NASA CR-145135

(NASA-CR-145135) MEASUREMENT, ESTIMATION, AND PREDICTION OF SOFTWARE RELIABILITY (Aerospace Corp., El Segundo, Calif.) 43 p HC A03/MF A01 CSCL 09B G3/61 — N77-21866 Unclas 24395

MEASUREMENT, ESTIMATION, AND PREDICTION OF SOFTWARE RELIABILITY

Prepared by

Herbert Hecht
Computer Directorate
Advanced Programs Division
Development Operations

January 1977

Advanced Programs Division
THE AEROSPACE CORPORATION
El Segundo, California

Prepared for

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

Contract No. NAS1-14392

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-145135		

4. Title and Subtitle	5. Report Date
MEASUREMENT, ESTIMATION, AND PREDICTION OF SOFTWARE RELIABILITY	January 1977
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Herbert Hecht	
	10. Work Unit No.
	505-07-31-02
9. Performing Organization Name and Address	11. Contract or Grant No.
The Aerospace Corporation Advanced Programs Division El Segundo, CA 90009	NAS1-14392
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics & Space Administration Washington, DC 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Mid-Term Report.

Mr. Gerard E. Migneault served as Langley Technical Monitor.

16. Abstract

Quantitative indices of software reliability are required for project management software function management and for research. The report defines the concept and application of three important indices: reliability measurement, reliability estimation, and reliability prediction. State of the art techniques for each of these procedures are presented, together with considerations of data acquisition. Failure classifications and other documentation for comprehensive software reliability evaluation are described.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Software reliability Reliability measurement Reliability estimation Reliability prediction Software error modeling Software failure classification	Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	46	$4.00

* For sale by the National Technical Information Service, Springfield, Virginia 22161

MEASUREMENT, ESTIMATION, AND PREDICTION
OF SOFTWARE RELIABILITY

Prepared by	Approved by
H. Hecht	G. W. Anderson, Group Director
Computer Directorate	Development Group Directorate
Advanced Programs Division	Advanced Programs Division
Development Operations	Development Operations

ACKNOWLEDGMENTS

Reviews of the paper by Mr. Myron Lipow and Prof. Martin L. Shooman were very helpful in arriving at the present format. Portions of this work were accomplished under Contract NAS1-14392 with the NASA Langley Research Center under the technical guidance of Mr. G. E. Migneault. The opinions expressed are those of the author and are not necessarily endorsed by the sponsoring agency or the reviewers.

TABLE OF CONTENTS

MEASUREMENT, ESTIMATION, AND PREDICTION OF SOFTWARE RELIABILITY

Herbert Hecht
The Aerospace Corporation

SUMMARY

Quantitative indices of software reliability are required for project management, management of the software function, and for research aimed at achieving more reliable software, e.g., through test tools and special languages. The purpose of this report is to clarify the applicability of reliability measurement, estimation, and prediction to software development and to describe state-of-the-art techniques for each of these procedures.

For reliability measurement, the software is operated over a period of time, segments of the operation are scored as failure or success, and from these scores a single indicator of measured reliability is generated. The most obvious application of software reliability measurement is to determine compliance with a reliability requirement that may have been imposed by contract or specification.

Estimation is performed by taking software reliability measurements on an existing program and modifying the result to represent the reliability in a different operating environment.

A typical application for reliability estimation is to determine during test whether an operational reliability goal can be met.

Reliability prediction is a statement about the reliability of a program based on size, complexity, or similar factors. Prediction of reliability can be made early in the project. It can be used for resource allocation to modules among the total software and for hardware/software tradeoffs.

Data requirements methods for data acquisition and computational techniques for all procedures are discussed. Failure classifications and other documentation for comprehensive software reliability evaluation are described.

INTRODUCTION

In the rapidly growing literature on the reliability of computer programs, and in several symposia specifically dedicated to this subject (Refs. 1,2) there has been comparatively little emphasis on quantitative indices of software reliability. Yet, any rational approach to software management requires quantifiable data on failure frequency, the cost of failures, and the cost and effectiveness of remedial measures. In addition, current research in fields of computer languages, development methodology, and test tools proceeds on the underlying assumption that it will produce improvements in reliability. Measurement of these improvements is therefore essential for evaluation of this research. Thus, while software reliability measurement by itself does not "solve" the software failure problem, it is an essential tcol for the direction of management activities and for demonstrating the accomplishments of research. This report aims to clarify some of the essential concepts in the numerical evaluation of software reliability and to introduce the reader to simple mathematical relations that have been found useful in the field.

It might be well to start here with a statement of what we mean by reliable software: It is software that is correct (capable of execution and yields correct results) and that meets other requirements imposed by the user such as timing and interfacing with the environment. This concept is consistent with an earlier statement "Software possesses reliability to the extent that it can be expected to perform its intended functions satisfactorily" (Ref. 3). The legalistically inclined reader will be justifiably concerned about any attempt to base measurement on "intended functions" but more restrictive formulations tend to prevent recognition of reliability problems arising from poorly drawn specifications. We see a need to evaluate software reliability against formally specified as well as against more loosely defined (and particularly implied) requirements and will attempt to deal with both of these conditions in the development of numerical indices in the following.

For reliability measurement the software is operated over a period of time, segments of the operation are scored as failure or success by the qualitative criteria cited above, and from these scores a single indicator of measured reliability is generated. Typically, the software will not be modified during the period of measurement, and the developed reliability numeric is applicable to the measurement period and then existing software configuration only.

Estimation of software reliability is performed by taking reliability measurements (as above) on an existing program and modifying the result to represent the reliability in a different operating environment. Estimation requires some quantifiable relationship between the measurement environment and the environment for which the estimate is to be valid.

Prediction of software reliability is any statement about the reliability of a computer program that is not based on measurement taken on that particular program. While this terse definition may permit predictions based on casting of dice or even less respectable methods (which, according to rumors, are sometimes utilized for that purpose), prediction is normally based on comparison of program length, complexity and environmental requirements with those of a program for which measurements exist.

Practical applications of the software reliability numerics are discussed in the next section. This is followed by three sections dealing with specific techniques of measurement, estimation, and prediction, respectively. The final section discusses classifications of software failure in relation to quantitative statements about software reliability.

APPLICATIONS OF MEASUREMENT, ESTIMATION, AND PREDICTION

Prediction of software reliability as it has been defined in the introduction is possible before the program has been written or even specified in much detail. Estimation of software reliability requires that the program exist but it may not necessarily be ready for operation in the intended environment. Software reliability measurement requires a "full up" availability of the program. In terms of the software life-cycle, it is therefore seen that the processes of measurement, estimation, and prediction occur in reverse sequence. However, any significant technical discussion of these subjects should proceed in the order listed because estimation of a quantity is best discussed after there is agreement on how the quantity is finally going to be measured. And a prediction made without a clear understanding of the quantitative formulation of software reliability in subsequent stages of the life cycle may be worse than useless.

Since software reliability measurement results in a quantitative index of reliability for software in its intended operating environment, the most obvious application of software reliability measurement is to determine compliance with a reliability requirement that may have been imposed by contract or specification. Another use of software reliability measurement is to determine in an already installed program that no deterioration of the reliability has taken place. Since software does not wear out, this latter application needs some explanation. Software failures are not necessarily due to obvious program errors. They can be due to unusual input data (out of range, unexpected data type), the computing environment, and to systems loading. To the extent that these factors can vary with time, it is therefore possible to see deterioration or improvement in the measured reliability. Reliability measurement may also be undertaken in support of research, e.g., to determine the value of new programming or test methods. A particularly important research application of reliability measurement is that necessary to develop and substantiate methods for reliability estimation and for reliability prediction.

A typical application for reliability estimation is to determine during development of a computer program whether the reliability goal expected for it can be attained. For this purpose, measurements will be taken over a limited period of time or with a limited set of test cases, and the results of this sample measurement will be interpreted in terms of a reliability measure at a future time (assuming reliability growth due to further testing and correction) and in a future operating environment. Reliability estimation may also be used to translate reliability measures from one computing environment into another one. A typical example in this area involves estimating the reliability of the operating system of a computer when new peripherals are to be added.

Reliability prediction is a numerical statement about the reliability of a computer program based on length, complexity of control structure, and other general characteristics rather than on data obtained from the program itself. Software reliability prediction can therefore be made very early in the program development cycle before the program itself is in existence. A typical application of software reliability prediction is for project management purposes: to scope the test and correction effort that may be involved for a specific program module or for an entire software system. Software reliability prediction furnishes one of the required inputs for forecasting operational down time that should be expected for a new software system. It would also seem appropriate to use software reliability prediction to guide program design to meet stated reliability requirements (in the sense that hardware reliability prediction is used to guide parts selection). However, this seems to be beyond the present state of the art. Some research in that area is just getting started and is referenced later.

In connection with these applications it is now possible to discuss whether the quantitative index of software reliability should be obtained with respect to a computer program specification or with respect to user requirements. When software measurement is being undertaken in order to determine compliance with a specific reliability requirement it is quite obvious that only deviations from the specification can be counted as failures. It would be rather unreasonable to expect anyone to undertake a contractual obligation with regard to ill-defined user expectations. On the other hand, if reliability measurement is being undertaken to select the best math package among a number of such programs, it may be quite appropriate to score as failure any deviation from an output which the user finds acceptable for the given input parameters. That the unacceptable output may conform to the program specification is rather immaterial in this case. Similarly, reliability estimation and reliability prediction may in some cases be made with respect to deviations from a specification where in other cases it might be with regard to satisfaction of user requirements. Rather than to champion one of these bases or the other, it is important to insist that the selected basis be clearly stated: Reliability with regard to the software specification, or reliability with regard to user requirements. The specifics of reliability measurement, estimation and prediction, are discussed in the following sections.

SOFTWARE RELIABILITY MEASUREMENT

In the most general sense, software reliability measurement is the identification of successful trials (S) among a predetermined total number of trials (N). The numerical index of software reliability obtained from this measurement is the ratio of successful trials to the total, or

$$R = S/N \tag{1}$$

The unreliability or failure ratio may be expressed as

$$U = F/N \tag{1a}$$

where F is the number of failures(1). This general definition can be directly applied in a conventional batch processing environment and in real-time systems dealing with discrete operations (e.g., telephone switching). For real-time systems dealing with continuous data streams (e.g., electric power distribution) a more natural and practical index is the mean-time-between-failures expressed as the predetermined total

---

(1)Although in keeping with common usage the title speaks of reliability measurement, etc., the numerical indices based on failures (as in Eq. (1a)) are frequently more useful: they have a natural origin at zero, can be more conveniently expressed as powers of 10, and have meaningful ratio relationships, e.g., the statement that one program is twice as reliable as another one usually implies an expectation of one-half the failure frequency.

running time (t) divided by the number of failures (F) in the interval 0 to t.

$$MTBF = t/F \tag{2}$$

The reciprocal of this quantity is the failure rate

$$u = F/t \tag{2a}$$

When no failure is observed in a predetermined time interval, one-third failure may be arbitrarily assigned in accordance with the hardware convention (Ref. 4). For software executing in an interactive manner, either the discrete or the continuous indices may be appropriate, depending on the application. Where essentially repetitive data sets are input into such a program, Eq. (1) may be applicable with S denoting the successfully processed data sets and N denoting the total number of data sets submitted. Where diverse data, involving different processing requirements, are submitted to an interactive system the second equation will be more applicable, with t here denoting CPU time. The discrete and continuous process equations can be related to each other when processing speed and associated factors are known. However, the failure mechanisms in the two environments are in most cases quite different so that commingling of their failure statistics is not desirable.

The reliability measures discussed above are meaningful only in the immediate environment in which they were obtained. If a contractual requirement calls for no more than one failure in 1000 runs on batch program ABLE, then the compliance or non-compliance with that requirement can indeed be determined from the reliability measurement described above. On the other hand, if the issue revolves about deficiencies in program BAKER where there have been 15 failures in the last 1000 runs, the reliability measurement of one failure per 1000 runs on program ABLE is not necessarily pertinent. ABLE might have been a very short program executing on a 16-bit minicomputer, while BAKER might be a very much longer program executing on a 60-bit scientific processor. If the quantitative indices obtained from reliability measurement are to be useful in the broader context, they must be normalized to account for such differences in exposure to failure between programs. A simple heuristic normalizing factor is program length. Corresponding to the elementary measured unreliability given in Eq. (1a) we can establish a normalized or global index of unreliability by adding a factor denoting program length (L) to the denominator, as shown in

$$U' = F/(N \times L) \tag{3}$$

The preferred numeric for L is the number of machine instructions submitted. The normalized measured reliability is then given by $R' = 1 - U'$.

If normalization for both program length and word length (W) is intended, Eq. (3) can be modified to

$$U'' = F/(N \times L \times W) \tag{4}$$

The value for W shall represent the average number of bits per instruction. In this formulation the index of unreliability in effect measures failures per bit submitted in the instruction deck.

For real-time systems operating in a continuous mode, a heuristic normalizing factor is the number of instructions executed per unit time (n). It permits meaningful comparison between failure frequencies on slow and fast machines on the basis of a normalized failure rate given by

$$u' = F/(t \times n) \tag{5}$$

If identical units of time are used to express t and n, then the dimension of the denominator in Eq. (5) is simply instructions executed. The normalized failure index given by Eq. (3) is based on failures per instruction submitted. This is related to, but not identical with, the normalized failure of Eq. (5). A further normalization for word length can also be incorporated for the continuous case. This yields

$$u'' = F/(t \times n \times W) \qquad (6)$$

which has the dimensions of failure per bit processed and is related to the index established for the discrete case in Eq. (4).

It has been suggested that numerical software reliability be defined as the ratio of all input data sets correctly processed to the total of all possible input data sets (Ref. 5). While the number of all possible data sets is less than infinite (due to the finite computer word length) it is still in many cases so large that measurement of software reliability by this method is not practical. Whether a specific set of input data results in correct or incorrect output depends on the specific path of program execution taken for that data set. Therefore, reliability measurement could also be based on a ratio of correct code segments (defined as non-branching sequences of statements) to the total number of segments. Note, however, that executing a code segment with one set of data does not assure correctness for all data, and one is again faced with the impossibility of comprehensive measurement. At present, the approaches based on input data classification are useful primarily for software reliability estimation and are discussed in that section.

Formal reliability measurement (e.g., for determining compliance with reliability provisions) will normally be based on Eqs. (1) or (2). In these circumstances the recordkeeping for both successful and unsuccessful runs may be required in any case and thus does not represent an obstacle to implementation of software reliability measurement. Normalizing factors may or may not be applied to the published data. In many other applications there is, however, an approach that yields normalized data directly and that avoids most or all of the recordkeeping associated with successful runs. Keeping track of the number of unsuccessful runs is still required but this is a less time-consuming task and will usually be necessary to comply with configuration management provisions. The simplified method makes use of a feature in the operating system of most large-scale computing installations that lists cumulative CPU time by job numbers. If a job number is assigned exclusively for operation of a given module (specifically excluding compilation, editing, etc.) then the number of instructions processed can be approximated to a fair degree by multiplying the CPU time by the nominal instruction speed for the given computer. When the number of observed failures is divided by this product, a normalized unreliability index corresponding to Eq. (5) is obtained. This can be converted to the form of Eq. (3) by multiplying by the ratio of instructions submitted to instructions executed, a factor that can usually be estimated for a given program. This procedure is particularly attractive for periodic monitoring of the reliability of computer programs.

In many applications it is desired to express the reliability of the total computing system. For these purposes it is significant that the measures of software reliability discussed here are in principle compatible with hardware reliability measures. For example, the expected number of failures for a specified time interval, obtainable from Eq. (2a), can be combined with the expected number of hardware failures for the same interval to yield a metric of total computer system reliability.

SOFTWARE RELIABILITY ESTIMATION

Data acquisition for software reliability estimation is almost indistinguishable from that for software reliability measurement. The significant difference is that in software reliability estimation the reliability (or failure) index is modified so as to yield the probability of failure of the functional software under test in a different environment or at a different time. The actual software reliability measurement is therefore interpreted as a sample measurement with the test runs representing a sample of operational runs.

If the test runs are completely representative of operational runs and if the software under test is expected to be used unmodified in the operational environment, then the reliability indices obtained by use of Eqs. (1) or (2) can be considered unbiased estimators of the reliability in the intended environment. In practice, of course, test cases are deliberately selected to stress the software more than the actual operating environment is expected to, and software will undergo changes (debugging) that presumably will reduce the likelihood of failure. Therefore the failure ratio (Eq. (1a)) or the failure rate (Eq. (2a)) obtained during test are pessimistic estimators of the equivalent indices that can be expected for the operational environment. Separate procedures for accounting for the severity of the test conditions and for the reliability growth expected due to debugging have been described in the literature and are synopsized below. A method for combining the two techniques is then presented.

A procedure for removing bias due to test data severity has been proposed by Brown and Lipow (Ref. 6). The probability of failure is ascribed to selection of input data. The total input data space is partitioned into subsets, Z_i, which are assumed to be homogeneous with regard to their failure-inducing properties. If, during test, N_j runs were made that used data from subset Z_j and produced F_j failures, then the estimated unreliability for data from this data set is given by

$$\hat{U}_j = F_j/N_j \tag{7}$$

The probability that failures due to data from Z_j will be observed in the operational environment will depend of course on the probability of then accessing data from Z_j which is given as $P(Z_j)$. An estimator of the operational unreliability U is therefore the sum over all data partitions of the unreliability index for a given partition multiplied by the probability that data from this partition will be encountered in the operational environment. This estimator is given by

$$\hat{U} = \sum_j (F_j/N_j) P(Z_j) \tag{8}$$

An equivalent estimator for continuous real-time programs can be formulated as

$$\hat{u} = \sum_j (F_j/t_j)\ P(Z_j) \tag{8a}$$

where t_j represents the time spent in processing data from partition Z_j.

The Brown and Lipow paper (Ref. 6) illustrates this technique on a Triangle Type Determination Program, that accepts input data sets consisting of three numbers. The program determines whether these numbers (interpreted as lengths of the sides) denote an equilateral, an isosceles, or a scalene triangle, or possibly no triangle at all. The data set partitioning is based on the type of triangle defined by the triad of numbers, e.g., Z_1 may represent all data sets that define an equilateral triangle. The application of this technique during test represents no particular problems, the only requirements above those for reliability measurement outlined in the preceding section are the typing of each data set to associate it with the appropriate Z_j. For proper estimation of the operational reliability, it is of course required that the probability of occurrence of the various input types, $P(Z_j)$, be accurately known. However, even under some mismatch of actual versus estimated probabilities, the resulting reliability will still be an acceptable estimate. The authors also point out some methods for selecting test cases that tend to desensitize the result against uncertainties in the usage probabilities.

Normal limitations on test budget and schedule, and the need to have a reasonable number of test cases in each category, obviously place limits on the number of categories that can be established. The question then arises whether all test cases falling within a given category are truly homogeneous with regard to failure probability. Even in terms of the simple Triangle Type Determination Program one must question whether integers, real numbers, very large or very small numbers all represent equally likely failure probabilities for the actual implementation of the routine that examines input data sets. Uncertainties in this regard can be removed by considering more detailed input set properties, although difficulties of determining probabilities of occurrence in the operational data set will obviously increase. An approach for more detailed analysis of the input data sets is described by Goodenough and Gerhart (Ref. 7)(2).

An implicit assumption in this technique is that the software itself will be transitioned without change from the test to the operational environment. In most situations, however, errors discovered during test will be corrected, causing failure

---

(2)A related partitioning of the input space based on access to specific paths in the program has been described by Shooman (Ref. 21).

estimates based on Eq. (8) to be unrealistically high. The amount of bias introduced into the estimate is obviously a function of the correction opportunities that will exist between the time of the sample measurements (Eq. (7)) and the time for which the reliability estimate is to be valid. So far, the techniques for modeling this reliability growth have been restricted to a homogeneous input data population, i.e., they regard all test data submitted during the sampling period to be as likely as leading to failure as data that might be submitted in the target environment (Refs. 8,9).

The reliability growth model assumes that the failure rate is directly proportional to the number of errors in a program (E) leading to the expression

$$u(t) = kE(t) \tag{9}$$

It is emphasized here that both u and E are expected to decrease during the testing process which is quantified in terms of program run time t. Specifically, at the beginning of test we may experience a high failure rate

$$u_0 = kE_0 \tag{10}$$

and at a later time, after C errors have been corrected, a lower failure rate

$$u_1 = k(E_0 - C) \tag{11}$$

Test records are depended on to furnish data on u_0, u_1, and C. Subtracting (Eq. 11) from (Eq. 10) we can then estimate k as

$$\hat{k} = (u_0 - u_1)/C \tag{12}$$

Further, by substituting the resulting value of k in (10) we obtain

$$\hat{E}_0 = u_0C/(u_0 - u_1) \tag{13}$$

There is some temptation to interpret $\hat{E}_0$ as a test termination criterion (i.e., to test until indeed $\hat{E}_0$ errors have been found). This should be discouraged because of the possible errors in the estimate that is obtained from a difference of two rates, and also because of the implicit assumption of homogeneity of error types in this technique which we have already mentioned. It is very difficult to hold that all errors will be of the same type (in terms of constant k), particularly as very low failure rates are approached. Instead, we would like to utilize these equations for estimation of failure rates at some future time at which the error types may still be expected to be reasonably close to those observed in the test conditions (particularly at the determination of u_1).
To keep the essentials of the approach in focus we introduce two simplifying assumptions:

a. Every software failure results in removal of an error, and

b. No new errors are introduced (in making corrections or by any other means).

Removal of these assumptions does not invalidate the methodology but leads to considerably more complex mathematical expressions (Ref. 10). The assumptions permit equating the failure rate with the correction or error removal rate

$$u = -dE/dt \quad (14)$$

This can be combined with (9) to yield

$$dE/dt = -kE(t) \quad (15)$$

which has the solution

$$E(t) = c_0 e^{-kt} \quad (16)$$

The constant of integration c_0 can be equated to E_0 and estimated by reference to the test results, e.g., u_0 or u_1. It is advisable to maintain records of total software operation time (t) during test to validate this estimation process. With k and c_0 known, Eqs. (14) and (15) can be combined to yield an estimate of failure rate as a function of operating time

$$\hat{u}(t) = \hat{k} E_0 e^{-\hat{k}t} \quad (17)$$

Again, it is cautioned that k is here not a "natural" constant (as in the discharge of a capacitor) and that the estimate should therefore not be projected too far in time or to a vastly different operating environment.

The above is a simplified estimation of the effect of error removal on software failure rate, and the cited references should be consulted for further detail. One area of simplification is due to our using operating time as the independent variable while the references predominantly use calendar time. Historically, failure data were only available in calendar sequence but current software support systems make operating time easily available, and this parameter should be used since it is much more indicative of the failure exposure than calendar time.

The accuracy of estimates obtained by this software reliability growth model will probably be improved if it is applied separately to each of the data partitions of the preceding discussion. Specifically, the concepts of Eqs. (8a) and (17) can be combined to furnish a composite estimate that accounts for differences in data mix and the effects of error removal as software is transitioned from test to operation:

$$\hat{u} = \sum_j \hat{k}_j E_{0j} e^{-\hat{k}_j t_j} P(Z_j) \qquad (18)$$

To conclude this section we mention briefly a method for estimating the total error content of a program from the success ratio in finding seeded or tagged errors. Total error content per se is a measure of software quality rather than reliability, but it may have a fairly direct relationship to failure rate (e.g., as shown in Eq. (10)).

The estimation procedure (Ref. 11,12) rests on the assumption that the ratio of seeded errors (E_s) to total errors ($E + E_s$) in a software program is the same as the ratio of seeded errors found (C_s) to total errors found ($C + C_s$) at a given time in the debugging process.

Thus,

$$E_s/(E + E_s) = C_s/(C + C_s) \qquad (19)$$

The unknown is the number of non-seeded errors, E, and this can be estimated by

$$\hat{E} = CE_s/C_s \qquad (20)$$

The accuracy of the estimate depends of course on seeded errors and naturally occurring ones being equally likely to be found. In most practical circumstances this cannot be assured a priori, and it will therefore be a major area of concern until the program is operationally proven, i.e., until better estimates of reliability are available. The equivalence (in likelihood of being found) of seeded and natural errors can be increased if the seeded errors are taken from the population of errors that were in the program to start. This can be done by having two independent test or debugging facilities, one of which furnishes "tagged" errors (equivalent to E_s) while the other one furnishes "total" errors (equivalent to $E + E_s$). Obviously, the debugging cost will be increased, but this may be offset by a more error-free program at the end of the process.

SOFTWARE RELIABILITY PREDICTION

The aim of reliability prediction in general is to make meaningful statements about the expected failure frequency of a device based on construction features and usage. This technique is widely practiced for predicting the reliability of electronic equipment based on parts population, individual parts stress factors, and overall equipment application factors (Ref. 13). These predictions are used to control equipment design (e.g., in limiting parts count or reducing the stress level on individual parts) and in application (e.g., in providing redundancy or in restricting the operating time of critical components). If similar predictive statements could be made with regard to software reliability they will obviously be valuable to the developer as well as to the user.

In trying to carry over hardware reliability prediction techniques into the software field one is of course confronted with the essential differences between the two areas. To the hardware reliability engineer, a computer is an assembly of semiconductor devices, capacitors, connectors, etc., all of which can be tested separately and for which failure rates and stress factors are published. The software engineer is confronted with the fact that (except in trivial cases) no two lines of code are alike, and, therefore, published failure data about elements of computer code will not be meaningful. Nevertheless, there is a feeling that the failure ratio must be affected by factors such as program size, complexity, and user environment. However, because of the inability to make meaningful tests on individual lines of code these relationships must be explored by regression analysis on existing programs that differ with regard to the independent variables that are to be explored.

A fairly extensive study of variables affecting software reliability was undertaken by TRW in support of an Air Force study of Command, Communication, and Information Processing Requirements in the 1980s (CCIP-85), and the results are discussed in Ref. 14. This study covered 88 software routines and considered 22 variables that might affect program reliability. As index of unreliability, the study used the total number of Software Problem Reports (SPRs) that were issued during the software test and operational phases. This is not a true reliability numeric(3), and a number of efforts are currently under way to establish an improved data base for software reliability prediction. The only statistically significant relationship identified in the CCIP-85 study was the dependence of number of SPRs on the number of instructions in the routine. The specific regression listed has the form

---

(3)Because it does not consider exposure to failure and thus furnishes none of the denominators used in Eqs. (1) or (2).

$$\text{SPR} = 2.14 + 0.00672 \times \text{Number of Instructions}$$

Other variables considered, including number of logical instructions, number of input/output instructions, number of interfaces, program difficulty rating, and several factors relating to programmers' experience all had a negligible effect on the number of SPRs written. Differences by program type were also investigated, and Ref. 14 concludes "there is no significant difference in the SPRs found as a function of routine type." However, routines that were classified as primary computational algorithms had only about eight SPRs per 1000 instructions while control routines had almost 15.

A number of investigators have published data on error density (the number of errors per thousand instructions) (Refs. 15,16,17). Many of these results cluster around 10 to 20 errors per 1000 instructions although a wider range is reported in Ref. 16. One of the limitations of this present data base is that very few of the authors identify over which phase of the program development these error totals are obtained. A recent theoretical study suggests a decided effect of program complexity (branches, loops) on error content (Ref. 18). This is also borne out by a high correlation of SPRs (that resulted in code change) with branching found in a recent analysis of a software data base (Ref. 19). The regression established there is

$$\text{SPR} = 0.060 \times \text{Number of Branches}$$

with a correlation coefficient of 0.98.

It may also be possible to predict error content from the scope of decisions and number of decisions in a computer program. The scope of decisions for an individual statement is determined by the number of operators and operands accessible to the programmer at that time which Funami and Halstead (Ref. 20) term the "vocabulary". The number of decisions is determined by the program length. The reference shows excellent agreement between computed and observed errors in post-facto analysis.

SOFTWARE FAILURE CLASSIFICATIONS

To permit useful inferences to be drawn from software reliability data it is required that the numerical reliability indices discussed in the preceding sections be supplemented by a methodology for failure classification. At least three descriptors are held to be necessary for meaningful interpretation of software failure data: time in the software life cycle during which failure occurred; manifestation of failure; and cause of failure.

Classification by time of failure occurrence should consider at least four categories: Initial debug; test and integration by developer; postdevelopment test; and operation. Each of these life cycle stages does not only have a characteristic level of failure incidence (in general, decreasing in the order listed here) but also the manifestations and causes of failures may conceivably be quite different. Merging of failure data, therefore, may obscure significant cause-and-effect relationships.

In terms of manifestation of failure, suggested classifications are: Abort of software system; abort of application program; persistent gross output errors; temporary gross output errors; inaccurate output; and other manifestations. From such a classification the developer and user can construct a scale of failure consequences that is particularly applicable to their environment. In a batch process environment the consequences of an application program abort and an inaccurate output may be almost the same, requiring rerun after correction of the program. On the other hand, in a real-time control system the consequences of an abort may be vastly more serious than those of a temporary incorrect output. By emphasizing the more universal classifications rather than a user-specific severity scale, we hope to facilitate interchange and merging of software reliability data from various sources, a step that is essential in order to further our understanding of the manifestations of software failures. In some environments the loss of computer system availability may be of more consequence than the event of failure. In such cases a severity index based on loss of computer time may be desirable. It cannot be directly constructed from the categories listed here because it is a function of manifestation of the failure as well as of the corrective effort that is available (manpower, backup programs, restart technique). However, even for these needs which are really somewhat apart from software reliability proper, the classifications proposed here may at least furnish some valuable insight.

Classification by cause of failure is desirable in order to organize remedial measures. This information is of value for the management of the immediate project on which it is obtained, for overall software management (e.g., in guiding the allocation of resources), and for the development of improved software engineering tools and procedures (language processors, test tools). With these users in mind at least the following categories should be established.

Specification errors

Conceptual errors in implementing the specification

Algorithmic errors (insufficient accuracy or neglect of singularities)

Logic and control errors

Exceedance of constraints (timing, memory, etc.)

Coding errors

Data structure errors

In the local environment and for specific attacks on the causes of software failures a more detailed classification of causes of software failure may be desirable (Ref. 19). It is believed, however, that for general reporting purposes the above categories will be found sufficiently comprehensive and that interchange of data among organizations and dissemination to the general software community will be facilitated by considering only a small number of categories.

The classification of software failures has been discussed here primarily with reference to reporting current or past events, i.e., in the context of software reliability measurement.

The use of the data, however, is primarily future oriented. On one hand, by virtue of the knowledge of the point in the life cycle at which failures can be expected and of knowledge of the immediate manifestation of the malfunction, the software development effort can be better organized and an acceptable product can be delivered in spite of the less than perfect performance of each line of code. On the other hand, knowledge of failure frequency and of causes of failure will permit improvement efforts to be concentrated on the functionally and economically most significant areas.

CONCLUSIONS

This, then, is the overall aim of software reliability measurement, estimation, and prediction: To permit better utilization of software capabilities that exist, and to help us guide the expenditure of limited resources for improvements where they are most needed.

The Aerospace Corporation
P. O. Box 92957
Los Angeles, CA 90009
15 November 1976

REFERENCES

1. Record, 1973 IEEE Symposium on Computer Software Reliability, IEEE Catalog No. 73CHO741-9CSR.
2. Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75CHO940-7CSR.
3. M. J. Merritt, et al., Characteristics of Software Quality, Report 25201-6001-RU-00, TRW Systems, Redondo Beach, CA (Dec 1973).
4. E. L. Welker and M. Lipow, "Estimating the Exponential Failure Rate from Data with No Failure Event," Proceedings 1974 Annual Reliability and Maintainability Symposium, IEEE Catalog No. 74CH0820-1RQC (January 1974).
5. E. C. Nelson, A Statistical Basis for Software Reliability Assessment, TRW-SS-73-03, TRW Systems Group, Redondo Beach, CA (1973).
6. J. R. Brown and M. Lipow, "Testing for Software Reliability," Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CHO940-7 CSR.
7. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CHO940-7 CSR.
8. J. C. Dickson, J. L. Hesse, A. C. Kientz, and M. L. Shooman, "Quantitative Analysis of Software Reliability,"

Proceedings, 1972 Annual Reliability and Maintainability Symposium, IEEE Catalog No. 72 CHO577-7 R, pp. 148-157.

9. Z. Jelinski and P. B. Moranda, "Software Reliability Research," Statistical Computer Performance Evaluation, (W. Freiberger, ed.) Academic Press (1972). Also available as MDAC Paper WD 1808, McDonnell Douglas Aircraft Corp., Huntington Beach, CA (1972).

10. M. L. Shooman and S. Natarajan, "Effect of Manpower Deployment and Error Generation on Software Reliability," Proceedings of the Symposium on Computer Software Engineering XXIV, MRI Symposia Series, Polytechnic Press, Brooklyn, NY (1976).

11. M. Lipow, Estimation of Software Package Residual Errors, TRW-SS-72-09, TRW Systems Group, Redondo Beach, CA (Nov 1972).

12. Beulah Rudner, "Design of a Seeding/Tagging Reliability Test," in Summary of Technical Progress, Software Modeling Studies, RADC-TR-76-143, Rome Air Development Center (May 1976).

13. Dept. of Defense, Military Standardization Handbook--Reliability Prediction of Electronic Equipment, MIL-HDBK-217B (Sept 1974).

14. R. W. Wolverton and G. J. Schick, Assessment of Software Reliability, TRW-SS-72-04, TRW Systems Group, Redondo Beach,

CA (1972). (Identical with paper in Proc. of the 11th Annual Meeting of the German Operations Research Society, Hamburg, GERMANY, Sept 1972.)

15. D. Itoh and T. Izutani, "FADEBUG-I, a New Tool for Program Debugging," Record, 1973 IEEE Symposium on Computer Software Reliability, IEEE Catalog No. 73 CHO741-9 CSR.

16. R. J. Rubey, "Quantitative Aspects of Software Validation," Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CHO940-7 CSR.

17. A. Endres, "An Analysis of Errors and Their Causes in System Programs," Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75 CHO940-7 CSR.

18. T. F. Green and N. F. Schneidewind, "Program Structure Complexity and Error Characteristics," Proceedings of the Symposium on Computer Software Engineering XXIV, MRI Symposia Series, Polytechnic Press, Brooklyn, NY (1976).

19. T. A. Thayer, et al., Software Reliability Study, Final Report, Report No. 76-2266.1.9-5, TRW Systems, One Space Park, Redondo Beach, CA 90278 (March 1976).

20. Y. Funami and M. H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," Proceedings of the MRI Symposium on Computer Software Engineering, Polytechnic Institute of New York (April 1976).

21. M. L. Shooman, "Structural Models for Software Reliability Prediction," Proceedings of the Second International Conference on Software Engineering, IEEE CATALOG No. 76CH1125-4C, San Francisco, CA (October 1976), p. 268.