What makes a reliable program—Few bugs, or a small failure rate?*

by B. LITTLEWOOD
Mathematics Department
The City University
London EC1V OHB
England.

INTRODUCTION

It is instructive to look at some of the reasons advanced by software developers for their reluctance to use software reliability measurement tools. Here are a few common ones:

(A) "Software reliability models are statistical. Programs are deterministic. If certain input conditions cause a malfunction today, then the same conditions are certain to cause a malfunction if they occur tomorrow. Where is the randomness?"

(B) "I am paid to write reliable programs. I use the best programming methodology to achieve this. Software reliability estimation procedures would not help me to improve the reliability of my programs."

(C) "We verify our software. When it leaves us it is correct."

(D) "I ran your software reliability measurement program on some data from a current project of ours. It said there was an infinite number of bugs left in the program. Who are you trying to kid?"

(E) (same manager as in D, but one week later) "We corrected a couple of bugs and ran the reliability measurement program again. This time it said that there were 200 bugs left. Infinity minus two equals two hundred? Is this the new math?"

(F) "We put a lot of effort into testing. The selection of test data is a systematic process designed to seek out bugs. Reliability estimation based on such test data would be no guide to the performance of the program in a use environment."

(G) "We are writing an air traffic control program. Total system crash would be catastrophic. Other failures range from serious to trivial. Reliability models do not distinguish between failures of differing severity."

Although I have been involved in software reliability modelling for the past decade, and have myself perpetrated a few models, I have a great deal of sympathy with some of the sentiments expressed above. I have a feeling that some of the early models have been oversold, that not enough emphasis has been placed on the underlying modelling assumptions, and that by concentrating on a simple reliability analysis we might be ignoring wider concerns. In this paper I shall be looking at one common deficiency of early models and suggesting a way in which it can be overcome. I hope that, in passing, some new insight into the wider issues will be gained.

THE PROBLEM AND ITS EARLY SOLUTION

In its simplest form the problem is this. We have available some data $t_1, t_2, \ldots, t_n$, representing successive (execution) times between failures of a program. What can we say about the current reliability of the program, and how this will change in the future?

This bald description needs some amplification. In the first place, are we sure what we mean by "reliability" in this context? In A, above, we see one of the difficulties. There is a sense in which software failures are completely predictable: if we know that an input caused a failure in the past, then the same input will cause a failure now (assuming the program is unchanged). Equally, if a program can correctly process an input once, the same program can correctly process the same input forever. Contrast this situation to that of hardware, from which conventional reliability terminology arises. Hardware devices exhibit wear-out and it is not possible to guarantee that the response of a device to a particular input will remain constant. More strongly, we can say that a hardware device is certain to fail ultimately, whereas a program, if perfect, is certain to remain failure-free. Of course, it is questionable whether there is much chance of writing a real-life program in such a way that it is perfect. The principle, however, remains: it is possible to conceive of a program which is never going to fail. This concept of the "perfect" program immediately suggests a way to define software reliability which would not have a hardware parallel. A program which will never fail is one

* This research was supported in part by the US Army, European Research Office, under Grant No. DAERO-79-G-0038.

containing no "defects": no errors (or bugs). The "reliability" of a program is its relative freedom from bugs. Such a concept of reliability, then, is essentially static: it describes the state of the program rather than how the program performs (its failure rate, mean time to failure, etc.). My own view is that we are almost always more concerned with the dynamic reliability of a program than the number of bugs it contains. There are, though, some situations where the number of bugs remaining in a program is of practical interest: the commonest such situation being that where we wish to be assured that *none* are left. It seems sensible, therefore, that we should have reliability models which enable both of the following interpretations of reliability to be used: relative freedom from bugs, relative freedom from failures in operation. It is the relationship between these two concepts of reliability—how the number of bugs remaining in a program affects the performance of the program—which will form the main theme of this paper.

This seems a convenient place to comment on C. When I talk of a perfect program I mean something *more* than correctness. There seem to be two basic objections to formal verification of programs. Most important is the logical objection: the most that can be achieved is a proof that the program is consistent with its specification, not with the informal requirements [1]. Those advocates of program verification who maintain that a program can be "correct," and yet fail to fulfill the requirements demanded by the customer, are just passing the buck. A problem does not disappear by declaring it to be someone else's responsibility. Another objection, which may ultimately be overcome, is that of cost: the sheer effort required to verify programs of realistic size is often completely prohibitive. This seems likely to remain true for a long time. I do not mean to imply that these ideas are not valuable, though. On the contrary, it is clear that they have already had a quite far-reaching and valuable impact on programming methodology.

Notice, by the way, that a program could perform "perfectly" and yet fail a proof of "correctness." Although we would be right to reject the program if we knew the result of the proof, it is clear that in the absence of such knowledge it may be possible to describe the program as highly reliable.

When we look at large real-life programs, written under time and cost constraints, discussions about perfectibility seem merely theological. We shall be almost certain that such programs do contain bugs, that they will eventually produce unacceptable output, and that proofs of correctness (if they were feasible) would fail. Our purpose, then, is to quantify this imperfection: this is why we need reliability studies.

Returning now the basic problem, it is important to be aware of the source of the inter-failure times $t_1, \ldots, t_n$. In most cases this data is collected during the test and integration phase of the project, whilst debugging is in process. We would expect, then, that the reliability of the program is increasing: i.e., there will be a tendency for the t's to be increasing. At any particular stage of this process it is the intention to use the model to measure the current reliability and *predict future reliability*. This brings us to F. These models can predict future performance of a program only on the assumption that there is continuity in the behaviour of the programming team and in the behaviour of the process selecting inputs. This assumption is commonly violated, and in such cases model predictions cannot be trusted. Possibly the commonest situation of this kind is when there is a discontinuity between the test and use environments. In many cases it is simply impractical, or prohibitively slow and expensive, to use an actual (or simulated) use environment to produce inputs for the test phase. Instead, inputs are generated with the specific intention of testing most rigorously those parts of the program which are known a priori to be likely to contain errors: a similar process is often used for hardware, called *stress testing*. It may sometimes be possible to use data from *other* projects to estimate the relationship between the severities of the test and use environments—what Musa [2] calls the *testing compression factor*. My own view is that this will rarely be justified, since every program is essentially unique. We would need to know not merely that the inputs for test and use environments were related similarly between the current program and its predecessors, but that *responses* of the programs to these inputs had the same relationship.

The other source of discontinuities of behaviour which prevent direct use of these models is system integration. If new modules are being integrated into the system during the collection of the t's, then new sources of failure are being introduced and it is unreasonable to expect the estimates based upon an earlier stage of system integration to be valid. It does seem likely in this case, though, that estimation of the magnitude of the reliability discontinuities will sometimes be possible. There is likely to be greater commonality of behaviour between modules of the same system than between different projects. This is an area where further research is needed; at present we shall have to assume that the models are used only after integration, or for the periods of homogeneous behaviour between module integrations.

It may seem, after these areas have been eliminated, that there is very little that software reliability models can be used for. However, if we have a system which has been totally integrated, and we are sure that the test environment (simulated or real) is representative of the use environment, we can use the models to estimate current reliability and predict future reliability *during debugging*. In those cases where it is possible to test *modules* under these conditions, then of course the same reliability estimation can be performed on them. It may even be possible to combine knowledge of the reliability of the modules with structural information about their roles in the system and calculate overall system reliability [3, 4].

Let us now return to the general problem and look at the early solutions. I shall use the notation of Jelinski and Moranda [5], but the models of Shooman [6] and Musa [2] are essentially the same (although it should be noted that Musa introduces many extra refinements over the basic model). It is assumed that the random variables T_i, representing the times between $(i-1)$th and ith failures of the program are *independent* and have the exponential distributions:

$$pdf(t_i) = \lambda_i e^{-\lambda_i t_i}, \quad t_i > 0 \qquad (1)$$

where $\{\lambda_i: \lambda_i > 0\}$ is the sequence of *failure rates* of the program. Note that Musa argues cogently for "time" in this context to represent *execution time,* rather than calendar time.

The reasoning behind assumption (1) is that the input space contains a subset of inputs which will induce failure and that this subset is *encountered randomly*. The process can thus be viewed as a Poisson process with a rate which changes at each event. The assumption seems to be a reasonable one so long as we define "failure" fairly carefully. We would, for example, have to treat a cascade of failures, caused by a single error in the program being encountered once, as a single failure. This accords with usual practice.

The important remaining question concerns the structure of the sequence $\{\lambda_i\}$. It is clearly impossible to estimate each λ_i separately, since there will generally only be a single observation, t_i. More importantly, we wish to be able to project λ_i for *future i*. Jelinski and Moranda make the following assumption (similar assumptions can be found in [2,6]:

"The failure rate at any time is assumed proportional to the current error content of the program . . . the proportionality constant is denoted by ϕ. . . ." ([5], p. 473).

This is equivalent to assuming

$$\lambda_i = (N - i + 1)\phi \qquad (2)$$

where N is the number of bugs (errors) in the program before debugging starts. Each remaining bug contributes an amount ϕ to the overall failure rate of the program, so that when $(i-1)$ bugs have been eliminated there remain $(N-i+1)$. Of course, this assumes that each *failure* of the program results in the immediate removal of one *bug*. In fact it is relatively easy to relax this assumption in order to represent imperfect debugging; this is an issue which I shall not examine here.

The model is now completely specified by the two unknown parameters N and ϕ. These can be estimated from the data $t_1, t_2, \ldots, t_n$ by, say Maximum Likelihood, and estimates of current and future reliability calculated.

A NEW SOLUTION

Consider the quotation from [5] which results in (2). What is being assumed is that each bug in the program contributes equally to the overall failure rate of the program. Thus when a failure occurs (and a bug is fixed) the overall failure rate drops by a fixed constant amount, ϕ. Between bug-fixes the failure rate remains constant. A plot of failure-rate against execution time is shown in Figure 1: all steps are of equal size.

It seems to me wrong to assume all bugs have the same effect on the overall failure rate. In fact it seems likely that the contributions from different bugs to the failure rate of the program will vary quite widely. There is, for example, evidence that the frequencies with which different portions of code are exercised vary enormously. A bug in frequently exercised code will cause failures more frequently than a bug in infrequently exercised code (other factors being equal), i.e., it will contribute more to the failure rate of the program.

A more plausible scenario, then, is that at the beginning of debugging the program contains a pool of N bugs with differing failure rates. Early failures of the program are more likely to be caused by those bugs with the greatest failure rates. Thus early bug-fixes, corresponding to the removal of bugs with larger failure rates, will have greater effect on the overall failure rate. Instead of a plot such as Figure 1, the steps will be of different sizes with larger steps occurring early in the debugging.

Before suggesting in detail how this effect can be modelled, it is instructive to examine the source of the random variation in software failure times as suggested by earlier authors [2,5,6]. All these models assume that the *sole* source of randomness (or uncertainty) lies in the nature of the input stream. Thus in (1), λ_i is treated as a *constant* (given by (2) if N, ϕ known) and the only random variable is T_i. This seems to me to ignore the uncertain nature of program writing and debugging itself. Since we shall be uncertain of the amount any bug contributes to the overall failure rate, we are uncertain of the relationship between λ_{i-1} and λ_i. Thus instead of a sequence $\{\lambda_i\}$ with a *deterministic* relationship between successive terms, as in (2), we should be dealing with a sequence $\{\Lambda_i\}$ of *random variable* failure rates. Another way of looking at this is as follows. Instead of treating the debugging process as a series of deterministic operations on "a program," it can be viewed as the creation of a series of programs, $P_1, P_2, \ldots, P_n$. Program P_i may differ from program P_{i-1} in only a small way—the result of fixing the $(i-1)$th bug—but *it is a different program,* and the difference is *unpredictable*. Just as it is not possible to predict what the sequence of debugging changes will be which produce the sequence $\{P_i\}$ of programs, so the sequence of failures rates is not predictable. It must be treated as a sequence of random variables $\{\Lambda_i\}$.

The two sources of uncertainty can be modelled in the following way. Assume, as do earlier authors [2,5,6], that the uncertainty in the input stream causes the execution time to next failure to be conditionally exponentially distributed. That is,

$$pdf(t \mid \Lambda = \lambda) = \lambda e^{-\lambda t}, \quad t > 0. \qquad (3)$$

We shall assume perfect debugging, for simplicity. So that when i failures have occurred we have removed i bugs. Let total execution time be $t^{(0)}$ at this stage (see Figure 3). Then

$$\Lambda = \Phi_1 + \Phi_2 + \cdots + \Phi_{N-i} \qquad (4)$$

where N is the initial number of bugs (unknown) and Φ_r represents the (random variable) contribution to the overall failure rate of the rth bug among the remaining $(N-i)$ bugs. It solely remains to find the distribution of Φ_r for all $t^{(0)}$, r. Clearly

$$pdf(\phi_r) \equiv pdf(\phi_s) \text{ for all } r,s.$$

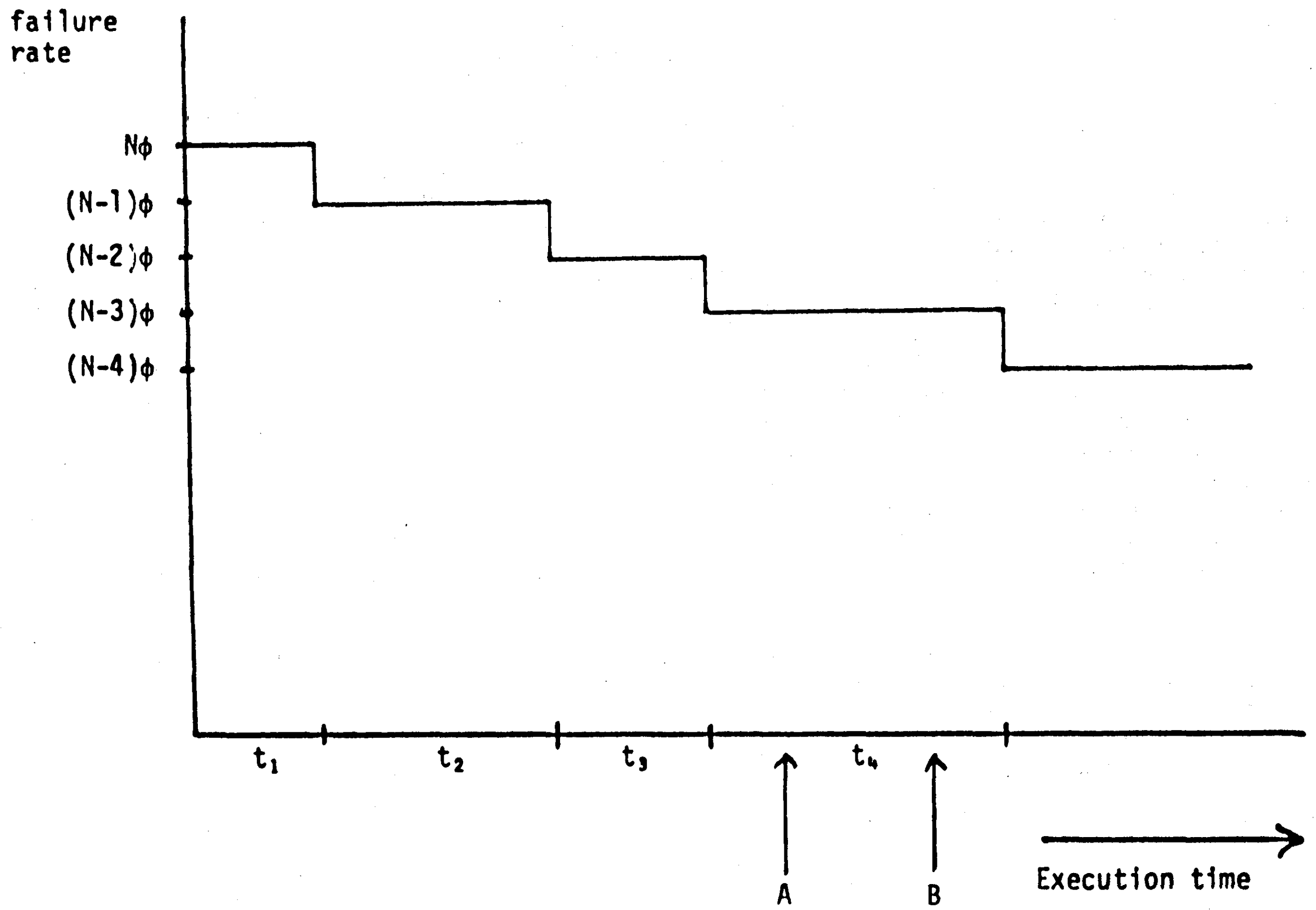

Figure 1.

This merely states that at each stage our uncertainty (our ignorance) about the bugs which remain is the same for each: we cannot distinguish between them.

Let us represent our initial uncertainty about the Φ's by a Gamma (α,β) distribution:

$$pdf(\phi)=\frac{\beta^{\alpha}\phi^{\alpha-1}e^{-\beta\phi}}{\Gamma(\alpha)}\ (\phi>0). \tag{5}$$

Then the distribution of each of the Φ's in (4), by Bayes Theorem, is

$$pdf(\phi\,|\,\text{bug has survived detection for a time } t^{(0)}) \tag{6}$$
$$=\frac{Pr\{\text{no failure of this bug in } (0,t^{(0)})\,|\,\Phi=\phi\}.pdf(\phi)}{\int Pr\{\text{no failure of this bug in } (0,t^{(0)})\,|\,\Phi=\phi\}.pdf(\phi)d\phi}.$$

Substituting (5) into (6) and simplifying we find that the distribution of a Φ in (4) is

$$\text{Gamma}(\alpha,\beta+t^{(0)}). \tag{7}$$

Since the sum of independent, identically distributed Gamma random variables is itself Gamma distributed, we find from (4) that the distribution of Λ is

$$\text{Gamma}((N-i)\alpha,\beta+t^{(0)}) \tag{8}$$

Finally, from (3) and (8) we find that the distribution of the time to next failure, T, when i bugs have been detected and execution time $t^{(0)}$ has elapsed (Figure 3) is

$$pdf(t)=\int_0^{\infty} pdf(t\,|\,\Lambda=\lambda)pdf(\lambda)d\lambda$$
$$=\frac{(N-i)\alpha(\beta+t^{(0)})^{(N-i)\alpha}}{(\beta+t^{(0)}+t)^{(N-i)\alpha+1}}, \tag{9}$$

which is a *Pareto distribution*. This result should be compared with the exponential distribution of the Jelinski-Moranda model, (1) and (2). Full details of this new model can be found in [7], including examples of how it can be used to predict future reliability.

Consider the failure rate at the arrowed epoch in Figure 3. This is

$$\frac{(N-i)\alpha}{\beta+t^{(0)}}. \tag{10}$$

Notice how this changes as debugging proceeds. When a failure occurs and a bug is fixed, the failure rate drops by an amount $\alpha/(\beta+t^{(0)})$; early bug-fixes, with small $t^{(0)}$, cause greater reductions in the program's failure rate than later

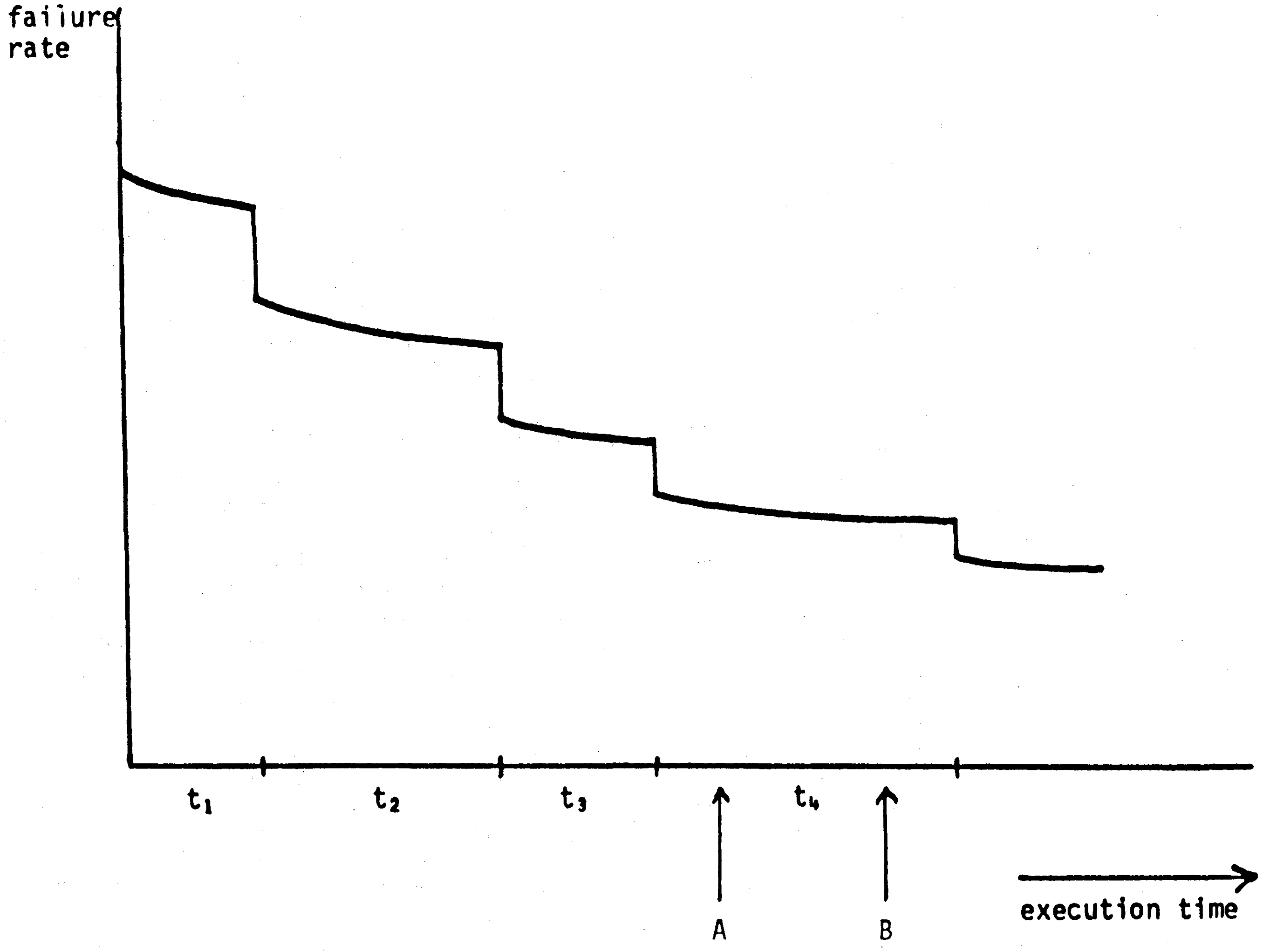

Figure 2.

ones. During periods of failure-free operation, between bug-fixes, the failure rate decreases continuously as $t^{(0)}$ increases. We thus get a plot of failure rate against time of the kind shown in Figure 2.

As a justification of the *decreasing failure rate* (DFR) property of the Pareto distributions, consider the two epochs A and B in Figures 1, 2. Assume that a judgment of the reliability of the program has been made at A. How would you expect the reliability to have changed at B, after a further period of failure-free operation? It seems to me more plausible that we should be reassured by the extra evidence (Figure 2), since this is evidence of good performance, than that we should believe the reliability unchanged (Figure 1). What is in fact happening, in this model, between A and B is that we are gathering new information about the distribution of the failure rates of remaining bugs—specifically we are increasing $t^{(0)}$ in (7).

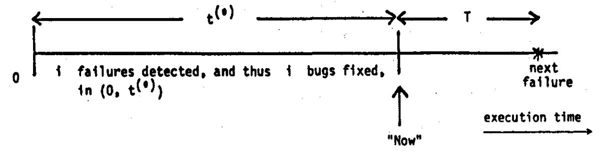
$t^{(\bullet)}$
T
0
i failures detected, and thus i bugs fixed, in (0, $t^{(\bullet)}$)
next failure
"Now"
execution time

Figure 3.

IMPLICATIONS OF THE NEW MODEL

The intention behind all models of this kind is the same. We wish to be able to estimate both static reliability (number of remaining bugs) and dynamic reliability (frequency of failures) of a program. I have argued in the previous section that early models make a false assumption about the relationship between these two measures. Let us now look at the implications of the new model for reliability estimation and, in particular, what consequences would follow from using one of the naive models.

It should be acknowledged, first, that the new model is a little more complicated. It is necessary to estimate three parameters (N, α, β) from the available data, rather than the two parameters (N, ϕ) of the earlier models. This should not present any unusual difficulties.

An important observation is that the Jelinski-Moranda model is a special case of the new model. If we let $\alpha \rightarrow \infty$, $\beta \rightarrow \infty$ in such a way that $\alpha/\beta = \phi$ in (5), we find that the Gamma (α,β) distribution becomes concentrated at ϕ. Thus, if a particular data set produces values of α and β which are very large, the Jelinski-Moranda model will provide a good approximation with $\phi = \alpha/\beta$. Of course, this operates in reverse: if the best-fitting values of α and β are not large, this implies that the Jelinski-Moranda model would be a poor approximation to the underlying process. In summary, then, nothing can be lost by using the new model instead of the old ones; and something important may be gained.

Assuming that the model does not reduce to the Jelinski-Moranda one, in what ways will it give a different picture of the reliability growth taking place during debugging?

In the first place, it suggests that there is a law of diminishing returns operating in debugging. The reliability improvements gained from successive bug-fixes gradually become smaller and smaller. This implies that estimates of N may be larger than for the Jelinski-Moranda model without necessarily implying equivalently large estimates of dynamic reliability (e.g. failure-rate). This law of diminishing returns suggests that it will often be appropriate to end debugging before the program is judged bug-free, without implying that such a program is unreliable. In other words, if we are solely interested in the performance of the program (failure rate, mean time to failure, etc.) we can accept a program known to contain many bugs, as long as we are assured that these bugs cause failures infrequently. This seems to me to accord better with intuition and real-life practice than the Jelinski-Moranda assumption, which deems all bugs to have the same contribution to overall reliability. We have all, I think, encountered programs containing bugs which we were prepared to live with.

My own view, then is that almost always the appropriate criterion to adopt is dynamic reliability rather than number of remaining bugs. There are, though, situations where we might wish to have a very high assurance that no bugs remained. Examples would be an air traffic control system or nuclear power station safety system. It would not be sufficient to know that the program was very reliable, whilst containing bugs, if these bugs included ones with catastrophic consequences. This observation reveals the weakness of an analysis purely in terms of the *counting* of failures and bugs (see quotation G). What we ought to have are models which enable us to predict the process of *consequences* of failures. There is, unfortunately, little data or research in this area—no doubt partly due to a natural reluctance to accept a quantification of the unthinkable. We demand, instead, a high assurance that the system is "perfect."

If we wish to stop debugging only when we have a high assurance that the last bug has been removed, the new model gives some disturbing answers. Since the model will often tend to suggest that many bugs remain (albeit ones with small failure rates), and the successive times between their removals are Pareto distributed (the Pareto distribution has a long tail, i.e. large values occur with greater frequency than in the exponential case), we find that estimates of the time required to end debugging are very large. Often, with large systems, they will be prohibitively large. It is, effectively, *impossible* to make a large system bug-free.

Again, this is not surprising: it seems to accord with experience. But it is worrying. Contrast this with the hardware case: one of the important results of hardware reliability is that it is possible to make a system with any given reliability using components of any given unreliability. We cannot do this for software. Does this mean that we cannot use software for such critical applications? In practice we seem to have little choice.

SUMMARY AND CONCLUSIONS

The new model that I have described does, I think, represent the relationship between static and dynamic measures of software reliability more naturally than earlier models. I would not, however, suggest that this or any other model is definitive. Indeed, I suspect that it will be a long time before we are able to apply these techniques with confidence to every software development project. In the meantime, we have some techniques which are useful when treated with care: in particular, it is necessary to be sure that the underlying modelling assumptions do apply to the project under examination. So my reply to speaker B would be that, whilst agreeing that software reliability techniques do not of themselves help to improve reliability, it is a brave manager who asserts that his programs are reliable without in some way measuring this reliability.

On the debit side, there is still a great deal of work remaining. In my view, the single biggest gap in our knowledge lies in the area of costs/consequences of failures. We have little in the way of theory, and very little data; yet we all recognise that a reliability theory is only one step on the road to a more comprehensive cost theory. This is an area which urgently requires study.

Another area where progress has been disappointing is that of structural models. It seems intuitively clear that the structure of a program will affect its reliability, but so far there is no effective way of incorporating into a reliability model the wealth of information available about program structure.

Finally, a comment on quotations D and E. It is true that "difficulties" have been experienced with parameter estimation of the early models, and this has tended to alienate some potential users. It should be said that these problems generally only occur with small data sets (i.e. at the very beginning of the debugging period), when the evidence for growth in reliability is slight. Since all the models depend upon an assumption of reliability growth, it is not surprising that things can go wrong when such growth is not clearly evident in the data. It must always be borne in mind that these techniques are not a magical panacea: they are simply systematic methods of estimating what is actually present in data.

REFERENCES

1. DeMillo, R. A., Lipton, R. J. and Perlis, A. J., "Social processes and proofs of theorems and programs," *Comm. ACM* May 1979, Vol. 22, No. 5, pp. 271-280.
2. Musa, J. D., "A theory of software reliability and its application," *IEEE Trans. on Software Engineering,* Vol. SE-1, Sept. 1975, pp. 312-327.
3. Littlewood, B., "A reliability model for systems with Markov structure," *Applied Statistics* (*J. Royal Statist. Soc.,* Series C), Vol. 24, No. 2, 1975, pp. 172-177.
4. Littlewood, B., "A software reliability model for modular program structure," *IEEE Trans. on Reliability* (Special Issue on Software Reliability), Vol. R-28, No. 3, August 1979, pp. 241-246.
5. Jelinski, Z. and Moranda, P. B., "Software reliability research," in *Statistical Computer Performance Evaluation,* Ed.: W. Freiberger. New York: Academic, pp. 465-484.
6. Shooman, M., "Operational testing and software reliability during program development," *Record 1973 IEEE Symposium on Computer Software Reliability,* New York, NY, April 30-May 2, 1973, pp. 51-57.
7. Littlewood, B., "A Bayesian differential debugging model for software reliability," in *Proceedings of Workshop on Quantitative Software Models,* Kiamesha Lake, NY, Oct. 9-11, 1979 (to appear).