Software reliability analysis

by P. N. Misra

Methods proposed for software reliability prediction are reviewed. A case study is then presented of the analysis of failure data from a Space Shuttle software project to predict the number of failures likely during a mission, and the subsequent verification of these predictions.

Buyers of large, expensive software critical to the success of important missions have begun to see the need for the developers to provide a sort of "limited warranty" with regard to the probability that the software will perform as specified during a mission. Specification of such reliability is customarily required of developers of hardware systems. The failure-inducing mechanisms in the two cases, however, are entirely different. The failures of software depend entirely upon the number and kinds of errors (or design faults/bugs) in it and the probability that they will be encountered during the mission. The reliability improves as errors are discovered and corrected, and error-free software, by definition, is one hundred percent reliable. There is no counterpart to this situation in hardware systems. On the other hand, aging and degradation play no role in software failures. These differences notwithstanding, the concepts developed in reliability theory for hardware systems have provided a starting point for modeling software failures.[1] The approaches include probabilistic models that aim at predicting reliability and other elements of software quality on the basis of program properties such as size and complexity, and statistical models that base reliability prediction on an analysis of failure data.

This paper deals with the statistical approach to software reliability prediction. We consider the estimation of the number of software failures likely during a mission on the basis of data on errors discovered during the test and validation phase. Several interesting models for such an analysis have been proposed over the past ten years, and the literature on the subject has grown large. Success to date, however, has only been modest, and some basic issues remain open. We provide a brief introduction to these models. It is not our purpose to present a comprehensive discussion or a comparative analysis. Papers comparing the competing models have appeared in the literature,[2] but the attempts appear premature due to a paucity of data on which to base the tests. Indeed, one of the principal problems in this area has been lack of a data base of systematically gathered failure data from various projects.

Our main objective is to present failure data from a large software project, analyze these through a simple model, and compare its predictions with the behavior subsequently observed. The failure data are from software developed by IBM's Federal Systems Divison under contract to NASA's Johnson

Space Center for the Space Shuttle Ground Processing System. The project, we felt, came close to providing "clean" data for reliability analysis. The software and data collection process are briefly described in a later section, followed by a discussion of the failure data, model fitting, and reliability estimation.

Software reliability

In the past decade, several models have been proposed for predicting software reliability in an environment similar to that during which the past failure data were collected. This is the so-called representativeness-of-testing assumption. A commonly used conceptual model of software considers a program as a mapping from an input space to an output space. The inputs are chosen by a random mechanism, and a failure is observed if the input is from a certain subset of the possible inputs. Clearly, if software failure data from tests are to provide a basis for predicting the behavior during the operational phase, this input selection mechanism must be the same for both.

The objective of software reliability analysis is to predict future behavior of the software.

Typically, data are available from software tests as a sequence $t_1, t_2, \cdots, t_j$ of successive times between failures, or alternatively as samples $x(t_1), x(t_2), \cdots, x(t_k)$ of a failure-counting process $x(t)$, defined as the number of failures observed up to time t of software execution. The objective of software reliability analysis, then, is to predict future behavior of the software in terms of random variables $t(j+1), t(j+2), \cdots$, which are the future interfailure times, or, in the second case, in terms of $x(t_k + T)$, which specifies the number of failures in the next T time units.

The reliability of software in the next T time units, given the record of failures encountered in interval $[0, t]$ is defined as

$$R(t, T) = \text{Probability}\{\text{no software failures in } [t, t+T]\}$$

Actually, one is interested in more: If the mission will not be error-free, how many failures may be encountered? If too many, how much more time is needed for testing and error correction before the software attains the prescribed reliability? Insofar as software failures may differ in their impact, the above analysis may be carried out separately for subsets of failures classified on the basis of their severity.

A number of models[3-5] of the software failure process make essentially similar assumptions:

1. The instantaneous failure rate of software is proportional to the number of errors remaining in it, each of which is equally likely to cause the next failure.
2. The time separations between failures are statistically independent and distributed exponentially with different failure rates.

Following Musa,[5,6] we shall take time to mean software execution time. Denoting the initial fault content of the software by N, we see that the failure rate between the occurrence of the $(i-1)$th and ith failures is

$$L(i) = c(N - i + 1)$$

where c is a constant of proportionality. Here we have assumed that the errors are corrected immediately as they are discovered. The probability density function of the corresponding interfailure time is

$$f(t) = L(i) \exp[-L(i)t]$$

The resulting failure-counting process $x(t)$, defined earlier, has a mean value function

$$m(t) = N[1 - \exp(-ct)]$$

The distribution of $x(t)$, however, is not simple to characterize.

The assumptions of the above model are open to question. Variations of these have been proposed, but generally without a cogent conceptual or empirical basis.[2] Littlewood and Varrall[6,7] appear to bring greater realism by dropping assumption 1 given above, although at the expense of considerable

complexity. Of course, as Littlewood points out,[7] complexity in models, especially if they are to be implemented on computers, is not to be feared if it brings the models closer to the real world. This, however, remains to be established by more tests with "real" data.

According to one model, the number of software failures in an interval follows a Poisson distribution.

A slightly simpler model is one proposed by Goel and Okumoto,[8,9] who assume that the failure process is a nonhomogeneous Poisson process. This model replaces assumptions 1 and 2 above with those corresponding to the structure of a Poisson process. The interfailure times are no longer independent, and the instantaneous failure rate between failures varies with time. The Poisson structure brings about a simplification in the analysis of failure data given either as interfailure times or as samples of a failure-counting process. Goel and Okumoto take the form of the mean value function of the Poisson process to be the same as that for the model described above. Of course, N and c can no longer be interpreted exactly as before. We change notation and write the mean value function as

$$m(t) = a[1 - \exp(-bt)]$$

with a corresponding instantaneous failure rate

$$L(t) = m'(t) = ab \exp(-bt)$$

The failure process $x(t)$ has a Poisson distribution with expected value $m(t)$ and

$$\text{Prob}\{x(t) = n\} = [m(t)]^n \exp[-m(t)]/n!,$$
$$n = 0, 1, 2, \cdots$$

The number of software failures in any interval follows a Poisson distribution. Furthermore, the number of errors which would be found if the testing went on forever is Poisson-distributed with mean a. Parameter b is a constant of proportionality determining the rate at which the remaining errors are being discovered.

Some general comments on these models are in order. First, the mean value function of failure process in both cases is a monotonically increasing function with decreasing slope. This conforms to our experience: When a piece of software is tested, the errors are found relatively quickly at first, and as testing proceeds, the rate of error discovery gets slower and slower. The errors in paths executed under more frequently occurring conditions are found quickly, and those in paths traversed under infrequent and unusual conditions remain to be discovered. We therefore expect to be able to fit roughly such a function to software failure data. The question is whether the probabilistic structure of the model can "explain" the data and predict future failures. Second, our discussion has omitted several important practical features of software testing and debugging which may be accounted for in the models. We have assumed, for example, that the error responsible for an observed failure is corrected instantaneously and perfectly. Clearly, recurrence of a failure until the associated error is fixed may simply be disregarded, as we have done in our analysis. Actually, recurrences of failures due to known errors convey information on the failure process. There does not appear to be a simple way, however, to work this into the models. The problem of imperfect debugging is relatively easy to account for on an average basis with only a modest increase in model complexity.[6,9] Musa,[6] in his calendar time model of reliability, has attempted to incorporate features related to management of manpower and computer time allocation for the software testing and debugging process.

Given a record of observed failures during tests, fitting a model consists of an estimation of its parameters. For the nonhomogeneous Poisson process model used in our analysis, computation of maximum likelihood estimates of parameters a and b is discussed in Reference 9. Suppose failure data are available from software tests carried out over a period of length t, and that a and b are the estimates of the corresponding model parameters based on these data. Then, according to the model, software failures during a mission of length T will have a Poisson distribution with mean

$$L = a[\exp(-bt) - \exp\{-b(t + T)\}]$$

The model also gives estimates of the number of remaining errors and the additional time needed for testing to improve reliability to a prescribed value.[8,9]

Shuttle Ground System software

The Shuttle Ground System provides the flight controllers at the Johnson Space Center with processing support to exercise command and control over flight operations. Such responsibility requires the Ground System to verify each function performed by the computers aboard the Shuttle, and to carry out analyses that are beyond the capacity of those computers. The workload consists of processing high-speed telemetry data and push-button and terminal interactions with the flight controllers.

Both independent verification and mission simulations execute the software in a manner quite similar to that in the mission.

With over one-half million source lines of code, it is one of the largest real-time systems developed to date.

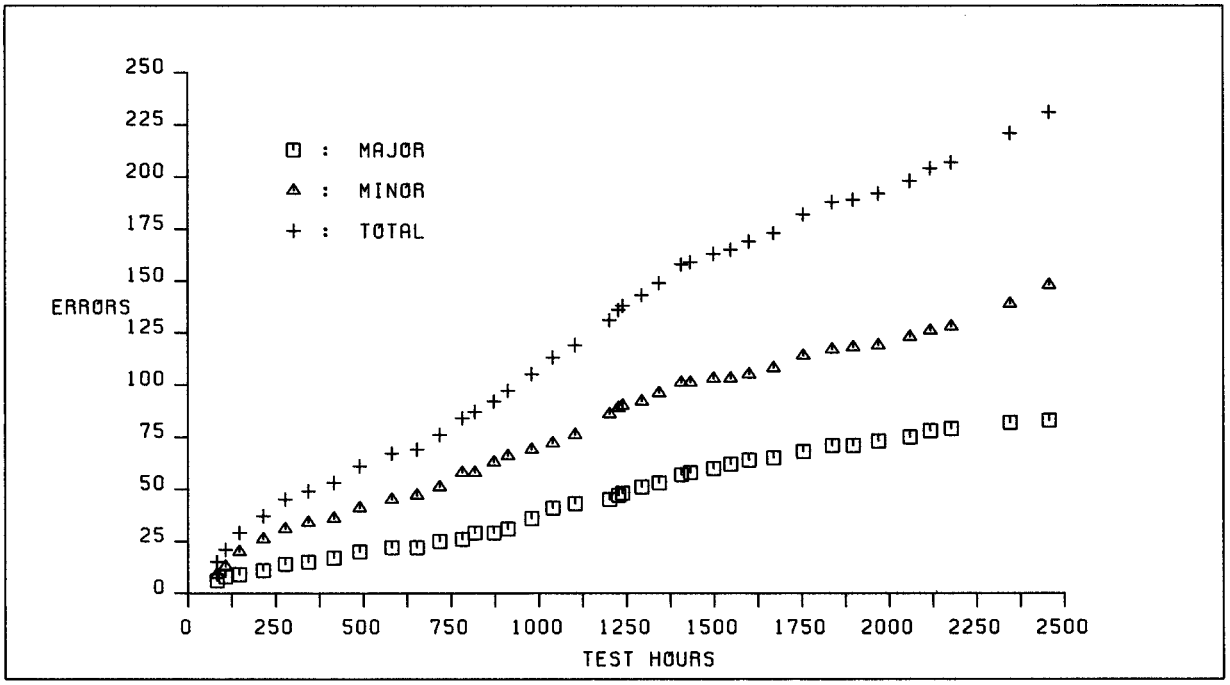
We have examined the pattern of discovery of errors in the software that supported Space Shuttle flights STS2, STS3, and STS4. Each successive release of this software was tested by an independent verification group and then delivered for operational use, which comprises mission simulations and the missions. We have taken data from a phase where the software had become stable. Release-to-release changes consisted of error fixing and minor modifications. Anomalous behavior of the software during independent verification, mission simulations, and the mission was documented in the so-called discrepancy reports. Each such report records the nature of anomalous behavior, the test session in which it was observed, the severity of the error in terms of its impact (critical, major, and minor, based on well-defined criteria), and other information relevant to error isolation. The discrepancy reports were examined by the appropriate development groups for validity, and errors, if any, were corrected in a subsequent release.

It should be noted that both independent verification and mission simulations execute the software in a manner that is quite similar to, though not identical with, that in the mission. Typically, verification is a scaled-down version of the mission planned to

Table 1 Software failure data: Weekly summary

Week	Test Hours	Critical Errors	Major Errors	Minor Errors
1	62.5	0	6	9
2	44.0	0	2	4
3	40.0	0	1	7
4	68.0	1	1	6
5	62.0	0	3	5
6	66.0	0	1	3
7	73.0	0	2	2
8	73.5	0	3	5
9	92.0	0	2	4
10	71.4	0	0	2
11	64.5	0	3	4
12	64.7	0	1	7
13	36.0	0	3	0
14	54.0	0	0	5
15	39.5	0	2	3
16	68.0	0	5	3
17	61.0	0	5	3
18	62.6	0	2	4
19	98.7	0	2	10
20	25.0	0	2	3
21	12.0	0	1	1
22	55.0	0	3	2
23	49.0	0	2	4
24	64.0	0	4	5
25	26.0	0	1	0
26	66.0	0	2	2
27	49.0	0	2	0
28	52.0	0	2	2
29	70.0	0	1	3
30	84.5	1	2	6
31	83.0	1	2	3
32	60.0	0	0	1
33	72.5	0	2	1
34	90.0	0	2	4
35	58.0	0	3	3
36	60.0	0	1	2
37	168.0	1	2	11
38	111.5	0	1	9

Figure 1 Software failures versus test hours



verify the error fixes and any new features in the release. It has no flight crew participation, and fewer telemetry streams and flight controller consoles are active than in a mission. The simulations are typically full-scale tests, but their main purpose is to train the flight crew and the controllers in the mission procedures and in various contingencies that may arise. A simulation session may concentrate entirely, for example, on repeatedly running various abort scenarios. Although the principal modules dealing with telemetry data processing, trajectory computations, and operating system services are still exercised, the range of system states and inputs may not be representative of the mission.

The project data bases maintained information on software test sessions and scenarios, discrepancy reports written during each, and dispositions of these reports. These data bases provide us with samples of the failure-counting process, one sample per test session. Note that data on software interfailure times, though richer in information content, require more of the data collection process and may be harder to come by in general.

A weekly summary of software test hours and the errors of various severities discovered is given in Table 1. The last two entries correspond to mission STS3. A plot of the cumulative test hours versus the number of errors found is given in Figure 1. The errors labeled as critical, being few in number, have been lumped together with the major errors. The flattening tendency is quite pronounced in the plot for the major errors, but less so for the minor errors. The plot for all errors is basically similar to that for the minor errors, the larger of its two constituents. The minor errors, which are occasionally referred to in the programmers' lingo as "nits," may not always be caught in their first occurrence; indeed, some may never be caught.

Model fitting and reliability analysis

The nonhomogeneous Poisson process model was fitted separately to errors classified as major and minor, and then to all errors. The last category, of course, is just the sum of the major and minor errors. This fact, however, is kept from the model. This creates a slight problem, for example, in inter-

preting the estimated values of parameter a in the three cases, but we will ignore it.

The maximum likelihood estimates of parameters a and b, computed from the failure data, are as follows:

	a	b
major errors	163.813	0.28759×10^{-3}
minor errors	315.551	0.25756×10^{-3}
all errors	597.887	0.20988×10^{-3}

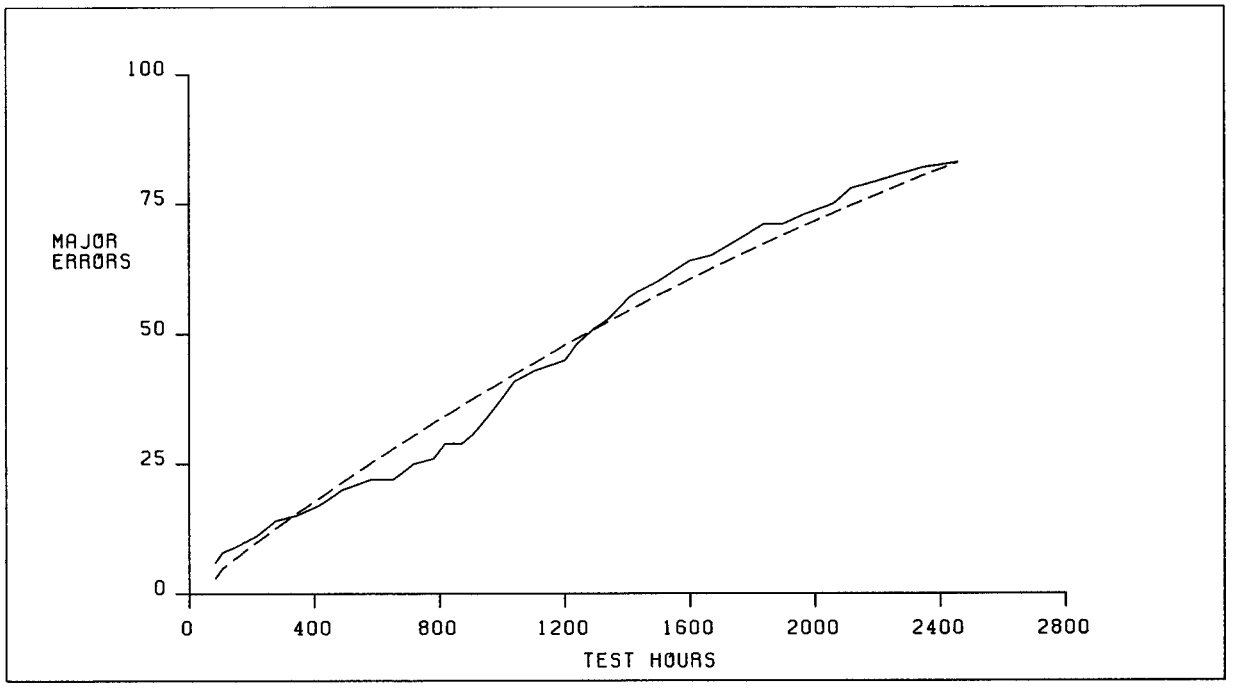
With these values of the parameters, the mean value functions have been superimposed on the error data in Figures 2 to 4. The fit in each case appears to be good, but, as noted earlier, it would be a mistake to read too much into it yet. The question is whether a nonhomogeneous Poisson process with a mean value function as fitted could reasonably have given rise to the observed realization. We do an easy test of computing 90th percentile upper and lower bounds of the Poisson process and find the observed realization in each case well within the bounds. Figure 5 gives such a plot for the major errors. This test suggests that the probabilistic structure of the software failure process is not inconsistent with that of the Poisson process.

Table 2 The predicted and actual software failures during STS4

	Software Failures		
	Major	Minor	Total
Predicted			
Median	3	7	13
90%	6	11	19
Actual	5	9	14

The main test, of course, is to compare the prediction of software failures for a future mission on the basis of this structure with the subsequent experience. We did just that for the STS4 mission. From our earlier discussion, these failures are predicted to have a Poisson distribution whose mean is specified by the corresponding estimates of parameters a and b, cumulative test hours t, and mission duration T. Table 2 gives the median and 90th percentile points

Figure 2 Actual and fitted major software failures versus test hours



of the predicted probability distributions corresponding to the major, minor, and all failures for the 200-hour mission. Also given are the numbers of software failures subsequently observed during the mission. The observed failures are consistent with the distribution functions predicted by the model.

Development of reliability models is important to both the customer and the developer of software.

We have also since done an after-the-fact analysis of what this model would have predicted for the STS2 and STS3 missions on the basis of error data available at each mission. In both cases, the numbers of observed failures were found consistent with the corresponding Poisson distributions predicted by the model.

Remarks

Our objective has been to present an analysis of software failure data from a project where the issue of reliability is vital. It is not our purpose to champion the Goel-Okumoto model. We chose it for its simplicity. The other models are not as tractable when failure data are given as samples of the failure-counting process. Yet, with appropriate simplifying assumptions, some of these models could have been "pushed through" and might have yielded reasonable results.

Development of models of software reliability is important for several reasons. The customer's need for specification of reliability has already been pointed out. There is a corresponding need on the

Figure 3 Actual and fitted minor software failures versus test hours

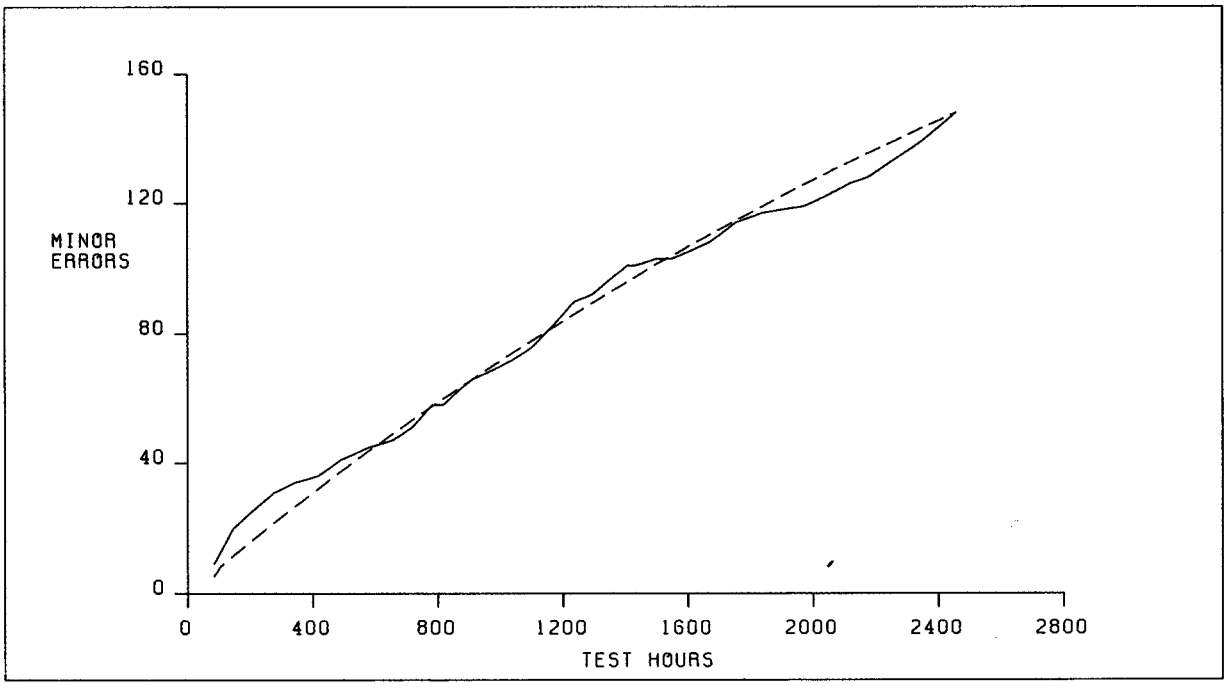
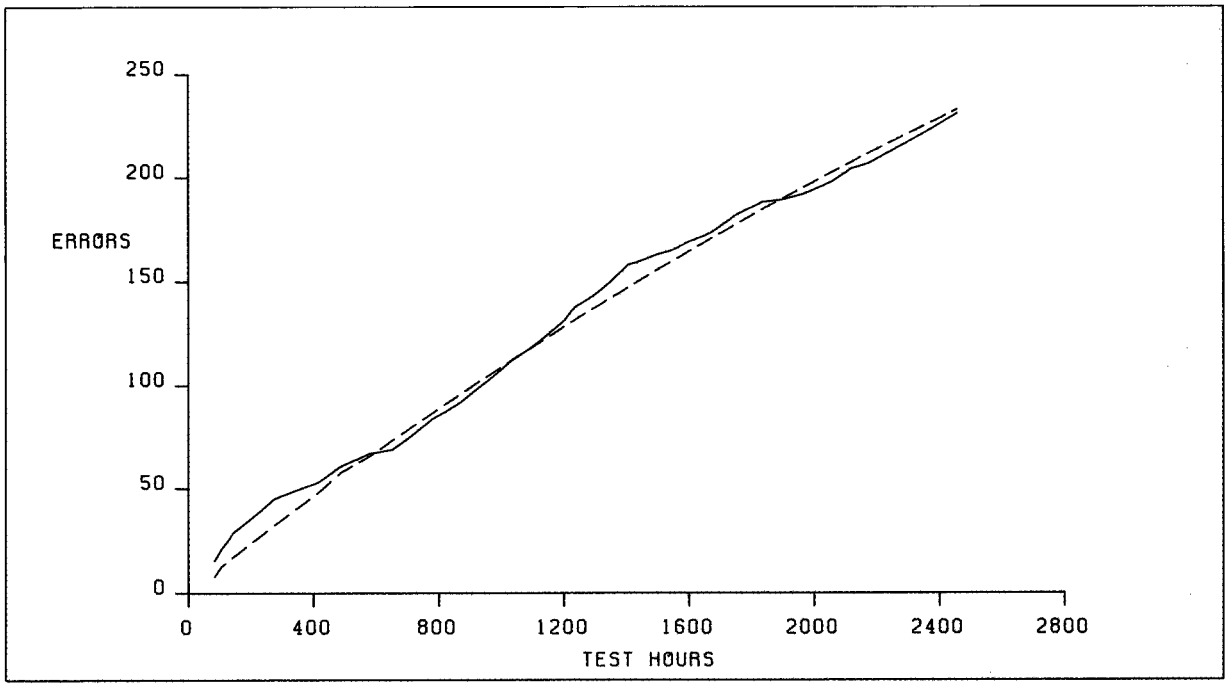


Figure 4 Actual and fitted software failures (total) versus test hours



part of project managers for the evaluation of methodologies of software development and testing, and for scheduling these activities to "build" the required reliability in the software. Again, the models of software reliability provide a basis for such decisions. As an example, note in Figures 2 to 4 that the instantaneous failure rate for our software declines by 20 to 25 percent for each one thousand hours of testing. Clearly, this is related to both the software and the techniques used in testing it. No conclusions on the efficacy of our testing can be drawn, however, due to a lack of comparable data from similar projects. The same is true of estimates of number of errors per, say, one thousand source lines of code. An empirical basis for these important indices is provided by an analysis of failure data based on reliability models. Much work, however, remains to be done in data collection and tests before these concepts can be proven to be of operational value in answering questions such as "How much testing is enough?", or "Is the software ready to fly a mission?"

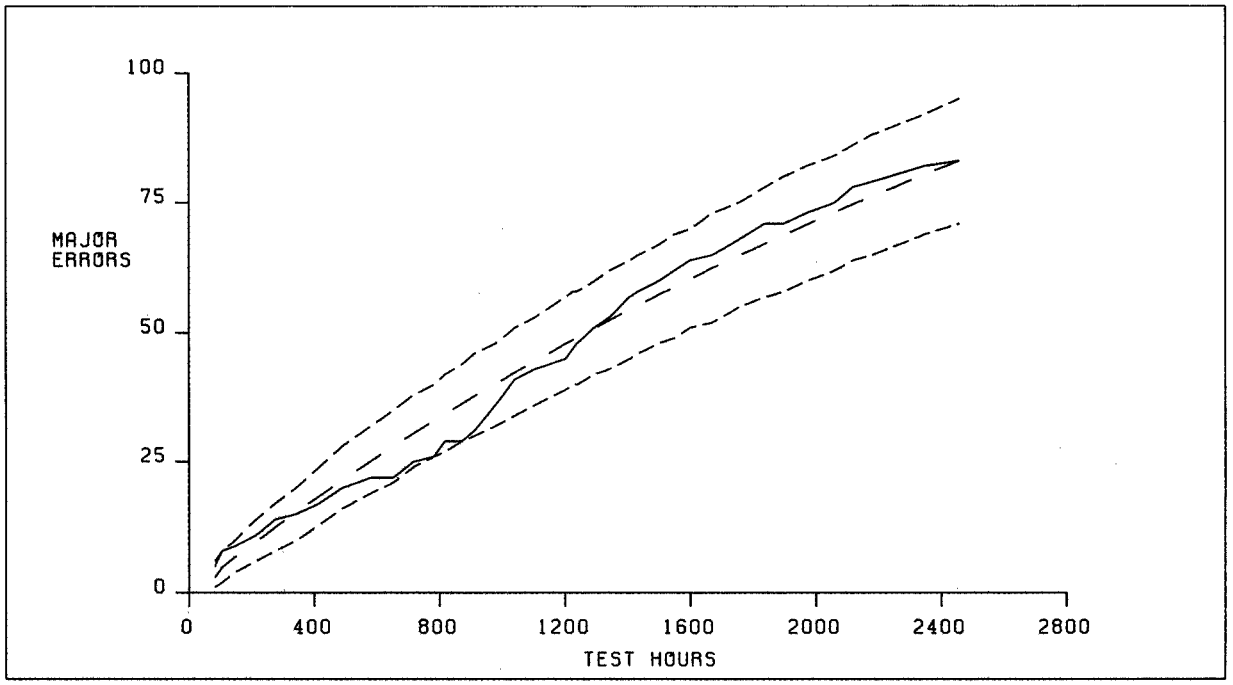
Summary

Software failure data from a project, gathered during tests, were analyzed through a model that postulates the failure process to be a nonhomogeneous Poisson process. The model was found to fit the data well, and its predictions of the number of failures likely during a subsequent mission have since been borne out. A considerable amount of work, however, remains to be done in model development and validation before these concepts can be proven to be of operational value.

Acknowledgments

The author is grateful to E. Clayton and G. Richeson of NASA/JSC, and to his former colleagues J. Capizola, A. Aldrich, C. Baldwin, and C. Greer of IBM Houston for their support. This work was performed for the NASA/Johnson Space Center under Contract NAS9-14350.

Figure 5 Actual and fitted major software failures and 90 percent confidence bounds derived from the model



Cited references

1. C. J. Dale and L. N. Harris, "Approach to software reliability prediction," *Proceedings of the 1982 Annual Reliability and Maintainability Symposium* (1982), pp. 167–175.
2. A. L. Goel, "A summary of discussion on 'An analysis of competing software reliability models,' " *IEEE Transactions on Software Engineering* **SE-6,** No. 5, 501–502 (1980).
3. J. Jelinski and P. B. Moranda, "Software reliability research," in *Statistical Performance Evaluation,* W. Freiberger, Editor, Academic Press, Inc., New York (1972), pp. 465–484.
4. M. L. Shooman, "Probabilistic models for software reliability prediction," in *Statistical Performance Evaluation,* W. Freiberger, Editor, Academic Press, Inc., New York (1972), pp. 485–502.
5. J. D. Musa, "A theory of software reliability prediction and its applications," *IEEE Transactions on Software Engineering* **SE-1,** No. 3, 312–327 (1975).
6. J. D. Musa, "The measurement and management of software reliability," *Proceedings of the IEEE* **68,** No. 9, 1131–1143 (1980).
7. B. Littlewood, "Theories of software reliability: How good are they and how they can be improved," *IEEE Transactions on Software Engineering* **SE-6,** No. 5, 489–500 (1980).
8. A. L. Goel and K. Okumoto, "Time-dependent error detection rate model for software reliability and other performance measures," *IEEE Transactions on Reliability* **R-28,** No. 3, 206–211 (1979).
9. A. L. Goel, "Software error detection model with applications," *Journal of Systems and Software* **1,** 243–249 (1980).

Pratap N. Misra *IBM Federal Systems Division, 18100 Frederick Pike, Gaithersburg, Maryland 20879.* Dr. Misra joined IBM in Houston in 1974. From 1974 to 1979 he worked on NASA-sponsored studies related to development of pattern recognition techniques to identify crops in Landsat images. Since 1979 he has been associated with system engineering groups, first in the Ground Based Space Systems project in Houston and currently in the FAA Program in Gaithersburg. Dr. Misra received a B.S. degree from the Indian Institute of Technology, Kanpur, India, in 1965, an M.S. degree from Lehigh University in 1967, both in mechanical engineering, and a Ph.D. in engineering sciences from the University of California, San Diego, in 1973.

Reprint Order No. G321-5195.