Software Reliability is Not an Equation

Glenford J. Myers

Moranda’s remarks in The Open Channel in Computer, April 1978, on my book, Software Reliability: Principles and Practices, fall into two general areas. First, he feels that the book “is not about software reliability as it has come to be defined.” Second, he seems defensive about my “low opinion” (his words) of probability-based models, particularly his model.

Concerning the first point, he never bothered to give us his opinion of what software reliability means, but the inference of his remarks is that software reliability is just a matter of numbers—for instance, predictions of MTTF, the number of errors in a program, and the like. I agree that this is related to software reliability, but it is only a small part. If we use the Program Committee of the 1975 International Conference on Reliable Software as mediators, one sees, by scanning the conference proceedings, that such topics as design, testing, languages, program proving, and management issues are considered important components in the study of software reliability.

I don’t have a “low opinion” of reliability models; rather, I am skeptical and am waiting to be convinced, and this is the message I conveyed in my book. If one studies the Jelinski/Moranda model, one sees that it is the old constant-hazard-rate hardware reliability model. The only change they made was to introduce a series of models; whenever an error is repaired, a new model is applied having a lower, but still constant, hazard function. I am skeptical of this model because, as we know, software reliability differs sharply from hardware reliability. The modeling of software reliability entails the modeling of two types of human behavior. First, software doesn’t wear out; it fails because of a mistake made by a human designer. Second, software reliability is not time-dependent; whether or not a compiler will fail in the next hour is dependent upon the inputs fed to the compiler by its users. Since Moranda’s model is based on the hardware model of purely time-dependent wear-out (the middle section of the bathtub curve), but since software errors are human-generated and appear as a function of usage rather than time, I can see no connection. To be convinced of a connection, I’d have to see tons of empirical evidence. Unfortunately, the many recent papers on reliability models have been largely devoid of such data.

Several of Moranda’s other points seem to contradict the experience of many of us who work in this field. I criticized his model because it assumes that all corrections are “correct.” Moranda refers to this as “not uncommon.” I have to disagree, since published error studies have indicated that a nontrivial percentage of software errors are due to prior “corrections.” Other modelers seem to agree with me, since models have been developed that assume that error corrections are not perfect. Moranda also refers to “errors of the kind normally edited out by utility programs.” I’m anxious to hear more about these invaluable mysterious programs. He also feels that the simple model employing two independent test groups (actually just a variation on the error-seeding model) will not work because crashes of the program will stall the testing efforts. In my experience, “crashes” are not a serious problem; they represent only a tiny, sometimes zero, percentage of the errors encountered. The vast majority of errors in programs involve incorrect function, incorrect output, destroyed data, etc.

I must also take issue with Moranda’s journalism—in particular, his quoting me out of context. For instance, in discussing assumption number one, Moranda states, “The first, which Myers considers the ‘basic assumption’ is that all errors have equal severity.” This is misleading, for Moranda has substituted his word “the” where I said “One basic assumption.” In fact, the reader can easily see that I do not consider this “the basic assumption,” since the next paragraph in my book (p. 334) begins with “Of course, the major assumption is concluding that all programs have the f(t) of Figure 18.2.” Note that Moranda avoids discussing this assumption. In fact, it is the crux of the model and has been recently criticized by others. Also, in discussing assumption one, Moranda points out that one can deal with the issue of error severity by applying the model on subcategories of the data. What he omits, however, is the fact that I made the same point by stating, “This assumption can be easily removed by categorizing the errors by severity and estimating a different N and K for each category.”

As another example of what I mean by inaccurate quotation, in referring to my discussion of the seeding of errors Moranda states, “He says that this seems to be a ‘solvable’ and ‘relatively minor’ problem.” However, the complete sentence in my book is “However, in comparison to the problems surrounding the other reliability models, this problem seems relatively minor and solvable.” There is quite a difference between what I said and what Moranda says I said.

Moranda also implies several times that I seem to be confused about the definition of software reliability. The problem seems to be his failure to
distinguish between measurement parameters and cause/effect parameters. Just as one can measure birth rates or velocities with respect to time, one can measure software reliability with respect to time. However, the fact that something is measured with respect to time does not necessarily imply that it is a function of time (i.e., caused by time). Just as we realize that birth rate and velocity are effects of other parameters and not of time, we should realize that software reliability is an effect of design errors and usage (inputs), not of absolute or even usage time. (Hardware reliability, on the other hand, is usually treated as an effect of time.)

Finally, Moranda missed one important point in my book. The vastly different hazard functions in Figure 18.4 are not hazard functions that I am seriously proposing. I included them to show that one could make seemingly plausible arguments for many different hazard functions, just as Jelinski and Moranda did for the one they introduced. The point is that plausible arguments and fancy equations aren't worth the price of a cup of coffee. Show me some scientific experimentation validating the model against a large set of actual program-error data, and I'll start listening. Until that time, I remain skeptical. Don't jump on me for my skepticism; ask yourself where your proof is. And if we ever reach this point, I still won't agree that software reliability is just reliability prediction.


As Editor Don Christiansen and the staff of IEEE Spectrum struggle to cope with the avalanche of entries in the epochal Micromouse Contest, scheduled for initial time trials at the NCC in Anaheim this month, Open Channel pauses to look at some of the mouse-related mail that's making its way over our transom these days.

Some Micromouse Strategies

A. Nonymouse

In most competitions, strategy is critical. The Micromouse Contest sponsored by IEEE Spectrum and Computer is no exception. (In case you're not familiar with it, the Micromouse Contest is aimed at building an electronic mouse that finds its way through a maze faster than anyone else's mouse does.)

Faced with choosing a strategy for my own mouse, I explored several possibilities. Now, having selected what I regard as the winning strategy, I freely offer the following hints to my competitors.

1. Pregnant Mouse—At each turn in the maze this mouse gives birth to another pregnant mouse, which in turn gives birth to another, until eventually all possible paths are tried.

2. Metal Munching Moon Mouse—Eats its way through the walls.

3. Kangaroo Mouse—Jumps over the walls.

4. Tunneler—Comes equipped with a rotary drill for drilling through walls.

5. The Rat—Eats other contestant mice.


10. Seymour Mouse—Communicates with a remote CRAY 1.

11. Trojan Mouse—Rolls across starting line, opens a door, and releases thousands of mice to try different paths.