International Journal of
Science and Engineering Investigations vol. 2, issue 12, January 2013
ISSN: 2251-8843

Different Issues in Predicting the Software Reliability

Bonthu Kotaiah[1], R. A. Khan[2]
[1] Research Scholar, Babasaheb Bhimrao Ambedkar University, Lucknow, India
[2] Associative Professor, Babasaheb Bhimrao Ambedkar University, Lucknow, India
([1]kotaiah_bonthuklce@yahoo.com, [2]khanraees@yahoo.com)

Abstract- This paper mainly concentrates on the different methods of software reliability assessment and prediction. We are also shown systematically and scientifically that the software reliability levels assessed are proved by facts and are relatively modest.

Keywords-*Software reliability, reliability prediction, SRGM.*

I. Introduction

Maintaining the safety in the complex and critical software systems involved an analysis of advantages and disadvantages related to the safety and reliability failures. Ex: safety of nuclear power stations. Freshness, complexity, criticalness become the threats to the safety and the reliability of the software systems. Software Reliability requirements will heavily depend upon a careful analysis of the general domain of which the developed software is one part. In a safety and complex system, like nuclear reactor protection system, the necessity can be expressed as a probability of software failure on demand(in the case of the software-based Sizewell B Primary Protection System, the requirement was generally 10^{-3} *pfd).* In a continuous control system, for example in aircraft flight control, the reliability requirement might be expressed as a failure rate (e.g. 10^{-9} probability of failure per flight hour). Finally it may be helpful to mention the details of different misconceptions about the characteristics of software reliability generally by using a probabilistic approach.

II. The Characteristics of The Software Failure Process

There is a major differences in reliability engineering between random failures (generally hardware failures) and systematic failures (e.g. faults coming out of the design problems, generally not coming from software developed) is understand by the researchers in bad manner. The term systematic here indicates the fault mechanism.

There is common uncertainty in the process of the operational environment of the developed software. Importantly, there is imprecise about when a member of the set of input cases that trigger a particular fault will next occur. Thus there is no clear idea about when the next failure of the program or process will occur. This applies the usage of probabilistic indicators of reliability even for 'systematic' failures of the developed software. The most significant instruction is that the failure processes are not deterministic for either 'systematic' faults or for random faults. The same probabilistic measures of reliability are used in both types of faults (The probability models for evaluating software reliability differ from the models used for hardware faults [Lyu 1996]).

III. Reliability Growth Assessment and Prediction

Software Reliability Growth Models (SRGMs) are statistics based methods that perform the direct assessment of the reliability and authenticity of a software product from the observations of its actual failure processes during operation and maintenance. In simple terms, we can say that when a software failure happens there will be a scope to recognize and delete the SDLC design fault which caused the later failures, where the software is modified for further execution again, which leads finally to fail once again. The successive times of failure-free working of the software act as input to statistical models, which then uses the input data to assess the present reliability of the program under observation, and to assume how the software reliability will vary in the near future.

There are so many software reliability growth models exist, which are the detailed probability reliability assessment models to specify probabilistic software failure process. But there is no single and unique model that can assess the software reliability with accurate results in all circumstances. With the last 10 years research results, this problem has been overcome by the different methods for analyzing the predictive accuracy of different software reliability growth models on a particular source of failure data [Abdel-Ghaly, Chan et al. 1986; Brocklehurst and Littlewood 1992; Lyu 1996]. The final results are that we can now use and apply many of the available software reliability growth models(SRGMS) with the failure data coming from a particular software product, and eventually we can learn which (if any) of the different predictions of the software reliability can be trusted.

With the limitations specified above with the statistically representative test cases for the prediction of the software reliability, it is now possible to obtain correct software reliability estimations in different cases and most significantly to know when particular predictions can be understood and

trusted. But it is clear that such software reliability growth models are really only sufficient for the retrieval of the very relevant reliability goals. This can be seen by considering the following examples.

Table 1 shows a simple and straightforward analysis of limited software failure data from the testing and debugging of a command and control system of US Army, using a particular software reliability growth model. The question 'how the developed program is reliable now?' Is answered immediately following the 40th, 50th, . . , 130th failures, in the form, (in this case) of an average time to next failures. Alongside the mttf in the table is the total execution time on test that was needed to achieve that estimated mttf. We can say that, the mttf of this system and the reliability of the system improves as the testing of the program advances. However, the last column shows a clear law of diminishing returns: further modifications in the mttf need more software testing process.

TABLE I. An illustration of the law of diminishing returns in heroic debugging or defect removal of the software. Here the total execution time of the program (in seconds) is required to get a required mean time to software failures is compared and evaluated with the mean itself.

SAMPLE SIZE, i	ELAPSED TIME, ti	ACHIEVED mttf, mi	ti/mi
40	6380	288.8	22.1
50	10089	375.0	26.9
r60	12560	392.5	32.0
70	16186	437.5	37.0
80	20567	490.4	41.9
90	29361	617.3	47.7
100	42015	776.3	54.1
110	49416	841.6	58.7
120	56485	896.4	63.0
130	74364	1054.1	70.1

This is a single proof that involves the use of a specific measure or attribute of software reliability (mttf-mean time to failure), and uses a specific model to carry out the calculations, and a specific developed program under the observation for reliability assessment. Also, same kind of the reliability results are observed throughout all the phases of SDLC with different data sources and for different software reliability growth models (SRGMS). Figure 1 shows an analysis of software failure data from a system in operational use by the end user, for which software and hardware design modifications were being introduced as a result of the software failures. Here the current rate of occurrence of software failures (ROCOF) is calculated at different times, using different software reliability growth models (SRGMs) from that used in Table 1. The dotted line is fitted manually to give a visual impression of what, again, seems to be a very clear law of diminishing returns. Once again, the level of software reliability reached here is quite modest: about 10-2 failures per hour of operational use of the software, which is several times of magnitude, which is in short form we could call 'ultra-high dependability' (compare it with the 10-9 per hour requirement of the civil aircraft flight control systems). Most importantly, it is by no means clear about how the details of the future software reliability growth models of this system will look like. For example, it is not clear to what the curve is asymptotic: could one expect that eventually the ROCOF will approach zero, or not.

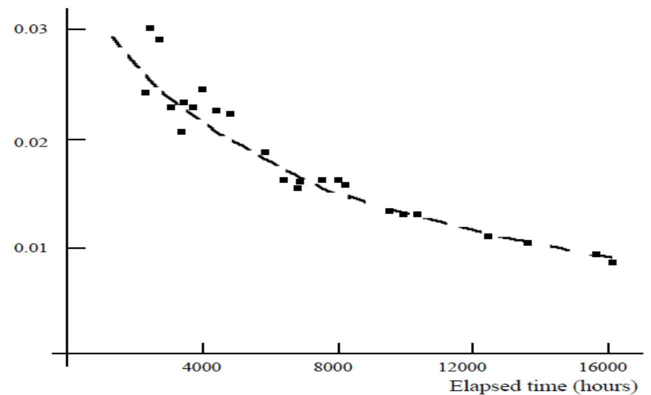

Figure 1. Estimates of the rate of occurrence of failures for a system experiencing failures due to software faults and hardware design faults. The broken line here is fitted by eye.

A program begins the life with a very limited number of faults and run time errors, and these will be happened randomly during operation of the developed program. Different software faults will leads to the overall unreliability of the program in different ways: some faults are 'larger' than others. 'Large' here means that the rate at which the fault would show itself for the effect of the program in execution is large: different faults have different rates of occurrence in the developed program. Table 2 specifies an example of this issue depending upon a large database of problem reports for some large IBM software systems that were developed already [Adams 1984].

TABLE II. Data from [Adams 1984]

	Rate Class							
	1	2	3	4	5	6	7	8
	Mean time to occurrence in kmonths for rate class							
	60	19	6	1.9	.6	.19	.06	.019
Product	Estimated percentage of faults in rate class							
1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	0.7
2	34.3	29.0	18.2	9.7	4.5	3.2	1.5	0.7
3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	0.4
4	34.2	28.5	18.7	11.9	4.4	2.0	0.3	0.1
5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	0.7
6	32.0	28.2	20.1	11.5	5.0	2.1	0.8	0.3
7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	0.6
8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
9	31.2	27.6	20.4	12.8	5.6	1.9	0.5	0.0

Table 2. Data from [Adams 1984], showing the very great difference in ‘sizes’ of software faults. Here size means the mean time to discover a software fault. Adams classified the faults into 8 classes as per their sizes, and the notable aspect of the above figures is the very large differences between the ‘largest’ and the ‘smallest’. Perhaps most startling is that about one third of faults fall into the 60 kmonth class. i.e. a fault from this class would only be seen at the rate of about once every 5000 years.

During software reliability growth we assume that a fix or correction is performed at every point of failure. Let us assume for simplicity that each fix attempt is successful. As debugging continuously progresses, there will be a scope for a software fault with a larger occurrence rate to show itself before a fault with a smaller occurrence rate in the program: more clearly, for any time t, the probability that fault A reveals itself during time t will be smaller than the probability that B reveals itself during t, if the rate of A is smaller than the rate of B. In other ways, we can say that the major faults will be removed before that the minor software faults. As debugging progresses and the program becomes more reliable day by day, it becomes harder to find the inner software faults effectively(because the failure rate of the program is becoming smaller and smaller), and the modifications of the software reliability resulting from these fault-removals are also becoming smaller and smaller successfully in a very less span of time.

IV. Indirect Ways of Evaluating Software Reliability

There are specific critical and most important limitations to the levels of software reliability that can be explained by using the direct software reliability evaluation approach, based on statistical analysis and reporting of operational software testing data. This problem we can overcome by using some other ways of the software reliability assessment. They are: claims based on quality of production, fault tolerance.

A. Reliability Claims Based on process quality

Even though we use very good design and development strategies for the efficient and effective development of the software in most of the software industries for getting the high reliability levels, but we fails to get the levels at acceptable stage. For example, in [RTCA 1992] there is the statement:

‘ . . . Techniques for estimating the post-verification probabilities of software errors were examined and verified. The goal was to implement numerical requirements for such probabilities for digital computer-based equipment and systems certification. But the conclusion was that presently available procedures do not produce the results in which acceptable level of confidence can be achieved for this requirement. Accordingly, this document does not state post-verification software error requirements in these terms.’

We can assume that the procedures and practices that are recommended in the document RTCA 1992 are enough to justify software reliability claims at the required levels irrespective of the necessity for direct measurement of the software reliability parameters.

The problem in finding out any reliability level for a developed program comes from evidence of the quality of the software development process came out from two sources. In the first place, there is small empirical evidence available of the operational reliability for software developed using particular development processes. Secondly, even if extensive evidence were available for a particular process, it would merely concern the software reliability that might be expected on average. The actual achieved reliabilities of the software would vary from one development to another in the form of programming languages, and thus there would be uncertainty involved in any reliability claim for a new software product to be developed and deployed in the future.

B. Fault tolerance based on design diversity

In hardware reliability engineering it is opined that the stochastic software failure processes of the different components in a parallel configuration are independent from one another. It is then easy to tell that a system of high reliability can be built by using unreliable components in many ways. If software versions were developed ‘independently’ of one another, then the version failures were also statistically independent, it would be possible to make claims for very high reliability. But, practicals show that design-diverse versions do not fail independently of one another as per [Knight and Leveson 1986b]. One reason for this is simply that the designers of the different versions tend to make similar mistakes. Another reason is more subtle [Eckhardt and Lee 1985; Littlewood and Miller 1989]: even if the different versions really are independent objects, they will still fail dependently as a result of change of the criticalness of the hardware problem from one input case to another. Put simply, the failure of version A on a particular input suggests that this is a ‘difficult’ input, and thus the probability that version B will also fail on the same input is greater than it otherwise would be.

On the other hand, the experiments [Anderson, Barrett et al. 1985; Knight and Leveson 1986a] find out that fault tolerance brings some improvement in reliability compared with single versions.

C. Formal verification

It is an attempt to move away from probability-based assessments of software reliability towards deterministic, logical claims for complete perfection for the software reliability. Thus if a formal specification of the domain problem could be trusted, a proof that the developed program truly implemented that formal specification and it is guarantee that no failures could arise with design faults in the implementation of the software. It could be said that the product was ‘perfectly reliable’ with respect to such a class of failures. It has limitations.

In the first place, proofs are subject to error. This might be direct human error, in proofs by hand or with the aid of semiautomatic tools, and/or error by the proof-producing

software in machine proof. Importantly, one could assign a failure probability to proofs, and use this in a probabilistic analysis. Anyway, such probabilities would be difficult to incorporate into a dependability evaluation of the software versions: if the proof were erroneous, then the true failure rate will be confused to estimate.

There are also practical problems. Although computerized approaches to formal verification progressed, there are still strict limits to the size and complexity of problems that can be addressed in this way. Finally, it has to be said that it is often unreasonable to assume that the formal specification really captures the more informal engineering requirements other than the other software reliability assessment methods. If the formal specification is wrong, a proof that the implementation of the program conforms to the specification will not guarantee that failures cannot occur.

V. SUMMARY AND CONCLUSIONS

This paper concentrated on the problem of assessing the reliability of software before it is deployed in a safety-critical system. Thus direct evaluation of reliability, using statistical methods based upon (real or simulated) operational data will allow quite modest claims to be made - putting it another way, to make claims for ultra-high reliability this way would require infeasibly large amounts of operational exposure of the developed software. There are some indirect methods to answer this problem.

Methods for evaluating reliability by combining evidence from different sources are helpful, but much better understanding is needed of the complex interdependencies between different types of evidence before these approaches can be trusted for safety-critical systems software reliability assessment. The question of how much it is reasonable to trust the judgment of experts leads to repay further study for these methods. (See [Henrion and Fischhoff 1986] for an interesting - and worrying - study showing the tendency for physicists, both individually and as a community, to be overconfident in their judgments.)

It should be noted that the results in this paper concern the limits to the levels of reliability that can be evaluated prior to the deployment of a system and not after the deployment of a system. This do not find out what levels of reliability can be achieved.

REFERENCES

[1] [Abdel-Ghaly, Chan et al. 1986] A.A. Abdel-Ghaly, P.Y. Chan and B. Littlewood, "Evaluation of Competing Software Reliability Predictions," IEEE Trans. on Software Engineering, vol. 12, no. 9, pp.950-967, 1986.

[2] [Adams 1984] E.N. Adams, "Optimizing preventive maintenance of software products," IBM J. of Research and Development, vol. 28, no. 1, pp.2-14, 1984.

[3] [Anderson, Barrett et al. 1985] T. Anderson, P.A. Barrett, D.N. Halliwell and M.R. Moulding. "An Evaluation of Software Fault Tolerance in a Practical System," in Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15), pp. 140-145, Ann Arbor, Mich., 1985.

[4] [Brocklehurst and Littlewood 1992] S. Brocklehurst and B. Littlewood, "New Ways to get Accurate Reliability Measures," IEEE Software, vol. 9, no. 4, pp.34-42, 1992.

[5] [Dyer 1992] M. Dyer. The Cleanroom Approach to Quality Software Development, Software Engineering Practice. New York, John Wiley and Sons, 1992.

[6] [Eckhardt and Lee 1985] D.E. Eckhardt and L.D. Lee, "A Theoretical Basis of Multi version Software Subject to Coincident Errors," IEEE Trans. on Software Engineering, vol. 11, pp.1511-1517, 1985.

[7] [Fenton, Littlewood et al. 1996] N. Fenton, B. Littlewood and M. Neil. "Applying Bayesian belief networks in systems dependability assessment," in Proc. Safety Critical Systems Symposium, pp. 71-94, Leeds, Springer-Verlag, 1996.

[8] [Henrion and Fischhoff 1986] M. Henrion and B. Fischhoff, "Assessing uncertainty in physical constants," Americal J. of Physics, vol. 54, no. 9, pp.791-798, 1986.