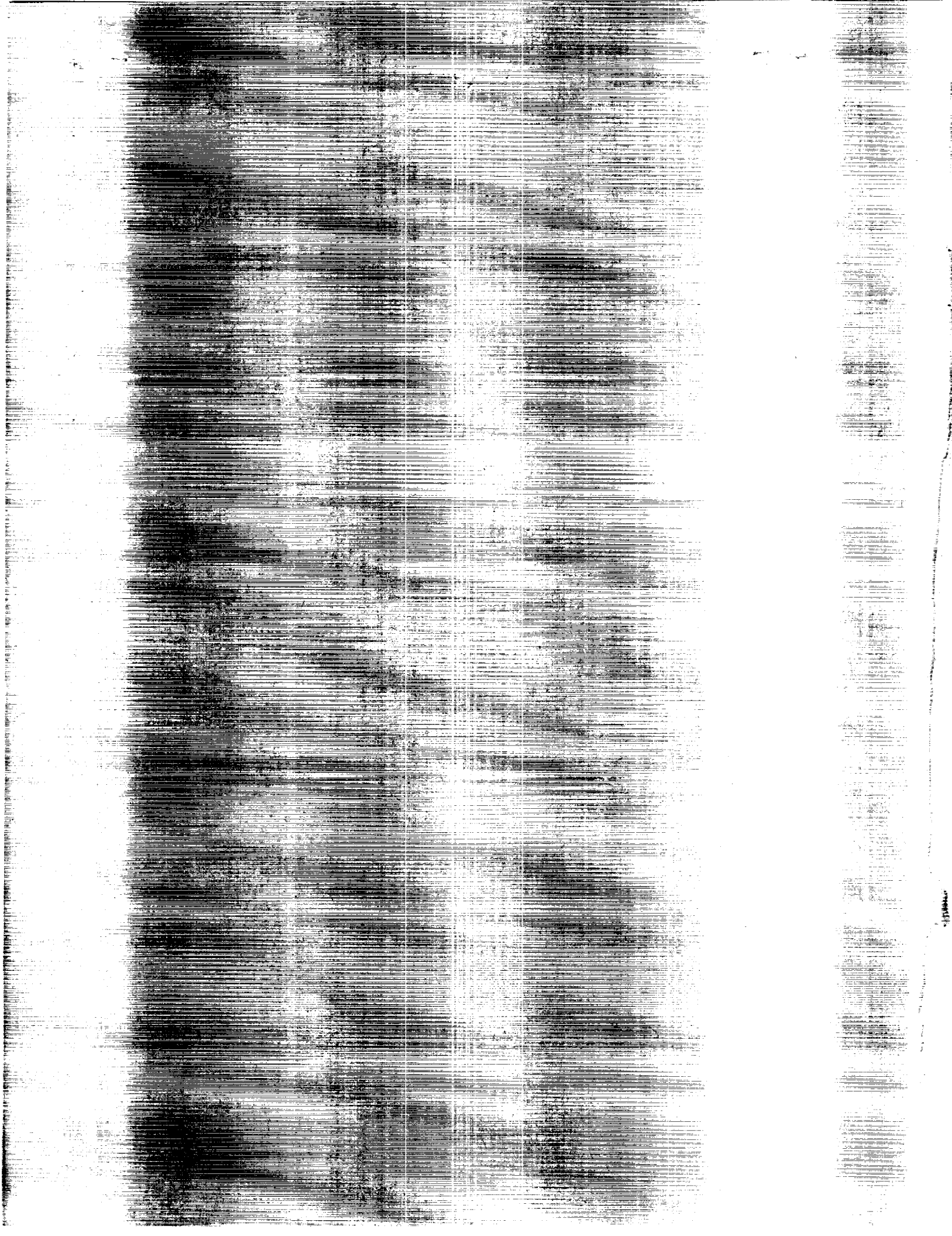
NASA Contractor Report 4197

Making Statistical Inferences About Software Reliability

Douglas R. Miller

GRANT NAG1-771
DECEMBER 1988



NASA Contractor Report 4197

Making Statistical Inferences About Software Reliability

Douglas R. Miller
School of Engineering and Applied Science
The George Washington University
Washington, D.C.

Prepared for
Langley Research Center
under Grant NAG1-771

National Aeronautics
and Space Administration

**Scientific and Technical
Information Division**

1988

TABLE OF CONTENTS

MAKING STATISTICAL INFERENCES ABOUT SOFTWARE RELIABILITY

by

Douglas R. Miller

1. Introduction

This paper presents some aspects of the statistical analysis of software reliability. The purpose is to point out some open problems and difficulties in the area, rather than to present solutions or new methods. This will be done first by considering moderate levels of reliability and a class of models (Exponential Order Statistic models) that appear to be useful. These moderate levels of reliability are characterized by the feasibility of testing for time durations that are at least an order of magnitude longer than desired mean times between failures. The second and primary concern of this paper is ultrahigh reliability, such as that occurring when failure rates are below one in one billion; for example, when failures occur at the rate of one per one billion missions, or one per one billion instances of using a program, or once per one billion units of operating time. In this case it is usually infeasible to test for a time duration even approaching the desired mean time between failures. Furthermore, it may be impossible to develop testing procedures which are sufficiently representative of field usage. It is highly questionable whether the statistical analyses useful at moderate levels of reliability are of any use here. Indeed,

it is unclear what role statistical methods can play in the verification of such ultrareliable software.

When a piece of software is put in service to perform some function, it may or may not perform as desired. If it has been carefully and skillfully developed, it will probably work as desired most of the time; however, it may occasionally not perform as required. A priori, there is uncertainty about how well it will perform. One approach is to model this uncertainty using probability theory and use statistical methods to make inferences about it. We would like to be able to translate everything we know about the software into estimates of how the software performs in the field. There is actually a vast amount of information available and fairly detailed aspects of performance which are of interest. For purposes of statistical reliability analysis, most information is ignored and the performance measure is often reduced to a failure rate. The statistical problem becomes one of estimating future failure rate from observed failures (which could have occurred during development, debugging, testing, or field usage). This is a highly simplified model of the situation, but it is still statistically rich and useful for situations in which sufficient data, i.e., failure data, can be collected.

2. A Model for Software Failure

Let us consider a model of software bugs and usage that will lead to observed failure times that are order statistics of independent, nonidentically distributed exponential random variables. We assume that the software operates by receiving data from some input space and transforming them into output data, which are either correct or

incorrect. This is done for a sequence of inputs selected randomly and independently from the space of all possible inputs according to some distribution over the input space. We assume that the internal state of the computer is identical for each input. If a subset, F, of the input space corresponds to inputs for which a certain bug in the software causes an incorrect output and the input distribution gives probability p to F, then in the above scenario this bug will have geometrically distributed interfailure times with mean 1/p. If p is small the interfailure time distribution can be approximated by an exponential distribution. If a bug is removed (perfect fix) when it manifests itself in an incorrect output, then we see a single exponentially distributed waiting time until manifestation. We further assume that the probability of simultaneous occurrence of more than one bug on any input is negligible; this allows us to model the separate manifestation times as independent continuous random variables. To summmarize: the Exponential Order Statistic (EOS) model can be described as follows. Let

$$0 \leq T_1 \leq T_2 \leq T_3 \leq \ldots \leq T_j \leq \ldots$$

be random variables corresponding to manifestation times of bugs in a piece of software. Each bug has a failure rate associated with it; if the bugs are arbitrarily indexed, then λ_i is the failure rate of the ith bug. Let X_i equal the occurrence time of the ith bug:

$$P\{X_i > t\} = \exp(-\lambda_i t).$$

The random variables $\{X_i, i = 1,2,3,\ldots\}$ are independent and their order statistics can be denoted as $\{T_j, j = 1,2,3,\ldots\}$. See Miller [7] and Scholz [11] for more information about EOS models.

It is important to consider the assumptions made by the EOS

model:

(i) Successive inputs are independently drawn from a single usage distribution, i.e., independent and identically distributed (i.i.d.) inputs.

(ii) Internal machine state is identical for each input.

(iii) Bug fixes are perfect.

(iv) There is no bug interaction.

It is equally important to note the absence of any assumptions about the failure rates

$$\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_i, \ldots .$$

They can be any finite set of values, or any infinite set of values for which

$$\sum_{i=1}^{\infty} \lambda_i < \infty.$$

There are some properties and special cases of EOS models that are of interest. If the rates (λ_i's) are chosen randomly from some distribution or as realizations of some nonhomogeneous Poisson process (NHPP), we get the family of doubly stochastic EOS models. This family of models is of special interest because of its richness--it contains all failure occurrence processes which are nonhomogeneous Poisson processes with completely monotone intensity functions. Many of the well-known parametric families of reliability growth models can be obtained in this way. For example, the Musa-Okumoto [8] model is a doubly stochastic EOS model with λ's from an NHPP with intensity

$$m(\lambda) = \gamma\lambda^{-1}\exp(-\beta\lambda).$$

In a certain sense this is a noisy version of Nagel's [9,10] log-linear failure rate model, where

$$\lambda_i = \delta\alpha^i, \quad i = 1,2,3,\ldots .$$

Consequently, assuming a parametric model often may be tantamount to assuming a particular pattern or distribution of the bug failure rates; for example, the Jelinski-Moranda [5] model assumes they are all identical. When viewed in this light it seems hard to imagine that there could be one pattern (or a few) of failure rate distribution which always arises to the exclusion of all others. It would make the inference much easier if it were true. If it is not true, it can cause severe estimation errors when one is trying to predict reliability, especially in the ultrareliable case.

Nagel's log-linear model is thought by some to be a possible candidate for a special prevalent pattern; but that would require much more justification. One problem is that the pattern is not closed under superposition, which would seem a desirable property inasmuch as it represents the combining of two programs into one. Suppose a software system consists of two modules, A and B, each of which is executed once for each application of the program. If the log-linear model were a universal model, then it would apply to both modules separately and to the system as a whole. Let $\Lambda_A = \{\lambda_1^A, \lambda_2^A, \ldots\}$ and $\Lambda_B = \{\lambda_1^B, \lambda_2^B, \ldots\}$ be the failure rates per application of the bugs in module A and module B, respectively. If $\lambda_i^A = \delta_A(\alpha_A)^i$ and $\lambda_i^B = \delta_B(\alpha_B)^i$, $i = 1,2,3,\ldots$, then it can be shown that the failure rates of the bugs in the system, $\Lambda_S = \Lambda_A \cup \Lambda_B$, do not in general exhibit a log-linear pattern.

It is useful to consider inference within the context of these Exponential Order Statistic models because it makes certain difficulties very apparent. In a sense the EOS models provide one possible

"best-case situation." Because many of the parametric models are special cases of EOS models, the same difficulties almost certainly apply to them. Some of the problems are:

(i) In order to make inferences about how the software will perform in the field, we must know the input domain as well as the input distribution that will be encountered.

(ii) The i.i.d. input assumptions for successive executions might be appropriate to certain batch types of application programs, but not to other software such as systems software or real time control software.

(iii) These models gloss over the problems of identification of separate bugs, imperfect fixes, dependencies between bugs and other factors which tend to make for messy data.

(iv) The richness of the possible models and some undesirable properties of completely monotone functions make accurate prediction difficult. (See Figure 4 and the accompanying discussion.)

3. Reliability Prediction

The prediction problem is of special interest for software maintenance. An estimate of the number of new bugs that will appear during some future time interval can be useful in planning. Using the EOS paradigm we see that this is a difficult problem because two completely monotone intensities can agree quite closely over a finite interval and then diverge over a future interval. They may agree closely enough that it is impossible to say which one best fits the data. This is illustrated by a sequence of figures: Figure 1 shows

some failure data, the cumulative number of bugs manifested by time t, $0 \leq t \leq 100$. Figure 2 shows two possible models; the straight line is

$$M(t) = t/2$$

and the curved line is

$$M(t) = 47 \log(.01t + 1)/\log 2.$$

These can represent the expected number of occurrences in NHPPs; the first is actually a homogeneous Poisson process (HPP) and the second is a case of the Musa-Okumoto model. The HPP is a limiting case of the EOS paradigm; the M-O model is an actual case of a doubly stochastic EOS model. Thus if one believes the EOS paradigm, both of these models must be considered. In Figure 3 the data are superimposed on the two mean functions. The point of Figure 3 is that the noise in the data is of the same or greater magnitude as the difference between the two models. Both models would probably pass hypothesis tests as the true model. Figure 4 shows the models extrapolated into the future. The two models differ by a factor of approximately two in the expected number of events in the interval [100,200]. This sequence of figures conveys a feeling for the randomness and the possible imprecision that arise when the number of future bugs is predicted from the occurrence times of previous bugs. The data were generated on a hand calculator from the HPP with $M(t) = t/2$. It seems that there are definite limitations to the inferences that can be made in the absence of additional information that will restrict the family of admissible models.

Reliability growth models have been used successfully in cases with moderate levels of reliability. For example, Currit, Dyer, and Mills [1] successfully use such models for a system in which they cite failure rates very roughly in the neighborhood of one failure per 5000

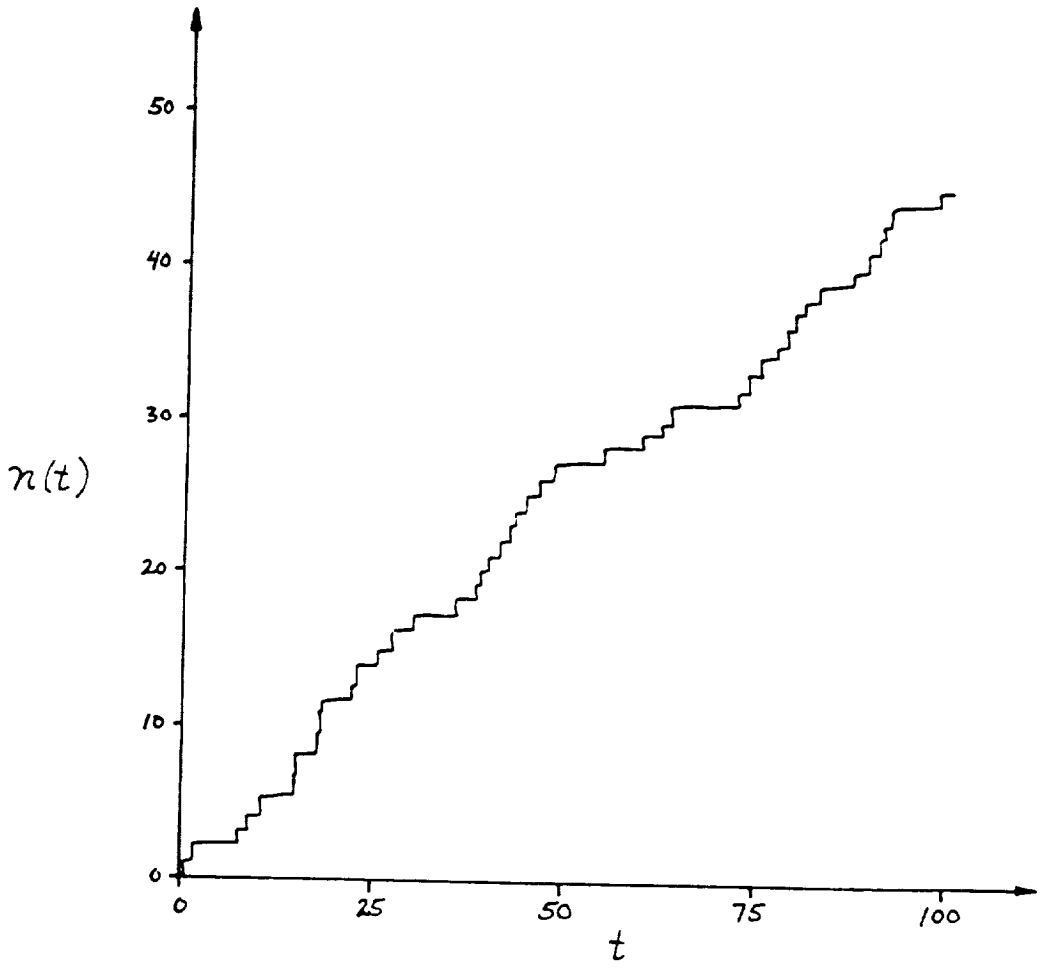


Figure 1. Cumulative failure data.

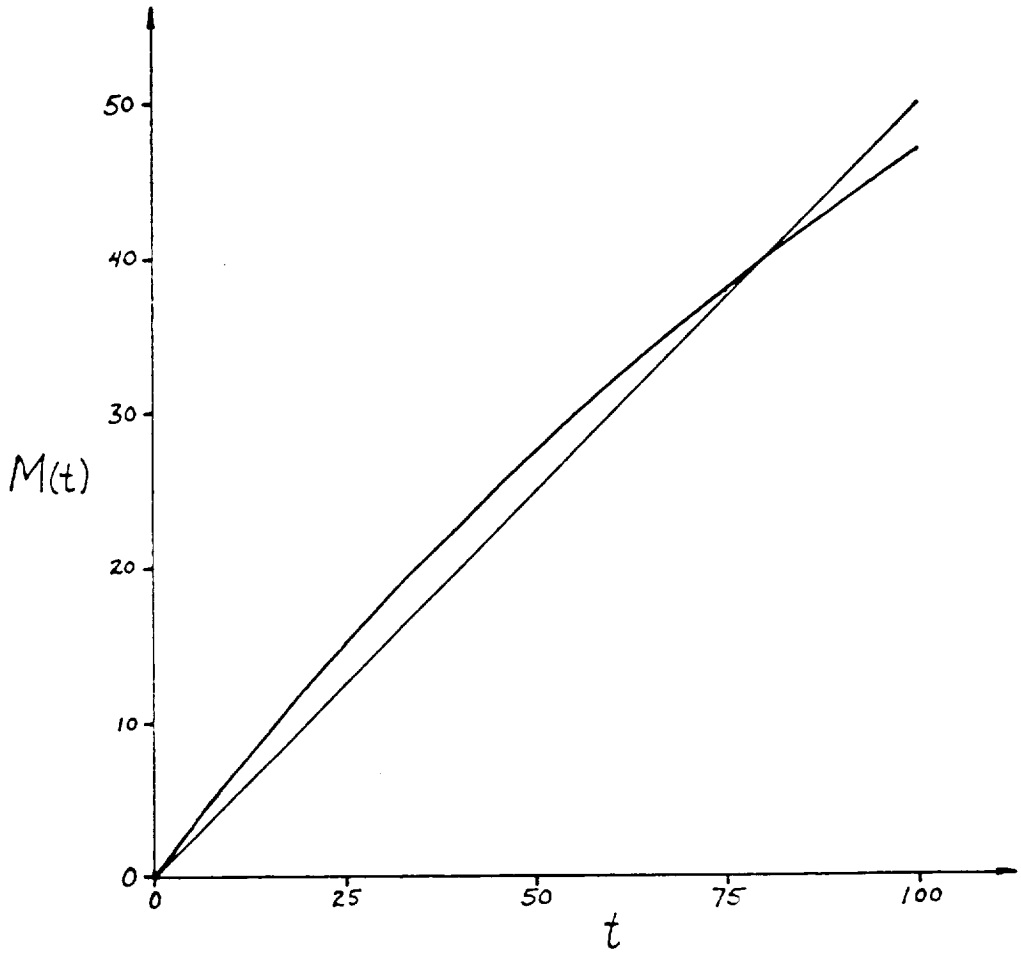


Figure 2. Two models with completely monotone intensities.

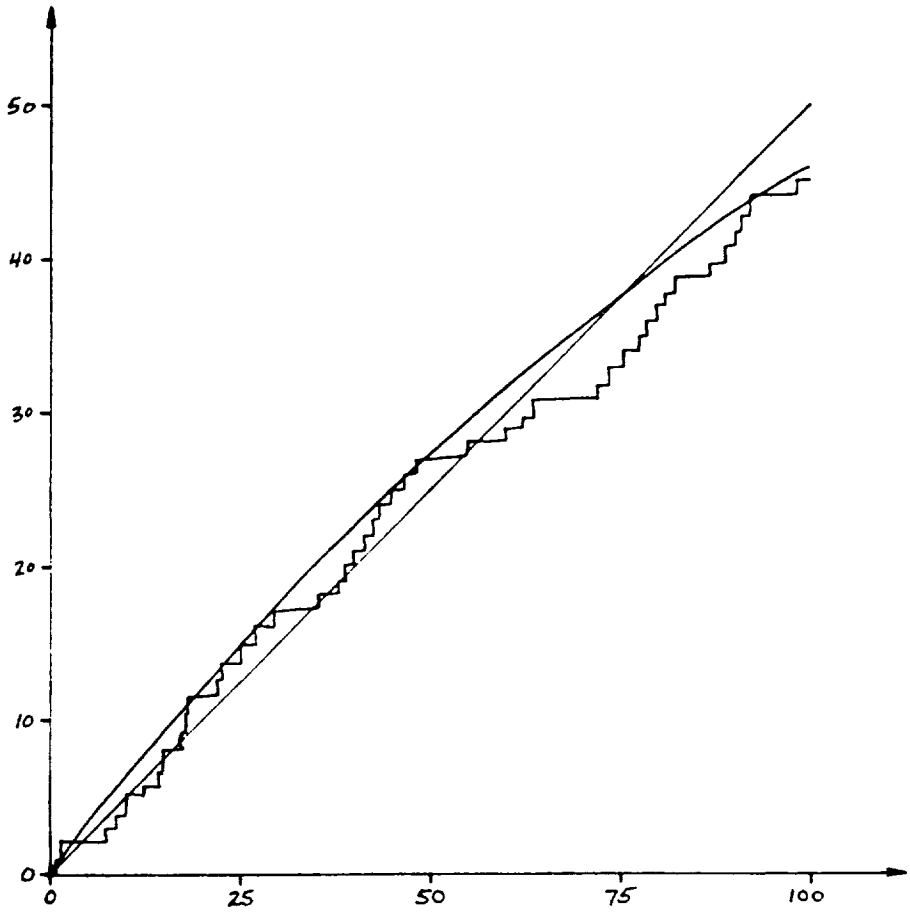


Figure 3. Superposition of models and data.

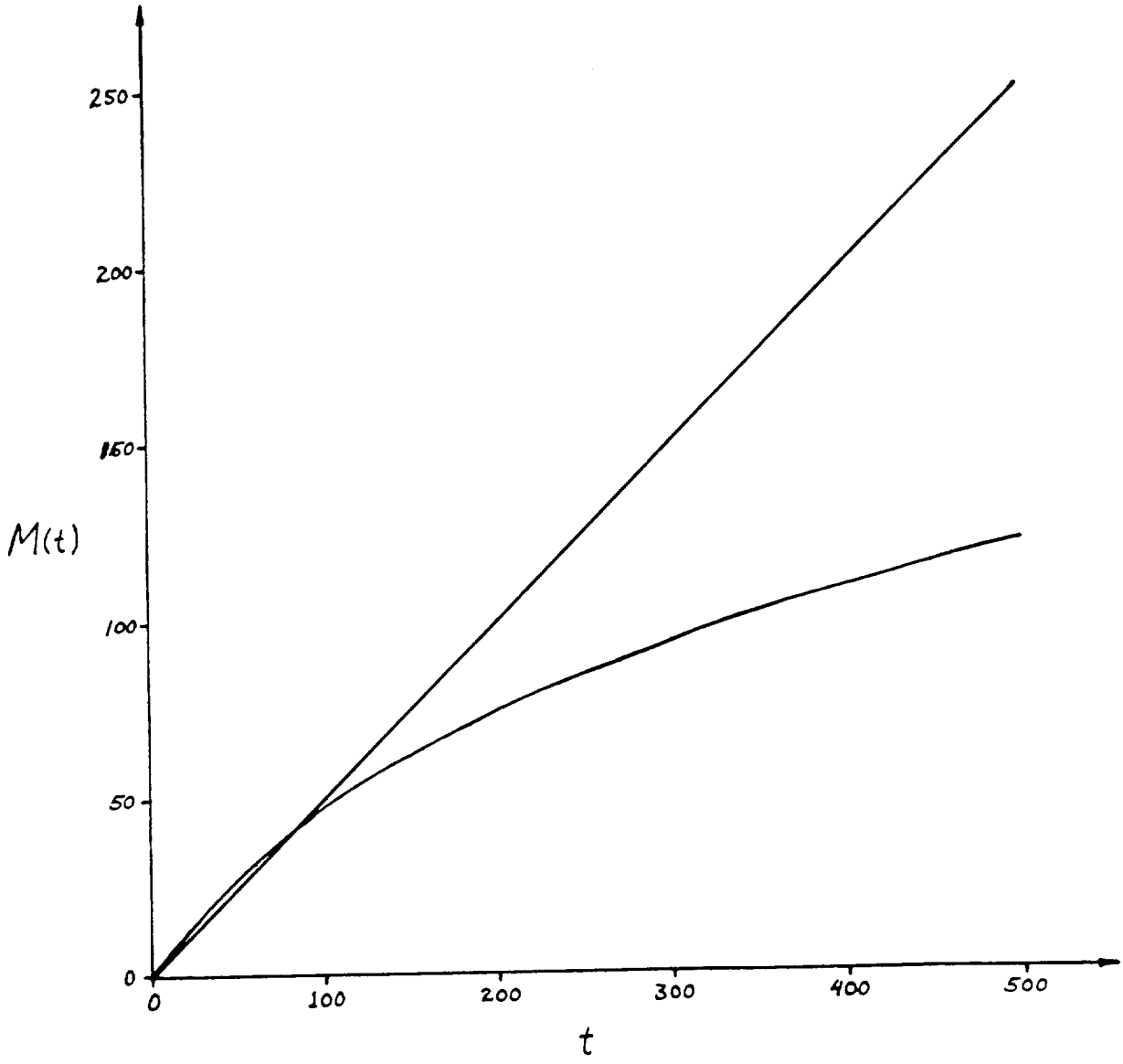


Figure 4. Extrapolation of models.

test cases. However, obtaining ultrahigh reliability through reliability growth requires very long testing. Bev Littlewood has observed the following interesting phenomenon, which the reader may check for himself: For the data used by Musa and Okumoto [8], the current reliability (as measured by the current time between failures) is at best in the neighborhood of 1/100 of the total accumulated test time and for higher reliability it could be less than 1/1000. These types of data, in which a fair number of failures is observed and the program is not ultrareliable, seems to be the appropriate domain for using statistical software reliability models. There may be value in getting estimates that are not extremely precise or in which we do not have an extremely high level of confidence. To this end probability models with incorrect assumptions can be used; the resulting approximations are acceptable and useful.

The general problem addressed above is to make inferences about future performance based on past performance of the software and any additional information. There are many open problems in modelling and inference with reliability growth data. Three main performance measures of interest are current reliability, expected number of bugs to be discovered over a future time interval, and additional debugging time to reach a desired level of reliability. In making inferences about these quantities, some open problems are:

(i) Determining accuracy of point estimates, perhaps by using confidence intervals.

(ii) Incorporating more information into inference procedures.

(iii) Finding ways to justify restricted classes of reliability growth models.

(iv) Correcting for uncertainty in the field usage distribution.

(v) Considering different sampling schemes, e.g., replicated debugging runs.

(vi) Quantifying the limitations of a purely statistical inference approach.

(vii) Handling imperfect fixes.

(viii) Distinguishing whether problems with accuracy are arising from bad models or from good models that have bad inference characteristics.

There is definitely a need for more sophisticated statistical techniques. They can be very useful as management tools. There seems to be opportunity for the mathematical statistical community to contribute more to this area. Correct and precise statistical analysis seems especially important for software which must be ultrareliable for safety reasons; but this may be an inherently different problem than the usual software reliability growth scenario, which addresses moderate levels of reliability.

4. Analysis of Ultrareliable Software

Software used in real-time control of safety-critical systems must be ultrareliable. The software in digital flight control computers aboard commercial aircraft will be critical to flight safety. Reliability on the order of 10^{-9} failures per hour or per mission is desired. Even trying to quantify such high reliability with statistical parameter values is very difficult and may be meaningless. The Radio Technical Commission for Aeronautics refrained from quantifying the reliability; in "Software Considerations in Airborne Systems and

Equipment Certification," DO-178A, they say:

> "During the preparation of this document, techniques for estimating the post-verification probabilities of software errors were examined. The objective was to develop numerical requirements for such probabilities for digital computer-based equipment and systems certification. The conclusion reached, however, was that currently available methods do not yield results in which confidence can be placed to the level required for this purpose. Accordingly, this document does not state post-verification software error requirements in these terms."

In Advisory Circular No. 25.1309-1 the FAA uses the value 10^{-9} to characterize "extremely improbable." Such events would be unlikely to occur during the entire lifetime of a fleet of aircraft. To see some rationale for this number, consider the following rough calculation: A single plane with a 30-year lifetime flies approximately 10^4 days, at most 10 hours per day. So a fleet of 10^3 planes would accumulate at most 10^8 flight hours. We would expect 0.10 occurrences of an event with 10^{-9} probability during this time. The Poisson distribution is a good model for such rare events; so the probability of no occurrences is $\exp(-0.10) = .90$, which is somewhat unlikely.

In theory, statistical verification of any level of reliability is possible. In practice, we encounter at least three major difficulties: the usage distribution used in testing may not perfectly fit the usage distribution encountered in the field; fixes may be imperfect; test time may be limited. The usage distribution is a severe problem: it may be desirable to bias the test distribution to go after bugs which the tester thinks are more likely to be in the software, but

perfect knowledge of the field usage distribution is needed to remove this bias in estimating the reliability. The problem of imperfect fixes can be avoided by considering the software to be completely new after each fix and thus not trying to base inferences about failure rate on previous versions. Furthermore it is unlikely that bugs would be allowed to remain in the program after they have been detected, so each version would be tested until it fails; then it becomes a new version. So an estimate of reliability would be based on failure-free tests. In order to have any degree of confidence that the failure rate is less than 10^{-9} failures/hour, it is necessary to test for more than 10^{9} hours and experience no failures.

Confidence intervals for failure probabilities based on error-free testing can be derived as follows. Let p denote the unknown probability of failure on a given randomly chosen test case. Suppose n test cases are run with no failure observed. In general, when n is large and p is small, the number of failures will be a random variable, X, with a Poisson distribution with mean $\mu = np$:

$$P(X = x) = e^{-\mu} \mu^{x}/x! \ , \quad x = 0,1,2,\ldots \ .$$

Thus we have observed data to which this model assigns probability

$$P(X = 0) = e^{-\mu} = e^{-np} .$$

The values of p for which these data are not statistically significant at level α are those satisfying

$$\alpha \leq P(X = 0) = e^{-np}$$

or

$$p \leq -\log\alpha/n \ ,$$

which constitutes a $100(1-\alpha)\%$ confidence interval for p, if n test cases

are run without failure. For various confidence levels the confidence intervals are:

Confidence Level	Confidence Interval
95%	$p < 3.00/n$
99%	$p < 4.61/n$
99.9%	$p < 6.91/n$
99.99%	$p < 9.21/n$
99.999%	$p < 11.51/n$

For example, to be 99% confident that the failure probability is less than 10^{-9} requires 4.6×10^{9} test cases without failure. If the unit of time is hours, this equals 525,000 years. (A more simplistic approach of 10^{9} failure free hours is still equal to 114,000 years of testing.) A side issue is that the software would have to be much more reliable than 10^{-9} in order to survive 4.6×10^{9} test cases without failing. Thus, infeasibly long testing times are necessary to verify high reliability and they probably would not work anyway because knowing the field input distribution precisely enough is a problem.

5. Bayesian Analysis

A Bayesian analysis of the reliabililty of software is attractive because it provides a way to incorporate more information into the inference. There is a lot of information available in addition to failure data during a software analysis. For example, if the software has been subjected to formal correctness-proving and has survived, this would improve perceived reliability while not guaranteeing perfection.

All this information could be incorporated into a prior distribution on the failure rate of the program. However, this only allows escape from huge samples when the software is acceptable a priori. The following example illustrates this: Suppose we assume the prior

$$P(\text{failure rate} = 0) = .99$$

$$P(\text{failure rate} = 10^{-6}) = .01.$$

The prior mean is 10^{-8} which is close to 10^{-9} but not quite acceptable. (Whether the mean is the appropriate measure is another question.) Previously we showed that we wanted an event to be unlikely during 10^{8} time units. Consider n test cases without a failure, then

n	$P(\text{f.r.} = 10^{-6} \mid \text{Data})$	$E(\text{events in } 10^{8})$	$P(0 \text{ events in } 10^{8})$
0	.010	1	.368
10^{5}	.00906	.9	.407
10^{6}	.00370	.37	.691
10^{7}	.00000046	.000046	.99995

We see that it is necessary to have a sample that is larger than the MTBF (10^{6} executions, in this case) of the bug which may only be present with .01 probability. This illustrates what appears to be a general property of the Bayesian approach: If the software is considered good enough a priori, no data are needed. If the software is good enough but the a priori feeling is that it is not quite good enough, then a large sample is needed. Such an extreme prior as given here also might not be viewed as credible.

In the above two analyses, a difficulty arises from the two-stage aspect of this problem. The first stage is whether a bug is present.

The second stage is whether the bug manifests itself. In the above Bayesian prior distribution we have two possibilities: a perfect program, or a flawed program which we can expect to fail 100 times during the lifetime of the design. If this second possibility occurs, the design will have to be modified, perhaps bankrupting the designer; this has probability .01. It is not clear that the mean of the prior plays a very significant role. A similar two stage feature occurs in the previous confidence interval approach: in this case it can be described in terms of "process" versus "product." The confidence level describes a property of the process of producing or testing software. To say we are 99% confident of having a good piece of software (one with failure rate 10^{-9}) sounds rather strange. So with these very high reliabilities, there seem to be some questions of interpretation of inference statements.

6. Analysis of Fault-Tolerant Software

There are two aspects of high reliability for safety-critical applications: achievement and verification. Fault tolerance has been used successfully to achieve very high levels of reliability; see the recent Proceedings of the International Symposium on Fault-Tolerant Computing [2]. One approach to software fault tolerance is n-version software. This allows for higher system design reliablity than that of individual components. Thus if higher software reliability is sought by using n-version programming, we should also be able to exploit this fact in making reliability verification. However, we encounter the problem that separate software components do not a priori fail independently (as can be assumed in hardware models). Eckhardt and Lee [3] have shown

theoretically why independence fails and Knight and Leveson [6] have shown experimentally that programs which are independently created will show dependencies in failing.

Consider a two-out-of-three software system in which three independently created versions of the software perform the same computational tasks and then use a flawless majority voter. Let F_S denote system failure and F_i denote failure of the ith component, i = 1,2,3; then

$$P(F_S) = P(F_1 \cap F_2) + P(F_1 \cap F_3) + P(F_2 \cap F_3) - 2P(F_1 \cap F_2 \cap F_3)$$

This may be manipulated into an equivalent expression:

$$\begin{aligned} P(F_S) = {} & P(F_1)P(F_2) + P(F_1)P(F_3) + P(F_2)P(F_3) \\ & - 2P(F_1)P(F_2)P(F_3) \\ & + (P(F_1 \cap F_2) - P(F_1)P(F_2)) + (P(F_1 \cap F_3) - P(F_1)P(F_3)) \\ & + (P(F_2 \cap F_3) - P(F_2)P(F_3)) \\ & - 2(P(F_1 \cap F_2 \cap F_3) - P(F_1)P(F_2)P(F_3)) \end{aligned}$$

In the second expression for $P(F_S)$ the last four terms are covariances which disappear if the three versions fail independently of one another. (The interpretation of these terms as "covariances" can be seen by considering the random variables X_1, X_2, and X_3, where $X_i = 1$ if component i fails and $X_i = 0$ otherwise. By definition, $Cov(X_1,X_2) = E(X_1X_2 - EX_1EX_2) = EX_1X_2 - EX_1EX_2 = P(X_1X_2 = 1) - P(X_1 = 1)P(X_2 = 1) = P(F_1 \cap F_2) - P(F_1)P(F_2)$.) If independence cannot be assumed these covariance terms must be estimated or bounded in some way. The difficulty is that in order for the three-version system to be a

significant improvement over the one-version system, the covariance terms must be very close to zero and this must be verified in order to verify the system reliability. So we must verify that

$$P(F_i \cap F_j) < 10^{-9}.$$

Hence we are back to the original problem which requires a huge sample, accurate knowledge of the usage domain and distribution, and so on. Any statistical approach that claims to support ultrareliability based on a moderate amount of data is almost certainly based on assumptions; verification of these assumptions would require a huge amount of data.

7. Conclusion

It is difficult to prove something is impossible, but the evidence suggests that a formal statistical verification of reliability will be impossible for various safety critical systems. The best software development and evaluation techniques will be used and huge amounts of documentation, as well as extensive testing, will be required for certification. But it is unlikely that this can be formed into a statistically rigorous verification. It is also fairly certain that such systems will be built. On the other hand, systems (such as bridges) have been built which turned out to be ultrareliable a posteriori. The initial verification seems more elusive than the achievement of ultrareliability.

REFERENCES

[1] CURRIT, P. A., M. DYER, and H. D. MILLS (1986). Certifying the reliablity of software. IEEE Transactions on Software Engineering, SE-12, 3-11.

[2] DIGEST OF PAPERS (1986). 16th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, Washington, D.C.

[3] ECKHARDT, D. E. and L. D. LEE (1985). A theoretical basis for the analysis of multiversion software subject to coincident errors. IEEE Transactions on Software Engineering, SE-11, 1511-1517.

[4] FEDERAL AVIATION ADMINISTRATION (1982). System design analysis. Advisory Circular AC-25.1309-1, U.S. Department of Transportation, September 7.

[5] JELINSKI, Z. and P. B. MORANDA (1972). Software reliability research. Statistical Computer Performance Evaluation (W. Freiberger, ed.), Academic Press, Inc., New York, 464-484.

[6] KNIGHT, J. C. and N. G. LEVESON (1986). An experimental evaluation of the assumption of independence in multiversion programming. IEEE Transactions on Software Engineering, SE-12, 96-109.

[7] MILLER, D. R. (1986). Exponential order statistic models of software reliablity growth. IEEE Transactions on Software Engineering, SE-12, 12-24.

[8] MUSA, J. D. and K. OKUMOTO (1984). A logarithmic Poisson execution time model for software reliablity measurement. Proceedings of the 7th International Conference on Software

Engineering, IEEE Computer Society Press, Washington, D.C., 230-238.

[9] NAGEL, P. M., F. W. SCHOLZ, and J. A. SKRIVAN (1984). Software reliability: additional investigations into modeling with replicated experiments. NASA CR-172378.

[10] NAGEL, P. M. and J. A. SKRIVAN (1982). Software reliablity: Repetitive run experimentation and modeling. NASA CR-165836.

[11] SCHOLZ, F.-W. (1986). Software reliability modeling and analysis. IEEE Transactions on Software Engineering, SE-12, 25-31.

[12] SPECIAL COMMITTEE 152 (1985). Software considerations in airborne systems and equipment certification. Radio Technical Commission for Aeronautics, DO-178A, Washington, D.C., March.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4197		

4. Title and Subtitle	5. Report Date
Making Statistical Inferences About Software Reliability	December 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Douglas R. Miller	
	10. Work Unit No. 505-66-21-03

9. Performing Organization Name and Address	11. Contract or Grant No.
The George Washington University School of Engineering and Applied Science Washington, DC 20052	NAG1-771

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	Contractor Report
	14. Sponsoring Agency Code

15. Supplementary Notes

NASA Langley Technical Monitor: George B. Finelli

Progress Report

16. Abstract

Failure times of software undergoing random debugging can be modelled as order statistics of independent but nonidentically distributed exponential random variables. Using this model inferences can be made about current reliability and, if debugging continues, future reliability. This model also shows the difficulty inherent in statistical verification of very highly reliable software such as that used by digital avionics in commercial aircraft.

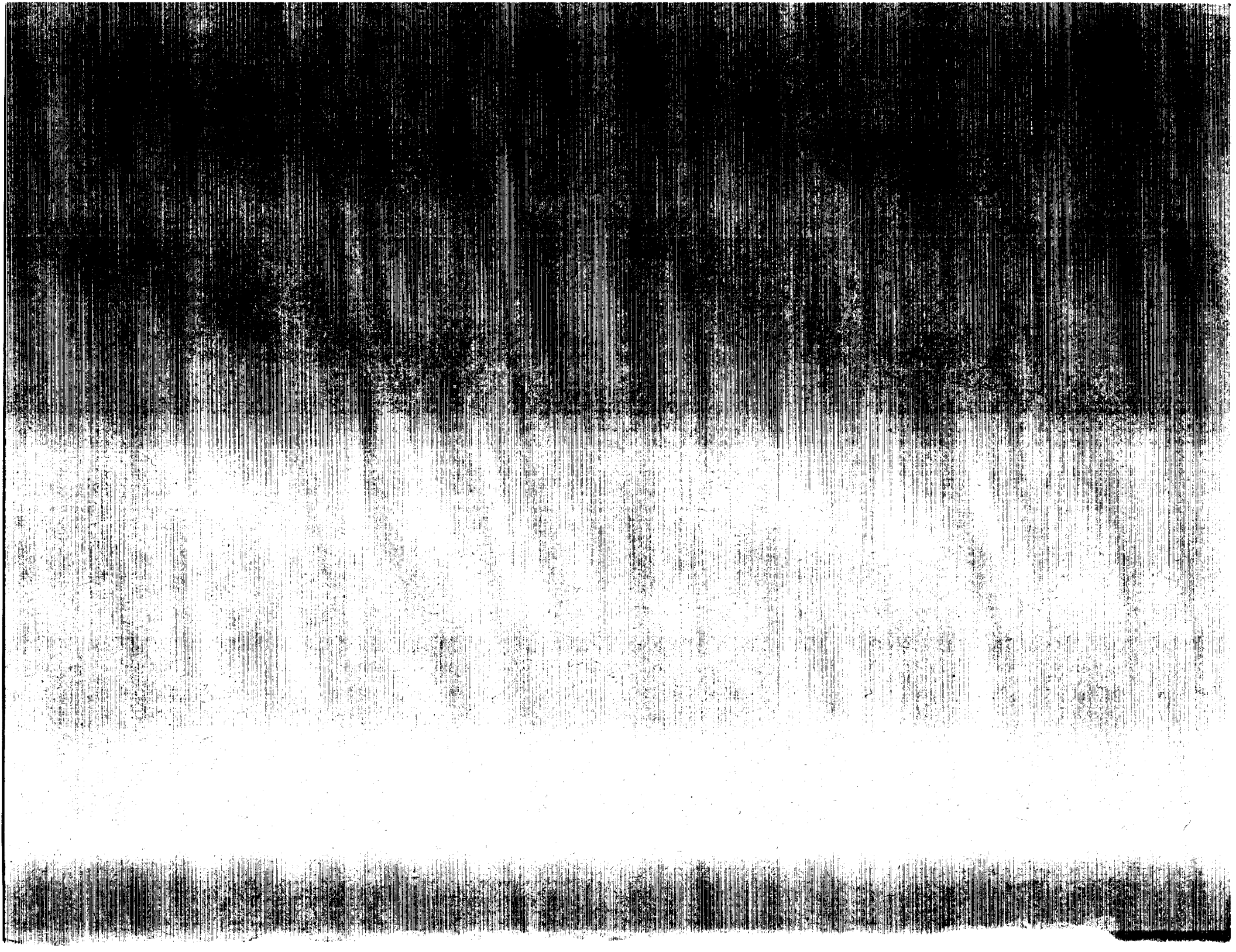
17. Key Words (Suggested by Author(s))	18. Distribution Statement
Software Reliability Reliability Growth Models Ultra-reliable Software Fault-Tolerant Software Software Quality Assurance	Unclassified—Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	28	A03

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171

NASA-Langley, 1988



BULK RATE
POSTAGE & FEES PAID
NASA
Permit No. G-27

POSTMASTER: If Undeliverable (Section 158 Postal Manual) Do Not Return