimating size and complexity of a program in the requirements stage, before any code has been written, are required.

Some researchers have taken an empirical approach to predicting the number of faults in software. This approach variously goes by the names "error seeding" [63]-[65] or "bebugging" [66]. One generates artificial faults in a program in some suitable random fashion, unknown to the people who will be testing and debugging the software. It is assumed that these "seeded" faults are equivalent to the natural or original faults occurring in the program in terms of difficulty of detection. Based on that hypothesis and measurement of the proportion of seeded faults that have been found at any point in time, one can predict the number of natural faults, since it is assumed that the same proportion will have been discovered. For example, assume that 100 faults are seeded into a program containing an unknown number of natural faults. After one week of debugging (and the debugging may be either based on testing or on code reading), assume that 20 seeded faults and 10 natural faults have been discovered. Since the seeded faults discovered represent 20 percent of the total number it would be assumed that the natural faults discovered would be in the same proportion. Therefore, the total number of natural faults is 50. In actuality, the situation is a bit more complex than this, since we are dealing with samples of random variables and we would be making probability statements about the number of natural faults falling into some interval. Unfortunately, it has proved to be very difficult to implement this simple concept because of the great difficulty involved in randomly generating artificial faults that are equivalent to natural faults in difficulty of discovery. It has proved much easier to find the seeded faults; therefore, the total number of natural faults is usually underestimated.

There are additional complications involved in estimating the number of faults that occur for a program which undergoes a large number of releases. Belady and Lehman [67] have studied the problem and developed a theory of growth dynamics for systems with multiple releases. They indicate that complexity increases with the number of releases. The system tends to become more and more unstructured, probably due to the fact that different people work on the system, repairs and changes disturb the structure, not all changes are properly documented, etc. It was noted above that other researchers have found correlation between complexity and the number of faults existing in a system; consequently, the rate at which faults occur in multirelease systems can be expected to increase with time. Belady and Lehman point out that the increasing complexity trend can be countered by deliberate activities (which may be costly) to restructure the system.

Initial data appears to indicate that the fault reduction factor B may be relatively stable across different projects, but more study is required. The fault exposure ratio K is expected to be dependent on the dynamic structure of the program and the degree to which faults are data dependent. Further investigation of the properties of this ratio and the factors upon which they depend is very important if we are to obtain good absolute software reliability predictions.

Although the present state of the art of software reliability prediction needs considerable improvement in terms of obtaining good absolute values, it is often possible to conduct studies involving relative comparison with reasonable results [28].

Further evaluation of the relative merits of different software reliability models is desirable; this will probably sharpen the issues between them and reduce the number in contention. Unfortunately, the major quantitative comparative evaluation that has been performed to date did not include some of the most important models [68].

In summary, the field of software reliability metrics has made substantial progress in the last decade. It cannot yet provide a standard cookbook approach for widespread application. There are several problem areas that need work. However, it is clearly beyond the pure theory stage and it can provide practical dividends for those who make the modest investment in time required to learn and apply it.

Acknowledgment

The author is indebted to A. L. Goel, H. Hecht, A. Iannino, B. Littlewood, P. B. Moranda, G. J. Schick, N. F. Schneidewind, and R. W. Wolverton for their helpful comments and suggestions.

References

[1] L. Thomas, *The Medusa and the Snail.* New York: Viking Press, 1979, pp. 37-40.
[2] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. 2nd Int. Conf. Software Engineering* (San Francisco, CA), October 13-15, pp. 592-605, 1976.
[3] C. Nokes, "Availability and reliability of teleprocessing systems," *Comput. Commun.*, vol. 1 no. 1, pp. 33-41, Feb. 1978.
[4] D. K. Lloyd and M. Lipow, *Reliability: Management, Methods, and Mathematics*, 2nd ed. Redondo Beach, CA, published by the authors, 1977, ch. 9.
[5] M. L. Shooman, *Probabilistic Reliability: An Engineering Approach.* New York: McGraw-Hill, 1968, ch. 3.
[6] S. A. Gloss-Soler, Ed., *The DACS Glossary—A Bibliography of Software Engineering Terms*, Data and Analysis Center for Software Rep. GLOS-1, Rome Air Development Center, Rome, NY.
[7] R. E. Barlow and F. Broschan, *Mathematical Theory of Reliability.* New York: Wiley, 1965, p. 7.
[8] B. Littlewood, "How to measure software reliability and how not to," in *Proc. 3rd Int. Conf. Software Engineering* (Atlanta, GA), pp. 39-45, May 10-12 1978.
[9] H. Hecht, "Measurement, estimation, and prediction of software reliability," in *Software Engineering Technology—Volume 2.* Maidenhead, Berkshire, England: Infotech International, 1977, pp. 209-224. (Also in NASA Rep. CR145135, Jan. 1977.)
[10] G. R. Hudson, Program errors as a birth and death process, System Development Corp. Rep. SP-3011, Dec. 4, 1967.
[11] Z. Jelinski and P. B. Moranda, "Software reliability research," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed. New York: Academic Press, 1972, pp. 465-484.
[12] P. Moranda, "Predictions of software reliability during debugging," in *Proc. Ann. Reliability and Maintainability Symp.* (Washington, DC), pp. 327-332, Jan. 1975.
[13] M. Shooman, "Probabilistic models for software reliability prediction," in *Statistical Computer Performance Evaluation*, W. Freidberger, Ed. New York: Academic Press, 1972, pp. 485-502.
[14] M. L. Shooman and S. Natarajan, "Effect of manpower deployment and bug generation on software error models," in *Proc. Symp. Computer Software Engineering* (New York), pp. 155-170, 1976.
[15] G. J. Schick and R. W. Wolverton, "Assessment of software reliability," presented at 11th Annual Meeting of German Operations Research Society, Hamburg, Germany, Sept. 6-8, 1972; in *Proc. Operations Research*, Physica-Verlag, Wurzburg-Wien, 1973, pp. 395-422.
[16] ——, "An analysis of competing software reliability models," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 104-120, Mar. 1978.
[17] N. F. Schneidewind, "An approach to software in reliability prediction and quality control," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, pp. 837-847.
[18] ——, "A methodology for software reliability prediction and quality control," NTIS Rep. AD 754377.
[19] ——, "Analysis of error processes in computer software," in *Proc. 1975 Int. Conf. Reliable Software* (Los Angeles, CA), pp. 337-346, Apr. 21-23, 1975.
[20] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312-327, Sept. 1975.
[21] H. Hecht, "Comparison of calendar-time-based and execution-time-based software reliability measurements," to be published.
[22] J. D. Musa, "Validity of the execution time theory of software reliability, *IEEE Trans. Rel.*, vol. R-28, pp. 181-191, Aug. 1979.
[23] ——, Software reliability data, report available from Data and Analysis Center for Software, Rome Air Development Center, Rome, NY.
[24] I. Miyamoto, "Software reliability in on-line real time environment," in *Proc. 1975 Int. Conf. Reliable Software* (Los Angeles, CA), pp. 195-203, Apr. 21-23, 1975.
[25] J. D. Musa, "Software reliability measurement," in *Software phenomenology: Working papers of the Software Life Cycle Management Workshop* (Airlie, VA), pp. 427-451, Aug. 21-23, 1977; *J. Syst. Software*, to be published.
[26] ——, "The use of software reliability measures in project manage-

ment," in *Proc. COMPSAC 78* (Chicago, IL), pp. 493-498, Nov. 13-16 1978.

[27] P. Hamilton and J. D. Musa, "Measuring reliability of computation center software," in *Proc. 3rd. Int. Conf. Software Engineering* (Atlanta, GA), pp. 29-36, May 10-12 1978.

[28] J. D. Musa, "Software reliability measures applied to system engineering," in *1979 Nat. Computer Conf. Proc.* (New York), pp. 941-946, June 4-7 1979.

[29] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *J. Roy. Stat. Soc.–Series C*, vol. 22 no. 3, pp. 332-346, 1973.

[30] ——, "A Bayesian reliability growth model for computer software," in *Proc. 1973 IEEE Symp. Computer Software Reliability* (New York), pp. 70-77, Apr. 30-May 2 1973.

[31] ——, "A Bayesian reliability model with a stochastically monotone failure rate," *IEEE Trans. Rel.*, vol. R-23, pp. 108-114, June 1974.

[32] B. Littlewood, "What makes a reliable program–Few bugs, or a small failure rate?," submitted for *Nat. Computer Conf.* 1980.

[33] A. L. Goel and K. Okumoto, Bayesian software prediction models, Rome Air Development Center, Rome, NY, Rep. RADC-TR-78-155 (5 volumes).

[34] ——, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Trans. Rel.*, vol. R-28, pp. 206-211, Aug. 1979.

[35] W. H. MacWilliams, "Reliability of large, real time control software systems," in *Proc. 1973 IEEE Symp. Computer Software Reliability* (New York), pp. 1-6, Apr. 30-May 2, 1973.

[36] T. A. Thayer, Software reliability study, Rome Air Development Center, Rome, NY, Rep. RADC-TR-76-238.

[37] J. D. Musa and A. Iannino, "Estimation of software reliability of sequentially integrated systems," to be published.

[38] M. A. Herndon and N. T. Keenan, "Analysis of error remediation expenditures during validation," in *Proc. Int. Conf. Software Engineering* (Atlanta, GA), pp. 202-206, May 10-12, 1978.

[39] R. W. Motley and W. D. Brooks, Statistical Prediction of Programming errors, Rome Air Development Center, Rome, NY, Rep. RADC-TR-77-175.

[40] K. Akiyama, "An example of software system debugging," in *Proc. IFIPS Congr. 1971*. Amsterdam: The Netherlands, North-Holland, 1971, pp. 353-359.

[41] A. R. Feuer and E. B. Fowlkes, "Some results from an empirical study of computer software," in *Proc. 4th. Int Conf. Software Engineering* (Munich, Germany), pp. 351-355, Sept. 17-19, 1979.

[42] A. Endres, "An analysis of errors and their causes in system programs," in *Proc. 1975 Int. Conf. Reliable Software* (Los Angeles, CA), pp. 327-336, Apr. 21-23, 1975.

[43] M. H. Halstead, *Elements of Software Science.* New York: Elsevier North Holland, 1977.

[44] Y. Funami and M. H. Halstead, "A software physics analysis of Akiyama's debugging data," in *Proc. Symp. Computer Software Engineering* (New York), pp. 133-138, 1976.

[45] L. M. Ottenstein, "Quantitative estimates of debugging requirements," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 504-514, Sept. 1979.

[46] N. F. Schneidewind, H. M. Hoffman, "An experiment in software error data collection and analysis," *IEEE Trans. Software Engineering*, vol. SE-5, pp. 276-286, May 1979.

[47] L. A. Belady, "On software complexity," in *Proc. Workshop Quantitative Software Models* (Kiamesha Lake, NY), pp. 90-94, Oct. 9-11, 1979.

[48] B. Curtis, "In search of software complexity," in *Proc. Workshop, Quantitative Software Models* (Kiamesha Lake, NY), pp. 95-106, Oct. 9-11, 1979.

[49] J. C. Rault, "An approach towards reliable software," in *Proc. 4th. Int. Conf. Software Engineering* (Munich, Germany), pp. 220-230, Sept. 17-19, 1979.

[50] M. Lipow and T. A. Thayer, "Prediction of software failures," in *Proc. Symp. Reliability Maintainability, 1977*, pp. 489-494.

[51] C. E. Martin, A model for estimating the number of residual errors in COBOL programs, Ph.D. dissertation, Auburn Univ., Auburn, AL, June 1977.

[52] N. F. Schneidewind, "The use of simulation in the evaluation of software," *Comput.*, vol. 10, no. 4, pp. 47-53, Apr. 1977.

[53] J. D. Musa, *Program for software reliability and system test schedule estimation-user's guide*, available from the author.

[54] J. D. Musa and P. A. Hamilton, *Program for software reliability and system test schedule estimation–program documentation*, available from the author.

[55] B. Littlewood, "MTBF is meaningless in software reliability," *IEEE Trans. Rel.* (corres.), vol. 24, p. 82, Apr. 1975.

[56] J. D. Musa, private communication to B. Littlewood.

[57] A. Iannino, private communication to J. D. Musa and B. Littlewood.

[58] B. Littlewood, "Validation of a software reliability model," in *Proc. 2nd Software Life Cycle Management Workshop* (Atlanta, GA), pp. 146-152, Aug. 21-22, 1978.

[59] ——, "A semi-Markov model for software reliability with failure costs," in *Proc. Symp. Computer Software Engineering* (New York), pp. 281-300, 1976.

[60] A. K. Trivedi and M. L. Shooman, Computer Software Reliability: Many-State Markov Modeling Techniques, Polytechnic Inst. New York, Rep. POLY-EE/EP-75-005/EER 116, Mar. 1975.

[61] A. Costes, C. Landrault, and J. C. LaPrie, "Reliability and availability models for maintained systems featuring hardware failures and design faults," *IEEE Trans. Comput.*, vol. C-27, pp. 548-560, June 1978.

[62] J. D. Musa, "Software reliability modeling–Where are we and where should we be going?," in *Proc. 4th NASA Software Engineering Workshop* (Greenbelt, MD), Nov. 19, 1979, to be published.

[63] H. D. Mills, "On the statistical validation of computer programs," IBM Federal Systems Div., Gaithersburg, MD, Rep. FSC-72-6015, 1972.

[64] S. L. Basin, *Estimation of Software Error Rates via Capture-Recapture Sampling: A Critical Review.* Palo Alto, CA, Science Applications, Sept. 1973.

[65] B. Rudner, "Seeding/tagging estimation of software errors: Models and estimates," Rome Air Development Center, Rome, NY, Rep. RADC-TR-77-15.

[66] T. Gilb, *Software Metrics.* Cambridge, MA: Winthrop, 1977, p. 28.

[67] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 225-252, 1976.

[68] A. N. Sukert, "Empirical validation of three software error prediction models," *IEEE Trans. Rel.*, vol. R-28, no. 3, pp. 199-205, Aug. 1979.