SOFTWARE RELIABILITY ESTIMATIONS/PROJECTIONS, CUMULATIVE AND INSTANTANEOUS

David Dwyer*
Software Reliability Engineer, BAE Systems, Nashua, NH
M.S. in Computer Science, Rivier College 1999

Keywords: *Software Reliability, Operational Profile, Predictors, Estimators, Instantaneous Software Reliability*

Abstract

Musa's methods [1] for software development and test planning combined with Duane's learning curve approach [2] for hardware reliability growth testing provide an efficient means for estimating and demonstrating reliability requirements. This paper reviews the analyses of Tractenberg [3] and Downs [4] that provide a foundation for Musa's basic (linear) model. I have used Musa's basic model in combination with an approach similar to that used by Duane and Codier [2] for derivation of a formula for instantaneous failure rate for hardware to develop formulas for the estimation of instantaneous failure rate for software. These calculations show significant correlation with interval estimates and provide an efficient method for showing the achievement of goal reliability for software without a separate demonstration test.

1 Introduction

1.1 Background

The data for this paper was chosen from a large software development program where there was a significant interest in reliability. The software test data has been analyzed to estimate software reliability can be estimated although initial test planning did not follow accepted Software Reliability guidelines. The software testing discussed in this paper is from a console-based system where the sequence of execution paths closely resembles testing to the operational profile. The actual deviation from the operational profile is unknown but the deviation is been assumed to result in errors under 10%. The methods for presenting data that are used in this paper are described by Ann Marie Neufelder [5] in her book "Ensuring Software Reliability".

The following notations are used:

λ_i instantaneous failure rate, or "error rate"
λ_{cum} cumulative failure rate after some number of faults, 'j' are detected
j the number of faults [aka "error sites"] removed by time 'T'

T	test time during which 'j' faults occur
Φ	constant of proportionality between λ and j
Pr	probability'
c	Number of paths affected by a fault
M	total number of paths.

1.1 Digression on Software vs. Hardware Reliability

Hardware reliability engineering started in the 1940s when it was observed that electronic equipment that passed qualification tests and quality inspections often did not last long in service, i.e., had a low MTBF. For electronic reliability, the measure of complexity is in the number and type of electrical components and the stresses imposed on them and these relate to its failure rate, a value which may be measured by regression analysis. Some approaches to electronic reliability assume that all failures involve wear-out mechanisms in the components related to the fatigue life of their materials under their imposed mechanical stresses. For software, the measure of complexity is related primarily to *LOC* (lines of code), *ELOC* (executable lines of code) or *SLOC* (source lines of code). Structural complexity, related to the data structures used ('if' statements, records, etc.) is a better measure, however most metrics have been tabulated in terms of *SLOC*. Hardware reliability requirements provided an impetus to provide for safety margins in the mechanical stresses, reduced variability gain tolerances, input impedance, breakdown voltage, etc. Reliability engineering brought on a proliferation of design guidelines, statistical tests, etc., to address the problems of hardware complexity.

But margins of safety do not mean as much in software because it does not wear out or fatigue and time is not the best measure of its reliability because software doesn't really have x failures per million [processor] operating hours, it has *x* failures per million unique executions. Unique because once a process has been successfully executed, it is not going to fail in the future. But executions are hard to keep track of so test time is the usual metric for monitoring failure rate. Not only that, but "wall clock" time, not processor time, is the best that is generally available. For the testing that produced the data for this project, the number of eight-hour work shifts/day was all that was known so that became the time basis for calculating failure rate. The assumption was made that, on average, the number of executions per work shift stayed the same throughout the test period. Thus, the metric used for failure rate was failures (detected faults)/eight-hour work shift.

2 Model Development

2.1 Effects of Software Structure and Test Methodology

Tractenberg simulated software with errors spaced throughout the code in six different patterns and tested this simulation in four different ways. He defined the following fundamental terms:

- an "error site" is a mistake made in the requirements, design or coding of software which, if executed, results in undesirable processing;
- "error rate" is the rate of detecting new error sites during system testing or operation;
- "uniform testing" is a condition wherein, during equal periods of testing, every instruction in a software system is tested a constant amount and has the same probability of being tested.

The results of his simulation testing showed that the error rates were linearly proportional to the number of remaining error sites when all error sites have an equal detection probability. The result would be a plot of failure rate that would decrease linearly as errors were corrected. An example of non-linear testing examined by Tractenberg was the common practice of function testing wherein each function is exhaustively tested, one at a time. Another non-linear method he examined was testing to the operational profile, or "biased testing". The resulting plot of function testing is an error rate that is flat over time (or executions). With regard to the use of Musa's linear model where the testing was to the operational profile, Tractenberg [5] stated: "As for the applicability of the linear model to operational environments, these simulation results indicate that the model can be used (linear correlation coefficient>0.90) where the least used functions in a system are run at least 25% as often as the most used functions."

2.2 Effects of Biased Testing and Fault Density

Downs [4] also investigated the issues associated with random vs. biased testing. He stated that an ideal approach to structuring tests for software reliability would take the following into consideration:

- The execution of software takes the form of execution of a sequence of paths;
- '*c*', the actual number of paths affected by an arbitrary fault, is unknown and can be treated as a random variable;
- Not all paths are equally likely to be executed in a randomly selected execution [operational] profile.

According to Downs [4], very little is known about the nature of the random variable of the number of paths affected by an arbitrary fault. "This problem is not easy analytically and is made more difficult by the fact that there is very little information available relating to the type of distribution which should be ascribed to [the number of paths]." Downs [4] states:

- In the operational phase of many large software systems, some sections of code are executed much more frequently than others are. In addition,
- Faults located in heavily used sections of code are much more likely to be detected early.

These two facts indicate that the step increases in the failure rate functions should be large in the initial stages of testing and gradually decrease as testing proceeds. If a plot of failure rate is made VS the number of faults, a "convex" curve should be obtained. This paper is concerned with the problem of estimating the reliability of software during a structured test environment, and then predicting what it would be during an actual use environment. Because of this, the testing must:

- resemble the way the software will finally be used, and
- the predictions must include the effects of all corrective actions implemented, not just be a cumulative measure of test results.

A pure approach for a structured test environment for uniform testing would assume the following:

- Each time a path is selected for testing, all paths are equally likely to be selected.
- The actual number of paths affected by an arbitrary fault is a constant.

Such uniform testing would most quickly reduce the number of errors in the software but would not be efficient in reducing operational failure rate. But our testing will be biased, not uniform. Test data will be collected at the system level, since lower level tests, although important in debugging software, do not indicate all the interaction problems which become evident at the system level. Software would be exercised by a sequence of unique test vectors and the results measured and estimates of *MTTF*

(Mean Time to Failure) can be made from the data. The number of failures per execution (failure rate) will be plotted on the *x* axis. The total number of faults will be plotted on the *y*-axis. Initially, the plot may indicate an increasing failure rate VS faults but eventually, the failure rate should follow a straight line constant decreasing path that points to the estimation of the total number of faults, *N*, on the *y* axis.

The execution profile defines the probabilities with which individual paths are selected for execution. This study will assume the following:

- The input data that is supplied to the software system is governed by an execution profile which remains invariant in the intervals between fault removals
- The number of paths affected by each fault is a constant.
- Downs showed that the error introduced by the second approximation is insignificant in most real software systems.

Downs derived the following *Lemma* [4]: "If the execution profile of a software system is invariant in the intervals between fault removals", then the software failure rate in any such interval is given by the following formula:

$$\lambda = -r \log[Pr\{\text{a path selected for execution is fault free}\}] \quad (1)$$

Where r = the number of paths executed over unit time.

Downs [4] derived this without any assumption being made regarding the manner in which faults are distributed over a software system. The following paragraphs, however, describe assumptions made regarding the distribution of the number of faults in a path.

Downs investigated uniform testing wherein every time the software is executed; each path is equally likely to be selected. This selection procedure does not discriminate between paths which have already been tested and those which have not; in other words, it corresponds to the classical statistical procedure of random selection *with replacement.* Also, the faults are allocated, one at a time, to 'c' randomly chosen paths. Given that and with *M* total paths, then each time a fault is allocated, any given path receives a fault with probability *c/M*. With this path selection strategy, the execution profile does not change in the intervals between fault removals.

Downs then shows [4] that for software containing *N* initial faults, (randomly distributed over *M* logic paths), the probability that an arbitrarily selected path is fault free is $(1 - c/M)^N$. Then the initial failure rate, λ_0, is given by the following formula:

$$\lambda_0 = -Nr \log(1 - c/M) \quad (2)$$

Removal of faults from the software affects the distribution of the number of faults in a path. The manner in which this distribution is affected depends upon the way in which faults are removed from paths containing more than one fault. If, for instance, it is assumed that execution of a path containing more than one fault leads to the detection and removal of only one fault, then the distribution of the number of faults in a path will cease to be binomial after the removal of the first fault. This is because under such an assumption, considering that all paths are equally likely to be selected, those faults occurring in paths containing more than one fault are less likely to be eliminated than those occurring in paths containing one fault only. If, on the other hand, it is assumed that execution of a path containing more than one fault leads to detection and removal of all faults in that path, then all faults have an equal likelihood of removal and the distribution of the number of faults in a path will remain binomial.

Fortunately, in relation to software systems which are large enough for models of the type Downs developed, the discussion contained in the above paragraph has little relevance. This follows from the fact that, in large software systems, the number of logic paths, *M*, is an extremely large number,

implying that the parameter c/M is a very small number. This in turn implies that the number of paths containing more than one fault is a very small fraction of M. Consequently, even if it cannot be assumed that all faults are removed from each path whose execution leads to system failure, the assumption that each fault has equal probability of removal will lead to very little error. Hence, in such cases, a very good approximation is obtained by assuming that the distribution of the number of faults in a path remains binomial as faults are removed.

With this assumption, after the removal of j faults, the failure rate is:

$$\lambda_j = (N - j)\Phi, \tag{3}$$

where j = the number of corrected faults, and

$$\Phi = -r \log(1 - c/M) \tag{4}$$

This result indicates that when testing takes the form of random (and uniform) selection of paths (with replacement) and when the distribution of the number of faults per path is binomial, the failure rate is proportional to the number of faults in the software [4]. Also, it is worth noting that, when $c<<M$, we have the following:

$$\lambda_j \approx (N - j)r(c/M) \tag{5}$$

This equation indicates that the failure rate is approximately proportional to the number of faults per path. This also means that for a structured test, the failure rate (λ_j) plotted against the number of faults (j) should be a straight line. So far, we have discussed failures VS: paths, executions, processor hours, wall clock hours and eight-hour work shifts. Discussions from here on refer to failures per eight hour work shift as failure rate. Figure 1 from Down's paper [4] showing this plot of failure rate VS faults is reproduced below, along with a definition of terms.

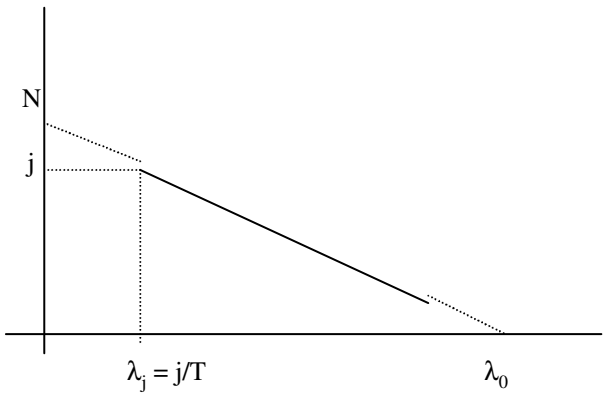

Figure 1: Failure Rate Plotted vs. Failure Number

Given:

N – total initial number of faults
$\lambda(0)$ – initial failure rate => 0 faults detected/ corrected (at start of testing)
λ_j – cumulative failure rate after some number of faults is detected, 'j'
j – the number of faults removed over time
λ_i – instantaneous failure rate

T – time.

Downs [4] examined test efficiency schemes to achieve a desired operational failure rate when the software is operated in accordance with the operational profile. Ratios of *c/M* from 10^{-5} to 10^{-6} were combined with total initial failure counts of 100 and 1,000. The operational profile assumed 40% of the code is used 90% of the time. The results in all cases indicate that most used sections of code must be tested more; however, testing to the exact operational profile was not necessary.

2.3 Derivation of Formula for Instantaneous Failure Rate for Software

The following shows Duane's derivation for instantaneous failure rate for hardware:

$$\lambda_c = F / T$$
$$= kT^{(-m)}$$
$$F = kT^{(1-m)}$$
$$\lambda_i = \partial F / \partial T \quad \text{//instantaneous failure rate for hardware//}$$
$$\lambda_i = k(1-m)T^{(-m)}$$
$$\lambda_i = (1-m)\lambda_c$$

The following is an analogous derivation of a formula for instantaneous failure rate for software [aka "failure intensity"] based on the well known Duane derivation of instantaneous failure rate for hardware ('*j*' here substituted for Duane's '*c*' to avoid confusion with Down's '*c*'):

$$\lambda_j = j / T \quad \text{//definition of cumulative software failure rate//}$$
$$= (N - j)\Phi \quad \text{//from Equation 2 above//}$$

$$j = T(N - j)\Phi \tag{6}$$

$$\lambda_i = \partial j / \partial T \quad \text{//instantaneous software failure rate//} \tag{7}$$

$$\lambda_i = (N - j)\Phi + T(-\partial j / \partial T)\Phi \tag{8}$$

From Eq. (3):

$$\lambda_i(1 + T\Phi) = \lambda_j \tag{9}$$

$$\lambda_i = \lambda_j / (1 + T\Phi) \tag{10}$$

3 Study Results

The following rules will apply to analyses of software test data for this study:

- Time will not be corrected to processor hours.
- The cumulative number of eight hour shifts will be recorded VS the number of failures.
- Each first instance of an error will be plotted.
- The data point for the last failure will be plotted at the end of the test time, when there is a significant interval of time at the end of testing without error.
- Only system level test data will be used.
- Errors will be designated as priority 1, 2, 3, 4 or 5 where:
  - Priority 1: "Prevents mission essential capability"
  - Priority 2: "Adversely affects mission essential capability with no alternative workaround"
  - Priority 3: "Adversely affects mission essential capability with alternative workaround"
  - Priority 4: "Operator inconvenience and does not affect mission"
  - Priority 5: "All other problem"
- The following data will be plotted:
- Priority 1 only,
- Priority 1 and 2 combined, and
- Priority 1, 2 and 3 combined.

3.1 Discriminating Between Priority 1 and Priority 2 Test Data

In this analysis, Musa's basic model is applied to the test data for a large software system (over 400,000 LOC) which was tested at a console. Testing was not structured to be in accordance with the operational profile but is considered close to that and "most used sections of code must be tested more". The testing was not uniform testing.

Figure 2 shows that when the data plots for Priority 1 and 2 failures combined show an interval of constant failure rate where the number of failures is increasing but the failure rate remains the same. This is because there were about eight times as many priority 2 failures as there were priority 1 failures and a decision was made to focus only on priority 1 failures until they were fixed. The result was that a large number of priority 2 failures occurred before the failure rate started to become reduced. This underscores the importance of distinguishing between the priority levels of failures and of taking into consideration changes in direction of test conduct in plotting data. Another approach is to consider plotting only the first occurrence of failure; however caution must be exercised if it appears that corrective action effectiveness is low.

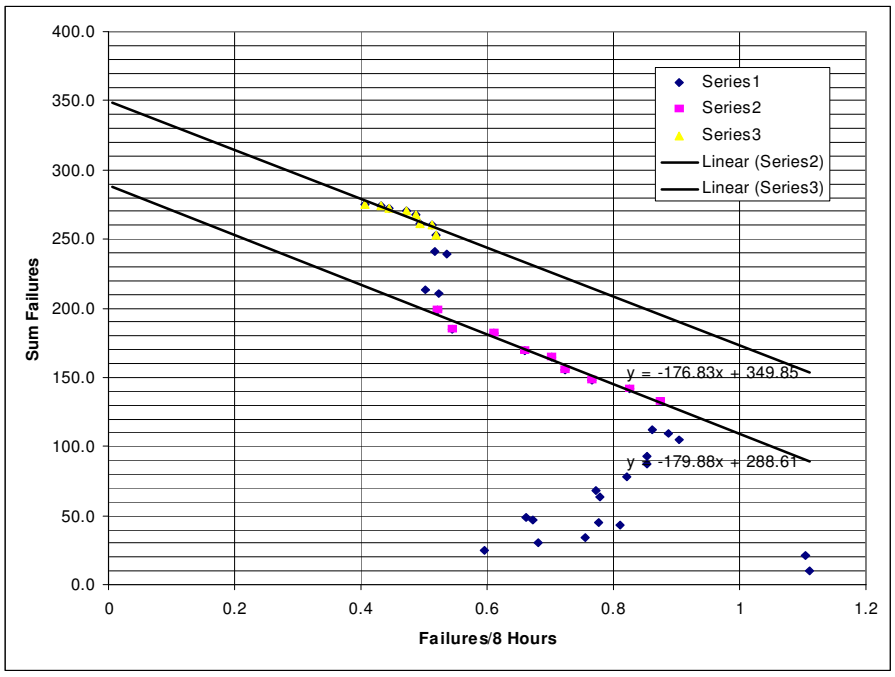

Figure 2: Priority 1 and 2 Failures Combined

3.2 Plots of Priority 1 Data

Figure 3 shows a plot of the data for priority 1 failures only. Since priority 1 failures were always addressed as soon as they occurred, the resultant plot follows a straight line after an initial debugging period where software modules were incrementally added to the system. Thus, with each test interval, the probability of failure followed the expected result for linear testing.

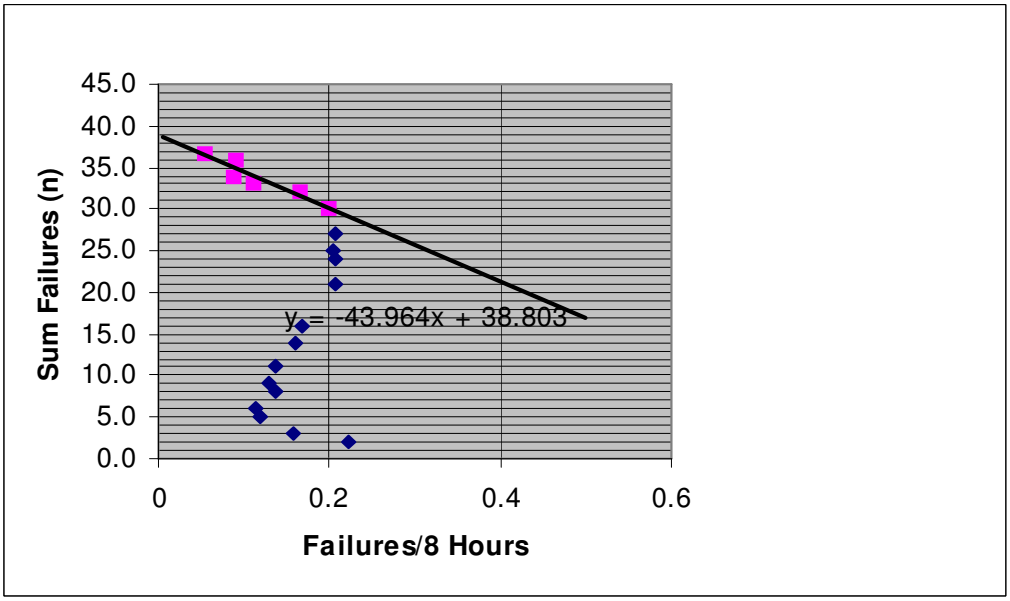

Figure 3: Priority 1 Failures

3.3 Instantaneous Failure Rate Calculations and Correlation with Interval Measurements

Figure 4 shows the results of subsequent testing where only the first occurrence of a failure was plotted. The linear plot is what would be expected as long as concurrent corrective actions did not introduce new failures.

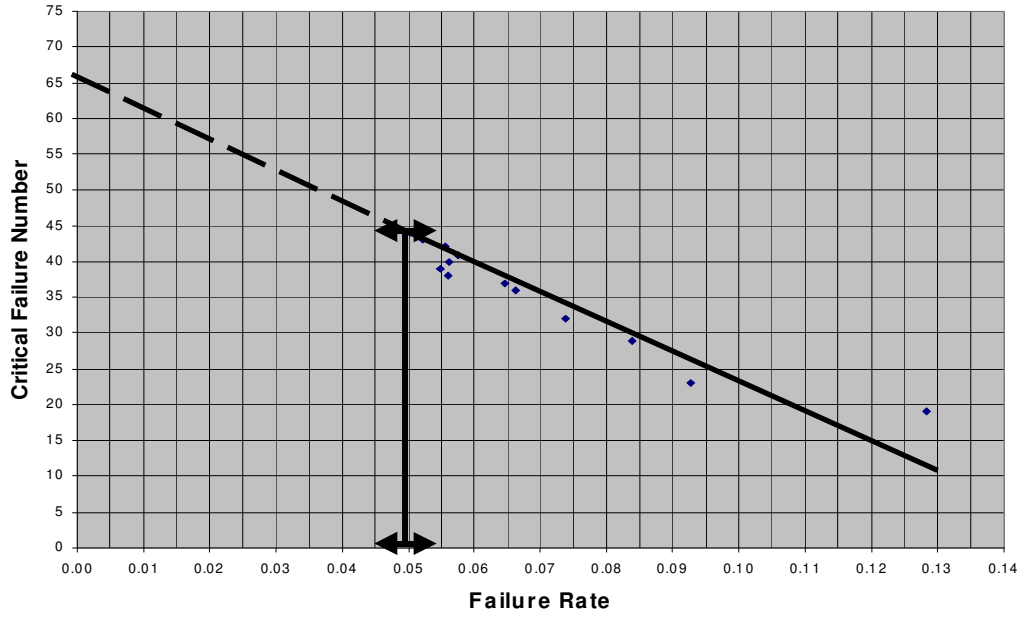

Figure 4: First Occurrence Only Plotted

A comparison can be made between the value of instantaneous failure rate calculated at $\lambda_j = 0.050$. From the previous graph:

$$j = -431\lambda_j + 66 \tag{11}$$

The data from the graph is summarized in Table 1:

Table 1. Interval Failure Rate Data

j	λ_j	$T = j/\lambda_j$
44.00	0.050	880
41.84	0.055	761
46.16	0.045	1026

From this data, we can estimate the following:

$$\lambda_i = (46.16 - 41.84)/(1{,}026 - 761) = 4.32/265 = \mathbf{0.016}$$

From the formula for instantaneous failure rate, $\lambda_i = \lambda_j/(1 + T\Phi)$, at $\Phi = 1/431$ and $T = 880$:

$$\lambda_i = 0.050/(1 + 880/431) = 0.050/(1 + 2.04) = 0.050/3.04 = \mathbf{0.016.}$$

Although the results of two approaches agree, the second calculation is much more significant since it involves all the 880 hours of data.

The last diagram shows a subsequent plot of data wherein the following directives of Duane and Codier [2] that are considered applicable to Musa's plots: "When fitting a straight line to a Duane plot, the cumulative nature of the data points should be born in mind. The latter points, having more information content must be given more weight than earlier points and the normal curve-fitting procedure of drawing the line through the 'center of gravity' of all the points should not be used. Unless the data are exceptionally noisy, the best procedure is to start the line on the last data point and seek the region of highest density of points to the left of it."

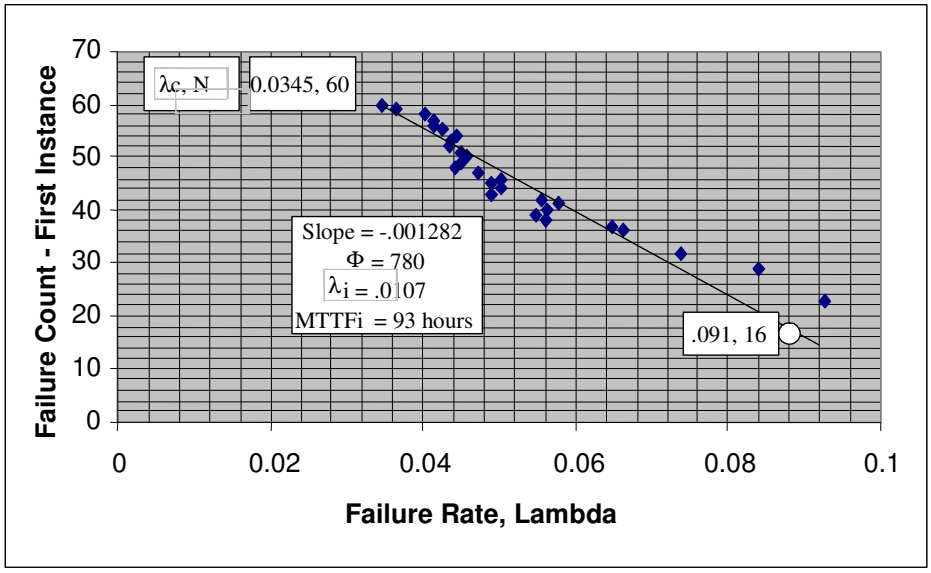

Figure 5: Final Data Plot

In the case of the plot above, Figure 5, significant error would be introduced by using the conventional approach of using a least squares fit, too conveniently available on most spreadsheets and, for these plots, we seek the "region of highest density of points" to the right, not the left.

4 References

[1] John D. Musa, Anthony Iannino, Kazuhira Okumoto, "Software Reliability," McGraw-Hill, 1987.

[2] Ernest O. Codier, "Reliability Growth in Real Life", Proceedings, 1968 Annual Symposium on Reliability, New York, IEEE, Jan., 1968, pp 458-469.

[3] Martin Trachtenberg, "The Linear Software Reliability Model and Uniform Testing," IEEE Transactions on Reliability, 1985, pp 8-16.

[4] Thomas Downs, "An Approach to the Modeling of Software Testing with Some Applications," IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April, 1985, pp 375-380.

[5] Ann Marie Neufelder, "Ensuring Software Reliability," Marcel Dekker, 1993.

---

* **DAVID DWYER** has worked both as a reliability engineer and as a software engineer while at BAE Systems over the last 32 years. Before that, he was a flight instructor at the University of Illinois for 3 ½ years. He has a B.S. in Physics (Providence College, 1963), M.S. in Electrical Engineering (Northeastern University, 1980), and M.S. in Computer Science (Rivier College, 1999). He has presented several papers on Software Reliability at different conferences (including IEEE Annual Reliability & Maintainability Symposia [RAMS] since 1987). The current paper, "Software Reliability Estimations/Projections, Cumulative & Instantaneous" is based on David's project completed during his studies at Rivier College. David presented this paper at the IEEE RAMS Symposium in 2004.