One or more of the Following Statements may affect this Document

General Disclaimer
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. **Title**: Measurement, Estimation, and Prediction of Software Reliability |
| 2. **Contractor Report** |
| 3. **Report Date**: January 1977 |
| 4. **Performing Organization**: The Aerospace Corporation, El Segundo, CA 90009 |
| 5. **National Aeronautics & Space Administration, Washington, DC 20546** |
| 6. **Author(s)**: Herbert Hecht |
| 7. Mr. Gerard E. Migneault served as Langley Technical Monitor. |
| 8. **NMS-14392** |
| 9. **Contract Report** |
| 10. **Security Classification**: Unclassified - Unlimited |
| 11. **Key Words (Suggested by Author)**: Software reliability, Reliability measurement, Reliability estimation, Reliability prediction, Software error modeling, Software failure classification. |
| 12. **Distribution Statement**: Unclassified |
| 13. **Price**: $4.00 |

**Abstract**: Quantitative indices of software reliability are required for project management and 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 for comprehensive software reliability evaluation. Failure classifications and other documentation for comprehensive software reliability evaluation are described.
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.

Portions 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.

ACKNOWLEDGMENTS
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 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 software 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 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. Data requirements and other documentation for comprehensive computational techniques for all procedures are discussed. Software and for hardware/software tradeoffs can be used for resource allocation to modules among the total. Prediction of reliability can be made early in the project. It can be based on size, complexity, or similar factors. Reliability prediction is a statement about the reliability during test whether an operational reliability goal can be met. A typical application for reliability estimation is to determine.
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 testing of computer languages, development tools, and test tools proceeds on the underlying assumption that software reliability is measurable, and test tools produce improvements in reliability. Measurement of these improvements is the essential requirement for evaluation of this research. Thus, while software reliability measurement by itself does not "solve" the software failure problem, it is an essential tool for the direction of management activities and for demonstrating the accomplishments of research. This report aims to introduce the reader to some of the essential concepts in the numerical evaluation of software reliability and to clarify some of the misunderstandings of research. It might be well to start here with a statement of what we mean by "reliable" software; it is software that is correct by test, and that meets the simple mathematical relations that have been found useful in the evaluation of software reliability and to introduce the reader to the essential concepts in the numerical demonstration of the accomplishments of research. Thus, while software reliability measurement by itself does not "solve" the software failure problem, it is an essential tool for the direction of management activities and for demonstrating the accomplishments of research. In addition, current research in testing of computer languages, development tools, and test tools proceeds on the underlying assumption that software reliability is measurable, and test tools 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 tool for the direction of management activities and for demonstrating the accomplishments of research.

It might be well to start here with a statement of what we mean by "reliable" software: It is software that is correct by test, and that meets the simple mathematical relations that have been found useful in the evaluation of software reliability and to introduce the reader to some of the essential concepts in the numerical demonstration of the accomplishments of research.
Estimation of software reliability is performed by taking
software configuration only.

The period of measurement, and the developed reliability numeric
generated, typically, the software will not be modified during
the period of measurement. These scores a single indicator of measured reliability as
or success by the qualitative criteria cited above, and from
segments of the operation are scored as failure
for reliability measurement the software is operated over a

For reliability measurement the software is operated over a period of time, segments of the operation are scored as failure

corresponding to the development of numerical indicators in the
requirements and will attempt to deal with both of these
as agnostic more loosely defined (and particularly impotent) evaluation software reliability against formally specified as well
evaluating from poorly drawn specifications. We see a need to
formulations tend to prevent recognition of reliability problems
measurement on "intended functions" but more restrictive
will be justifiably concerned about any attempt to base
the degradation indicated reader
extent that it can be expected to perform its intended functions
with an earlier statement "software possesses reliability to the
interacting with the environment. This concept is consistent
other requirements imposed by the user such as timing and

Relevant to measurement (as above) on an existing program and
quantitative statements about software reliability.

Discussion of software reliability is an important aspect of software engineering, and prediction, respectively. The final section discusses dealing with specific techniques of measurement, and estimation, respectively. This is followed by three sections discussing practical applications of the software reliability numerics. Practical applications exist.

Environmental requirements with those of a program for which based on comparison of program length, complexity and even less respectable methods (which, according to rumors, are sometimes utilized for that purpose), prediction is normally determined may permit predictions based on casting of die or measurement taken on that particular program, while the terce prediction of software reliability is any statement about environment for which the estimate is to be valid.

Relationship between the measurement environment and the operating environment. Estimation requires some quantifiable modifying the result to represent the reliability in a different operating environment.
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 be ready for operation in the intended environment. Since software reliability measurement results in a quantitative index of reliability for software in the intended operating environment, the most obvious application of software reliability measurement is to determine compliance with a reliability requirement imposed by contract or specification. Another use of software reliability measurement that may have been imposed by contract or specification is to determine compliance with a reliability requirement imposed by contract or specification. Therefore, the processes of measurement, estimation, and prediction occur in reverse sequence. However, any significant technical discussion of these subjects should start with measurement and proceed in the order listed because estimation of a quantity is best discussed after there is agreement on how the quantity is to be measured. And a prediction made without a clear understanding of the quantitative formulation of software reliability is 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.

Software reliability measurement requires a "full up" state of the life cycle may be worse than useless. Reliability in subsequent stages of the life cycle may be worse without a clear understanding of the quantitative formulation of software reliability. And a prediction made without an estimation of how the quantity is to be measured is worse than useless. Therefore, the processes of measurement, estimation, and prediction occur in reverse sequence. However, it is therefore seen that the processes of measurement, estimation, and prediction occur in reverse sequence. However, the introduction of software reliability measurement as it has been described in the text is the best discussed after there is agreement on how the quantity is to be measured.
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 be identified and to a lesser extent that these factors can be explained, software failures are not necessarily due to deterioration of the reliability that has taken place. Since software deterioration may also be understood to.

Further testing and correction (and in a future operating environment, reliability estimation may also be used to measure at a future time (assuming reliability growth due to a sample measurement will be interpreted in terms of a reliability goal expected for it can be achieved, for this purpose, measurements will be taken over a limited period of time and with a limited set of test cases, and the results of this measurement will be interpreted in terms of a reliability goal expected for it can be achieved, for this purpose, a typical application for reliability estimation is to determine during development of a computer program whether the reliability estimation and for reliability prediction, necessary to develop and substantiate methods for reliability measurement.

Important research application of reliability measurement is that the value of new programming or test methods, a particularly may also be undertaken in support of research, e.g., to determine improvement in the measured reliability. Reliability measurement may vary with time, it is therefore possible to see deterioration or no deterioration in an already installed program that no
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 is a number of cases to be considered: 1) computer program tests carried out as a part of the development cycle, and 2) reliability of a computer program based on length, complexity of control structure, and other general characteristic. Reliability prediction is a numerical statement about the reliability of a computer program tests carried out as a part of the development cycle, and 2) reliability of a computer program based on length, complexity of control structure, and other general characteristic.
software measurement, estimation, and prediction. 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 a specification which in some cases be made unacceptable output conform to the program specification. That the unacceptable output may conform to the given input parameters. Thus the score as failure any deviation from an output which the user finds acceptable for the given input parameters is being undertaken to select the best math package. Reliability with respect to deviations from a specification with regard to the software specification or reliability with regard to user requirements. Rather than to champion one or the selected basis be other, it is important to insist that the selected basis be the case it might be with regard to satisfaction of user requirements. Rather than to champion one or other with respect to deviations from a specification where in other cases it may be with regard to satisfaction of user requirements. Similarly, reliability estimation and reliability prediction may in some cases be made uncontrollable or may conform to the program specification. On the other hand, if reliability estimation is being undertaken to select the best math package, it would be rather unreasonable to expect anyone to undertake a contractual obligation with regard to ill-defined user expectations. It is quite obvious that any deviations from the specification can be compliance with a specific reliability requirement. It is quite software measurement is being undertaken in order to determine specification or with respect to user requirements.
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

\[ R = \frac{S}{N} \] (1)

where \( R \) is the number of failures (1). This general definition is the number of failures may be expressed as

\[ \frac{N}{P} = N \] (1a)

The unreliability or failure ratio may be expressed as

\[ \frac{U}{S} = N \] (1b)

Of successful trials to the total, or

software reliability obtained from this measurement is the ratio of predetermined total number of trials (N), the numerical index of software reliability measurement is the identification of successful trials (S) among a predetermined total number of trials (N), the numerical index of software reliability measurement.
Failure statistics is not describable.

In most cases quite different so that combining of their known. However, the failure mechanisms in the two environments are each other when processing speed and associated factors are the discrete and continuous process equations can be referred to. The equation will be more applicable, with it here denoting CPU time. Requirements are subjected to an interactive system the second requirement where diverse data, involving different processing processes data sets and in denoting the total number of data sets by Eq. (1) may be applicable with it denoting the success rate.

By essentiality repeated data sets are input into such a program, where may be appropriate, depending on the application. Where interactively manner, either the discrete or the continuous indices for software executing in an environment (Ref. 4) for software executing in an essentially failure may be arbitrarily assigned in accordance with when no failure is observed in a predetermined time interval.

\[
\frac{n}{P/F} = \frac{t/e}{n} = \frac{t}{e}
\]

The reciprocal of this quantity is the failure rate interval to t.

Running time (t) divided by the number of failures (f) in the interval 0 to t.
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 be a very much shorter 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 given in Eq. (3):

\[ U' = \frac{F}{N \times L} \]
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) is simply instructions dimension of the denominator in Eq. (5) is simply instructions of time are used to express t and n, then the

\[ U' = \frac{F}{N \times L \times W} \]  

(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. This is intended, Eq. (3) can be modified to

\[ R'' = 1 - \frac{F}{N \times L \times W} \]  

R' is then given by
normalization for word length can also be incorporated for the continuous case. This yields
\[ u^* = \frac{F}{t \times n \times W} \] (6)
which has the dimensions of failure per bit processed and is related to the index established for the discrete case in
\[ n^* = \frac{F}{t \times n \times W} \] (7)
contiguous case. The yields normalization for word length can also be incorporated for the repetitive measurement of software reliability by this method is

"The finite computer word length, it is still in many cases so large that measurement of software reliability cannot be based on a ratio of correct program execution taken for that data set. Therefore, correct or incorrect output depends on the specific path of code segment with one set of data. Note, however, that executing a code segment (detected as non-branching sequences of statements) to the total number of segments, 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 cannot be based on a ratio of correct to the total of all possible input data sets correctly processed and 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)."
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 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 not be applied to the published data. In many other applications software reliability measurement. Normalizing factors may or may and thus does not represent an obstacle to implementation of both successful and unsuccessful runs may be required in any case. However, an approach that yields normalized data on compliance with reliability provisions (e.g., for determining
the same interval to yield a metric of total computer system
reliability. This can be combined with the expected number of
hardware failures for a specified time interval, obtainable from Eq.
(2a), reliability measures, for example, the expected number of
discussed here are in principle compatible with hardware
reliability measures. For these purposes it
is significant that the measures of software reliability
total computing system. For these purposes it
In many applications it is desired to express the
periodic monitoring of the reliability of computer programs.
for a given program. This procedure is particularly attractive for
instructions executed, a factor that can usually be estimated for
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. Therefore, the failure rate (Eq. (2a)) or the failure frequency (Eq. (2)) are expected too, and software will undergo changes that presumably will reduce the likelihood of environment in which software is expected to be used unmodified. In practice, of course, test cases are deliberately considered unbiased estimators of the reliability in the intended environment. If the test runs are completely representative of operational runs, if the software under test is expected to be used unmodified in the operational environment, then the reliability index obtained by use of Eqs. (1) or (2) can be used unmodified in the operational environment, then the operational runs are completely representative of representing a sample of operational runs.

Therefore interpreted as a sample measurement with the test runs, the actual software reliability measurement is a different time. The actual software reliability measured at a different environment or at a different environment so as to yield the probability of failure (or failure frequency) index is the reliability estimation index that is obtained from that for software reliability estimation data acquisition for software reliability estimation.
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, \( N_i \) runs were made that used data from subset \( Z_i \) and produced \( F_i \) failures, then the estimated unreliability for data from this data set is given by

\[
U^i = \frac{F_i}{N_i} \quad (7)
\]

The probability that failures due to data from \( Z_i \) will be observed in the operational environment will depend of course on the probability that data to data from \( Z_i \) will be encountered in the operational environment. This estimator is given by

\[
U = \sum_i E\left(\frac{F_i}{N_i} \cdot P(Z_i)\right) \quad (8)
\]

A procedure for removing bias due to test data severity has been proposed by Brown and Lipow (Ref. 6).
An equivalent estimator for continuous real-time programs can be formulated as

\[ u \in \mathcal{F}(t^*) \]

where \( u \) represents the time spent in processing data from

\[ F(Z^*) \]

The Brown and Lipow paper (Ref. 6) illustrates this technique during test ensures 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, \( P(Z_i) \), be accurately known. However, even under some mismatch of actual vs. estimated probabilities, the resulting reliability will still be an acceptable estimate. The authors also point out some

\[ \frac{\sum F(Z^*_j)}{F(Z^*)} = \sigma \]

where \( \sigma \) represents the time spent in processing data from

An equivalent estimator for continuous real-time programs can be
errors discovered during test will be corrected, causing failure
to the operational environment. In most situations, however,
software tested will be transmitted without change from the test
An implicit assumption in this technique is that the
certainty (Ref. 7).(2)
and amplitudes of the input data sets is described by Goodenough and
set will obviously increase. An approach for more detailed
determining probabilities of occurrence in the operational data
detailed input set properties, although difficulties of
uncertainties in this regard can be removed by considering more
implementation of the routine that examines input data sets.
equally likely failure probabilities for the actual
real numbers, very large or very small numbers all represent
Type determination program one must question whether integers,
to failure probability. Even in terms of the simple Triangle
talling within a given category are truly homogeneous with regard
established, the question then arises whether all test cases
obviously place limits on the number of categories that can be
have a reasonable number of test cases in each category,
result against uncertainties in the usage probabilities.

methods for selecting test cases that tend to desensitize the
methods for selecting test cases that tend to desensitize the
results.
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 restricted to a homogeneous input data population, i.e., they regard all test data submitted 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.

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

$$ u(t) = kE(t) \quad (9) $$

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

$$ u_0 = kE_0 \quad (10) $$

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

$$ u(t) = k(E_0 - C) \quad (11) $$

In the target environment (Refs. 8,9), techniques for modeling this reliability growth have been which the reliability estimate is to be valid. So far, the 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. The estimates based on Eq. (8) to be unrealistically high.
Test records are depended on to furnish data on $u_0$, $Ulf$ and $C$. Subtracting (Eq. 11) from (Eq. 10) we can then estimate $k$ as

$$k = \frac{(u_0 - u_f)/C}{E_0} \quad (12)$$

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

$$E_0 = \frac{1}{1 - C/(u_0 - u_f)} \quad (13)$$

There is some temptation to interpret $E_0$ as a test termination criterion (i.e., to test until indeed $E_0$ 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 $K$).

Instead, we would like to utilize these equations for estimation of parameters as very low failure rates are approached. Instead, we have already mentioned, it is very difficult to hold that all errors will be of the same type (in terms of constant $K$).

There is some temptation to interpret $E_0$ as a test termination criterion (i.e., to test until indeed $E_0$ errors have been found). This should be discouraged 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 $K$).

$$\frac{I_n - 0_n}{C} = E_0 \quad (13)$$

As $C$, substituting (Eq. 11) from (Eq. 10) we can then estimate $k$ and $C$. Further, by substituting the resulting value of $k$ in (10) we obtain

$$\frac{C}{I_n - 0_n} = k \quad (12)$$

Test records are depended on to furnish data on $u_0$, $u_f$.\]
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

\[ n(t) = \frac{E_0}{e^{-kt}} \]  

This can be combined with (9) to yield

\[ \frac{dE}{dt} = -kE(t) \]  

which has the solution

\[ E(t) = E_0 e^{-kt} \]  

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

The correction or error removal rate

\[ n(t) = \frac{E_0}{e^{-kt}} \]

By any other means (in making corrections or removing software failure results in removal of an error).
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 the use of calendar 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. Software support systems make operating time easier to use since it is more easily available 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 that have been removed as software is transitioned from test to operation: accounts for differences in data mix and the effects of error removal for differences in data mix and the effects of error removal.

Equations (8a) and (17) can be combined to furnish a composite estimate that accounts for differences in data mix and the effects of error removal for differences in data mix and the effects of error removal.

\[
U = \sum_{j} \left( \prod_{k} \frac{p_{0j} e^{-k^2 \sum_{i} f_{ij}}}{k^2} \right)
\]

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.
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, and it will therefore be a major area of concern until the program is operationally proven, i.e., until better estimates of the accuracy of the estimate depend of course on seeded errors

\[
\frac{E_s}{E + E_s} = \frac{C_s}{C + C_s}
\]  

(19)

Thus, the unknown is the number of non-seeded errors, \( E \), and this can be estimated by

\[
E = \frac{C E_s}{C_s (C + E)}
\]  

(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.

\[
\text{e.g., as shown in Eq. (10)}
\]
The debugging cost will be increased, but this may be offset by a more error-free program at the end of the process. Obviously, the debugging cost will be increased, but this may be offset by a more error-free program at the end of the process.

In the program to start, these 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 $E + E_s$).
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 redundant equipment or in restricting the operating time of critical components). If similar predictive statements could be made with regard to software reliability, they would 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 statements of software, therefore, (except in trivial cases) no two lines of code are alike, and therefore, published failure data about elements of the two areas. To the software reliability engineer they will obviously be valuable to the software developer as well as to the user.
Because it does not consider exposure to failure and thus 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. An 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 index of unreliability, and the results are not a true measure of software reliability. 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 equation, $SPRs \approx \text{number of instructions}$, was the dependent variable, and the study used the total number of software problem reports (SPRs) for software routines that were considered in the study. 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. However, because it does not consider exposure to failure and thus 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.
SPR = 2.14 + 0.00672 x 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 program complexity. A recent theoretical study suggests a decided effect of program complexity on branch count (Refs. 13, 14). Thus, it is also borne out by a high correlation of SPRs (that resulted in code changes) on error counts (Ref. 18). This is also borne out by a high correlation of SPRs (that resulted in code changes) on error counts (Ref. 18).

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, whereas control routines had almost 15. (Ref. 15, 16, 17).

Many of the results reported in the present data base are that very few of the authors identify over which phase of the program development these error totals were obtained. A recent analysis of a software data base (Ref. 18) offers that error density (the number of errors per thousand instructions) is a number of investigators have published data on error density. However, routines that were classified as primary computational algorithms had only about eight SPRs per 1000 instructions, whereas control routines had almost 15. (Ref. 15, 16, 17). A recent analysis of a software data base (Ref. 18) offers that error density (the number of errors per thousand instructions) is a number of investigators have published data on error density. However, routines that were classified as primary computational algorithms had only about eight SPRs per 1000 instructions, whereas control routines had almost 15. (Ref. 15, 16, 17).

A recent analysis of a software data base (Ref. 18) offers that error density (the number of errors per thousand instructions) is a number of investigators have published data on error density. However, routines that were classified as primary computational algorithms had only about eight SPRs per 1000 instructions, whereas control routines had almost 15. (Ref. 15, 16, 17).

A recent analysis of a software data base (Ref. 18) offers that error density (the number of errors per thousand instructions) is a number of investigators have published data on error density. However, routines that were classified as primary computational algorithms had only about eight SPRs per 1000 instructions, whereas control routines had almost 15. (Ref. 15, 16, 17).
SFR - 0.060 x Number of Branches with a correlation coefficient of 0.98.

It may also be possible to predict error content from the program length. The reference shows excellent agreement between computed and observed errors in post-facto analyses. The scope of decisions is determined by the "vocabulary" of an individual statement as determined by the programmer at that time which Parnas 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.

\[ SFR = 0.060 \times \text{Number of Branches} \]
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 of occurrence of failure, type, and cause.

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; and other manifestations. Relationships may obscure significant cause-and-effect concordance if data are gathered on failure data. Moreover, the manifestation of failure and causes of failure may here (but also after the manifestation of failure) have a characteristic level of occurrence. In general, decreasing in the order listed, failure incidence at each life cycle stage does not only have a characteristic level of occurrence, but also merging of test and integration data at least four categories: Initial debug; development test; postdevelopment test; and operation. Each of these categories by time of failure occurrence should consider the manifestation and cause of failure. Each life cycle during which failure occurred, manifestations of software failure data: time in the software development process. Moreover, it is necessary for meaningful methodology for failure classification. At least three indicators discussed in the preceding sections be supplemented by a reliability data that is required that the numerical reliability to be drawn from software
classification by cause of failure is desirable in order to

In order to further our understanding of the manifestations of
reliability data from various sources, a step that is essential
scale, we hope to facilitate interchange and merging of software
universality classifications rather than a user-specific severity
thesis of a temporary incorrect output. By emphasizing the more
the consequences of an abort may be vastly more serious than
output may be almost the same, requiring certain after correction
consequences of an application program abort and an inaccurate
their environment. In a batch process environment the
consequences of failure consequences that is particularly
applicable to

consequences of an application program abort and an inaccurate.

In a batch process environment the consequences of an application program abort and an inaccurate.
Owne, i.e., in the context of software reliability measurement.

The classification of software failures has been discussed
small number of categories.

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

data structure errors
Coding errors

Exceedance of constraints (timing, memory, etc.)

Logic and control errors

Specification errors
Logical and control errors (unpredictable accuracy or neglect of
Algorithmic errors (unpredictable accuracy or neglect of

Conceputal errors in implementing the specification

Specification errors
categories should be established.

With these users in mind at least the following
tools), with these users in mind at least the following
test
and for the development of improved software
resources), and for the development of improved software
overall software management (e.g., in guiding the allocation of
management of the immediate project on which it is obtained, for

The classification of software failures has been discussed
small number of categories.

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

data structure errors
Coding errors

Exceedance of constraints (timing, memory, etc.)

Logic and control errors

Specification errors
Logical and control errors (unpredictable accuracy or neglect of
Algorithmic errors (unpredictable accuracy or neglect of

Conceputal errors in implementing the specification

Specification errors
categories should be established.

With these users in mind at least the following
tools), with these users in mind at least the following
test
and for the development of improved software
resources), and for the development of improved software
overall software management (e.g., in guiding the allocation of
management of the immediate project on which it is obtained, for
The use of the data, however, is primarily future oriented. On the 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 performance of each line of code, on the other hand, knowledge of failure frequency and of causes of failure will permit product can be delivered in spite of the less than perfect development effort can be better organized and an acceptable improvement efforts to be concentrated on the functionally and economically most significant areas.
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

15 November 1976

Los Angeles, CA 90009

P. O. Box 9297

The Aerospace Corporation
REFERENCES

1. Record, 1973 IEEE Symposium on Commuter Software Reliability, IEEE Catalog No. 73CH0940-7 CSR.
2. Proceedings, 1975 International Conference on Reliable Software, IEEE Catalog No. 75CH0940-7 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 CH0940-7 CSR.

18. F. Green and N. P. Schneider, "Software Reliability, IEEE Catalog No. 75 CH0940-7 CSR.


