A Study on Software Reliability Engineering Present Paradigms and its Future Considerations

Deepak Pengoria, Saurabh Kumar (MS SE)
School of Computing Sciences, VIT University, Vellore, India
deepakpengoria2005@vit.ac.in saurabh4vit@gmail.com

Abstract-***Software reliability engineering is focused on engineering techniques for developing and maintaining software systems whose reliability can be quantitatively evaluated. In order to estimate as well as to predict the reliability of software systems, failure data need to be properly measured by various means during software development and operational phases. Although software reliability has remained an active research subject over the past 35 years, challenges and open questions still exist. The various modeling technique for Software Reliability is reaching its prosperity, but before using these technique, we must carefully select the appropriate model that can best suit our case. Measurement in software is still in its infancy. No good quantitative methods have been developed to represent Software Reliability without excessive limitations. In this paper, we review the introduction of software reliability engineering, the current trends and existing problems and specific difficulties, possible future directions and promising research subjects in software reliability engineering are also addressed.***

I. INTRODUCTION

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software.

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment [1]. Software reliability engineering (SRE) is therefore defined as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. As a proven technique, SRE has been adopted either as standard or as best current practice by more than 50 organizations in their software projects and reports [2], including AT&T, Lucent, IBM, NASA, Microsoft, and many others in Europe, Asia, and North America. However, this number is still relatively small compared to the large amount of software producers in the world.

Existing SRE techniques suffer from a number of weaknesses. First of all, current SRE techniques collect the failure data during integration testing or system testing phases. Failure data collected during the late testing phase may be too late for fundamental design changes. Secondly, the failure data collected in the in-house testing may be limited, and they may not represent failures that would be uncovered under actual operational environment. This is especially true for high-quality software systems which require extensive and wide-ranging testing. The reliability estimation and prediction using the restricted testing data may cause accuracy problems. Thirdly, current SRE techniques or modeling methods are based on some unrealistic assumptions that make the reliability estimation too optimistic relative to real situations. Of course, the existing software reliability models have had their successes; but every model can find successful cases to justify its existence. Without cross- industry validation, the modeling exercise may become merely of intellectual interest and would not be widely adopted in industry. Thus, although SRE has been around for a while, credible software reliability techniques are still urgently needed, particularly for modern software systems [3].

In the following sections we will discuss the present and the future of software reliability engineering. We first survey what the current trend is and what problems and concerns remain. We propose

the possible future directions in software reliability engineering

II. CURRENT PARADIGMS

The challenges in software reliability not only stem from the size, complexity, difficulty, and novelty of software applications in various domains, but also relate to the knowledge, training, experience and character of the software engineers involved. We address the current paradigms and problems from a number of software reliability engineering aspects.

2.1. Software and system reliability

Previous work has been focused on extending the classical reliability theories from hardware to software, so that by employing familiar mathematical modeling schemes, we can establish software reliability framework consistently from the same viewpoints as hardware. The advantages of such modeling approaches are: (1) The physical meaning of effect of failures on reliability, as measured in the form of failure rates, can be directly applied to the reliability models. (2) The combination of hardware reliability and software reliability to form system reliability models and measures can be provided in a unified theory. (3) System reliability models inherently engage system structure and modular design in block diagrams. The resulting reliability modeling process is not only intuitive (how components contribute to the overall reliability can be visualized), but also informative (reliability-critical components can be quickly identified).

The major drawbacks, while hardware failures may occur independently, software failures do not happen independently. The interdependency of software failures is also very hard to describe in detail or to model precisely. Furthermore, similar hardware systems are developed from similar specifications, and hardware failures, usually caused by hardware defects, are repeatable and predictable. Therefore, while failure mode and effect analysis (FMEA) and failure mode and effect criticality analysis (FMECA) have long been established for hardware systems, they are not very well understood for software systems.

2.2. Software reliability modeling

Among all software reliability models, SRGM is probably one of the most successful techniques in the literature, with more than 100 models existing in one form or another, through hundreds of publications. In practice, however, SRGMs encounter major challenges. First of all, software testers seldom follow the operational profile to test the software, so what is observed during software testing may not be directly extensible for operational use. Secondly, when the number of failures collected in a project is limited, it is hard to make statistically meaningful reliability predictions. Thirdly, some of the assumptions of SRGM are not realistic, e.g., the assumptions that the faults are independent of each other; that each fault has the same chance to be detected in one class; and that correction of a fault never introduces new faults [4].

Although some historical SRGMs have been widely adopted to predict software reliability, researchers believe they can further improve the prediction accuracy of these models by adding other important factors which affect the final software quality [5,6,7]. Among others, code coverage is a metric commonly engaged by software testers, as it indicates how completely a test set executes a software system under test, therefore influencing the resulting reliability measure. To incorporate the effect of code coverage on reliability in the traditional software reliability models, [5] proposes a technique using both time and code coverage measurement for reliability prediction. It reduces the execution time by a parameterized factor when the test case neither increases code coverage nor causes a failure. These models, known as adjusted Non-Homogeneous Poisson Process (NHPP) models, have been shown empirically to achieve more accurate predictions than the original ones.

In the literature, several models have been proposed to determine the relationship between the number of failures/faults and the test coverage achieved, with various distributions. [7] suggests that this relation is a variant of the Rayleigh distribution, while [6] shows that it can be expressed as a logarithmic-exponential formula, based on the assumption that both fault coverage and test coverage follow the logarithmic NHPP growth model with respect to the execution time. More metrics can be incorporated to further explore this new modeling avenue.

2.3. Metrics and measurements

As software complexity and software quality are highly related to software reliability, the measurements of software complexity and quality attributes have been explored for early prediction of software reliability [8]. Static as well as dynamic program complexity measurements have been

collected, such as lines of code, number of operators, relative program complexity, functional complexity, operational complexity, and so on. The complexity metrics can be further included in software reliability models for early reliability prediction, for example, to predict the initial software fault density and failure rate.

In SRGM, the two measurements related to reliability are: 1) the number of failures in a time period; and 2) time between failures. One key problem about software metrics and measurements is that they are not consistently defined and interpreted, again due to the lack of physical attributes of software. The achieved reliability measures may differ for different applications, yielding inconclusive results. A unified ontology to identify, describe, incorporate and understand reliability-related software metrics is therefore urgently needed.

2.4. Testing effectiveness and code coverage

As a typical mechanism for fault removal in software reliability engineering, software testing has been widely practiced in industry for quality assurance and reliability improvement. Effective testing is defined as uncovering of most if not all detectable faults. As the total number of inherent faults is not known, testing effectiveness is usually represented by a measurable testing index. Code coverage, as an indicator to show how thoroughly software has been stressed, has been proposed and is widely employed to represent fault coverage. The problem is testing result for published data did not support a causal dependency between code coverage and defect coverage.

A normalized reliability growth curve is used to predict to verify whether software reliability is attained. It can be used to determine when to stop testing. It can also be used to demonstrate the impact on reliability of a decision to deliver the software on an arbitrary date. Figure 1 illustrates what may happen when the process for fixing detected errors is not under control, or a major shift in design has occurred as a result of failures detected. Figure 2 overlays the desired outcome from applying software reliability engineering principles on a typical reliability curve. The goal is to converge quickly to the desired reliability goal thus reducing the rework and the time and resources required for testing and enabling a shorter time to delivery without sacrificing reliability

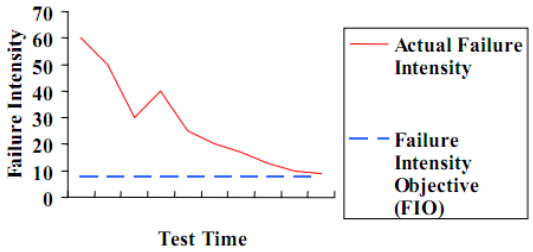

Fig 1: Reliability Growth Curve Reflecting Out-of-Control Process.

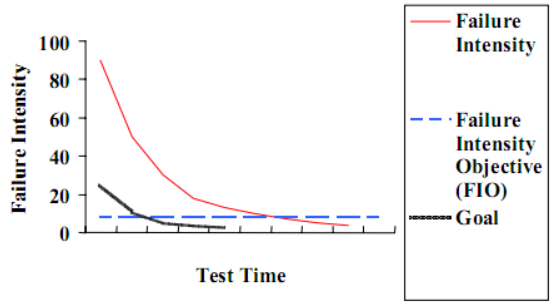

Fig 2: Reliability Growth Curve Desired v/s Typical.

2.5. Industry practice and concerns

Software practitioners often see reliability as a cost rather than a value, an investment rather than a return. Often the reliability attribute of a product takes less priority than its functionality or innovation. The main reason for the lack of industry enthusiasm in SRE is because its cost-effectiveness is not clear. Current SRE techniques incur visible overhead but yield invisible benefits. In contrast, a company's target is to have visible benefit but invisible overhead.

As the competitive advantage of product reliability is less obvious than that of other product quality attributes (such as performance or usability), few practitioners are willing to try out emerging techniques on SRE. The fact that there are so many software reliability models to choose from also intimidates practitioners. So instead of investigating which models are suitable for their environments or which model selection criteria can be applied, practitioners tend to simply take reliability measurements casually, and they are often suspicious about the reliability numbers obtained by the models. Many software projects claim to set reliability objectives such as five 9's or six 9's (meaning 0.99999 to 0.999999 availability or 10^{-5} to 10^{-6} failures per execution hour), but few can validate their reliability achievement.

III. POSSIBLE FUTURE DIRECTIONS

Software Reliability Engineering relates to whole software life cycle. We discuss possible future directions with respect to four areas: software architecture, testing and metrics.

3.1 Reliability for software architectures and off-the-shelf components

Due to the ever-increasing complexity of software systems, modern software is seldom built from scratch. Revolutionary and evolutionary object-oriented design and programming paradigms have vigorously pushed software reuse. In the light of this shift, reliability engineering for software development is focusing on two major aspects: software architecture, and component-based software engineering. The software architecture of a system consists of software components, their external properties, and their relationships with one another. As software architecture is the foundation of the final software product, the design and management of software architecture is becoming the dominant factor in software reliability engineering research. While software architecture represents the product view of software systems, component-based software engineering addresses the process view of software engineering. In this popular software development technique, many research issues are identified, such as reliability, reusability, clean interface design, fault tolerance etc.

3.2 Testing for reliability assessment

Software testing and software reliability have traditionally belonged to two separate communities. Software testers test software without referring to how software will operate in the field, as often the environment cannot be fully represented in the laboratory. Software reliability measurers insist that software should be tested according to its operational profile in order to allow accurate reliability estimation and prediction. One approach is to measure the test compression factor, which is defined as the ratio between the mean time between failures during operation and during testing. Another approach is to ascertain how other testing related factors can be incorporated into software reliability modeling, so that accurate measures can be obtained based on the effectiveness of testing efforts.

The effect of code coverage on fault detection may vary under different testing profiles. The correlation between code coverage and fault coverage should be examined across different testing schemes, including function testing, random testing, normal testing, and exception testing. In other words, white box testing and black box testing should be cross–checked for their effectiveness in exploring faults, and thus yielding increase in reliability. Including linking software testing and reliability with code coverage, statistical learning techniques may offer another promising avenue to explore. In particular, statistical debugging approaches [9, 10], whose original purpose was to identify software faults with probabilistic modeling of program predicates, can provide a fine quantitative assessment of program codes with respect to software faults. They can therefore help to establish accurate software reliability prediction models based on program structures under testing.

3.3 Metrics for reliability prediction

Today companies must collect software metrics as an indication of a maturing software development process. Industrial software engineering data, particularly those related to system failures, are historically hard to obtain across a range of organizations. Novel methods are used to improve reliability prediction are actively being researched. For example, by extracting rich information from metrics data using a sound statistical and probability foundation, Moreover, traditional reliability models can be enhanced to incorporate some testing completeness or effectiveness metrics, such as code coverage, as well as their traditional testing-time based metrics. The key idea is that failure detection is not only related to the time that the software is under testing, but also what fraction of the code has been executed by the testing.

One drawback of the current metrics and data collection process is that it is a one-way, open-loop avenue: while metrics of the development process can indicate or predict the outcome quality, such as the reliability, of the resulting product, they often cannot provide feedback to the process regarding how to make improvement. In the future, a reverse function is urgently called for: given a reliability goal, what should the reliability process (and the resulting metrics) look like? By providing such feedback, it is expected that a closed-loop software reliability engineering process can be informative as well as beneficial in achieving predictably reliable software.

IV. CONCLUSION

As the cost of software application failures grows and as these failures increasingly impact business performance, software reliability will become progressively more important. Employing effective software reliability engineering techniques to improve product and process reliability would be the industry's best interests as well as major challenges. In this paper, we have reviewed the current trends and existing problems and specific difficulties. Possible future directions and promising research problems in software reliability engineering have also been addressed. We have laid out the current and possible future trends for software reliability engineering in terms of meeting industry and customer needs. In particular, we have identified new software reliability engineering paradigms by taking software architectures, testing techniques, and software failure manifestation mechanisms into consideration.

ACKNOWLEDGEMENT

The authors would like to thank the reviewers who provided many constructive comments for us to improve the quality of this paper. The author would like to thank Prof H.R. Vishwakarma for assignment of linguist variables and guidance.

REFERENCES

[1] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, *STD-729-1991, ANSI/IEEE*, 1991.

[2] J.D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper (2nd Edition),* AuthorHouse, 2004

[3] B. Littlewood and L. Strigini, "Software Reliability and Dependability: A Roadmap," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE'2000),* Limerick, June 2000, pp. 177-188.

[4] M.L. Shooman, *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design*, Wiley, New York, 2002

[5] M. Chen, M.R. Lyu, and E. Wong, "Effect of Code Coverage on Software Reliability Measurement," *IEEE Transactions on Reliability*, vol. 50, no. 2, June 2001, pp.165-170.

[6] Y.K. Malaiya, N. Li, J.M. Bieman, and R. Karcich, "Software Reliability Growth with Test Coverage," IEEE *Transactions on Reliability*, vol. 51, no. 4, December 2002, pp. 420-426

[7] M.A. Vouk, "Using Reliability Models During Testing With Nonoperational Profiles," in *Proceedings of 2nd Bellcore/Purdue Workshop on Issues in Software Reliability Estimation*, October 1992, pp. 103-111.

[8] Rome Laboratory (RL), *Methodology for Software Reliability Prediction and Assessment*, Technical Report RL- TR-92-52, volumes 1 and 2, 1992.

[9] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical Debugging: A Hypothesis Testing-based Approach," *IEEE Transaction on Software Engineering*, vol. 32, no. 10, October, 2006, pp. 831-848.

[10] A.X. Zheng, M.I. Jordan, B. Libit, M. Naik, and A. Aiken, "Statistical Debugging: Simultaneous Identification of Multiple Bugs," in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006, pp. 1105-1112.