New Paradigm for Software Reliability Estimation

Ritika Wason
PhD Scholar,
Dept. of Computer Science and Engineering,
Sharda University,

P. Ahmed
Professor,
Dept. of Computer Science and Engineering,
Sharda University,

M.Qasim Rafiq
Professor,
Dept. of Computer Engineering,
Aligarh Muslim University,Aligarh

ABSTRACT
In common parlance, the traditional software reliability estimation methods often rely on assumptions like statistical distributions that are often dubious and unrealistic. This paper analyzes the assumptions of traditional reliability estimation methods and further evaluates the practical viability of the predictions offered by these models in the current scenario. We further propose a novel Finite Automata (FA) based reliability model that implicitly scores over the traditional models on many factors, most importantly due to the fact that it is based on the realistic assumption that a software system in execution is a Finite State Machine (FSM).

General Terms
Automata- Based Software, Software Reliability, Software Reliability Growth Model (SRGM).

Keywords
Software Reliability, Software Reliability Growth Model (SRGM), Finite-State Machine (FSM), Finite State Automata, Automata-Based Software Reliability Model.

1. INTRODUCTION
The fundamental goal of all software design and engineering is the production of dependable systems. Requirements Elicitation, Software Design, Implementation, Software Testing, Operation and Maintenance phases of any software system are expected to aid in the same. During the above software life cycle phases, software reliability is generally estimated during the software testing phase as it involves assimilating failure data that can be fitted to the mathematical expression of the reliability model. This evaluation of the correctness of the expected functionalities of the software, results in software reliability estimates which can be defined as the probability of failure-free software action for a particular period of time in a specific environment.

Many different software reliability models have been suggested for reliability prediction, estimation and measurement of real-time software. However, posterior black-box approaches remain the prevalent approaches for such modeling as they are based on post-implementation data regarding the interactions of the software with its operational environment. The inaccuracy of the traditional approaches does not require proof, as despite establishment of software reliability as an important quality characteristic of software, software failures continues to proliferate both in number and effects. Hence, software reliability continues to haunt developers even today. An effective software reliability prediction, estimation and measurement model could play an important role in managing the risks posed by unreliable software systems [17]. The above realization sufficiently justifies the need for an efficient and dependable software reliability estimation process that should be supplemented using different formal verification techniques like state-based models, model checking [10], [4] or finite state machines [15], [18],[21] which continue to be deployed in realization of processes that claim to provide an accurate, efficient and dependable reliability estimate.

Section 2 of this paper critically examines and evaluates the practicality of the existing reliability estimation models and processes. The evaluation further justifies the need for an innovative approach for reliability estimation as proposed in Section 3. Section 4 examines how attributes like intelligence and self-learning can further strengthen the reliability estimation model proposed in Section 3. Section 5 discusses how the proposed model shall expand the scope of the traditional reliability estimation models from simple reliability measurement and quantification to thorough software evolution through overall quality improvement.

2. SOFTWARE RELIABILITY: LIMITATIONS AND CHALLENGES
Software Reliability modeling is largely influenced by hardware reliability modeling which was well-established by the time efforts for estimating software reliability started. However, it is well-established that hardware reliability is an incorrect foundation for software reliability as software is not the same as hardware. Critiques [11], [12], [13] of software reliability models analyzed and pointed out the unrealistic assumptions of these models as early as late 1970s and early 1980s when the term software implied just a computer program. However, this criticism was never given due attention and reliability engineers and researchers continued to churn out one reliability model after another based on the same assumptions to determine the same set of parameters. This ignorance of important details created the reliability challenge of software we know of today. Conversely in present times when the software engineering economics has transformed and reliability of software systems is no longer a negotiable option we can no longer afford to follow such erroneous models and practices.

Software Reliability estimation became a dynamic research domain since the early 1970s [24]. Since then, many diverse reliability estimation models have been developed and implemented for different commercial applications employing different software systems deployed under varying operational

environments. However, even after dozens of reliability estimation models and decades of research the software industry still continues to suffer what we would term as the **reliability challenge**, due to which there still exists no single known model that can be applied under all contexts or even a model that can be repeatedly applied to the same software under different operational environments.

In order to better evaluate the above the challenge, this section critically examines the underlying assumptions of software reliability and the different models used for its estimation. Comparative research [11], [12], [13], [14] on traditional reliability estimation models reveals that the cause for the so-called reliability challenge is rooted in the dubious and unrealistic assumptions that form the basis of all the traditional models. To further support the claim, Table 1 below classifies some of the popular reliability estimation models along with their underlying assumptions and parameters used to predict system reliability.

A careful examination of the assumptions listed in Table 1 reveals that the cause for the inaccuracy of the predictions of traditional reliability estimation models is mainly due to their unrealistic nature and absence of mathematical implementation. All traditional software reliability growth models use system test data fitted to some distribution to predict the number of defects remaining in the software. However, real-time data was never actually fitted to these distributions or the distributions were never actually estimated. Further, the efficacy of each of these models is directly related to their analytical ability which implies that the number of residual defects predicted by the model should be same as the actual number found in field use [23]. Conversely under real-time operation this is never the case and hence the major reason for the inaccurate estimates by the traditional models. Having understood the underlying cause of the inaccuracy we now analyze the foundations for faulty predictions by the traditional models. The term software reliability quantifies our confidence in the ability of software to provide acceptable levels of performance under a given operational environment [25]. The inherent probabilistic nature of the term itself is a source of headache for the software designers and developers. Software performance under a given operational environment can be influenced by a large number of internal and environmental factors like schedule pressure, unstructured development practices, resource limitations, volatile and evolutionary user requirements, interdependence among modules etc. All the above factors can negatively impact software reliability estimation and measurement. Further it also becomes difficult to estimate whether the software being implemented is as reliable as predicted or not until the software is actually implemented. The above reliability estimation problem stems from the fact that we generally estimate the reliability of a software component during the testing phase on the assumption that its behavior during real-time execution is similar to the testing times when the failure data was actually collected. However, a hard to ignore fact that overrules the above assumption is that testing is always limited by the lack of realistic inputs and hence this is never the case in real-life.

All software reliability models make their own set of assumptions about testing and defect repair. However, many of these assumptions are questionable in the current scenario as they completely contrast the actual practice. Table 2 lists some common assumptions of traditional models and compares them with their corresponding real-time notions.

In times when software systems are a part and parcel of human life, unreliability of software becomes unbearable. Despite many different Software Reliability Growth Models (SRGMs) and practices [1], [2], [7], [8], [9], [11], [16] fact remains that software reliability still remains a dark grey area of software engineering. Table 2 clarifies that the underlying problem with all reliability estimation models is that all of them suffer under their own unrealistic foundations and assumptions like statistical distributions that are dubious themselves. A major misinterpretation being that though all models agree that reliability is the absence of failures, they quantify reliability using some kind of failure data (brute force). Also all reliability techniques can be classified as either a priori technique (build the software right) or a posteriori techniques (right the wrongs). Much of the current practice today is in a posteriori techniques. We build software that's not very good and through brute force, debug it into correctness [3]. By shifting some of the balance towards a priori efforts; we can go a long way towards correcting some of the most serious problems.

Table 1: Classification of Traditional Reliability Mode

S. No	Category	Class	Example	Assumptions	Parameters
1.	**Time-Domain Models [8]** **Basis:** Observed failure history used to estimate residual faults and fault-detection time. **Practice:** Model the underlying failure process of the software under consideration and use the observed failure history as a guideline to estimate the residual number of faults in the software. **Limitation:** Underestimate the number of remaining errors.	Time Between Failure/ Deterministic Models/ Homogeneous Markov Models [8],[14],[25]	Jelinski- Moranda De-eutrophication Model (1972)	i. Fixed, unknown number of independent, initial faults in software. ii. Instantaneous and perfect repair process.	$\mu(t) = N(1-\exp(-\varphi t)$ $\lambda(t) = N\Phi\exp(-\Phi t)$
			Goel-Okumoto Imperfect Debugging Model (1978)	i. Independent times between failures ii. Equal probability of each fault exposure iii. Embedded faults independent of each other iv. Debugging and perfect fault removal after each occurrence. v. Allow for imperfect debugging	$\mu(t) = (1/\theta)[\ln(\lambda_0\theta_T+1)]$ $\lambda(t) = \Phi\ [\ N- p(i-1)]$
		Other Models	Littlewood-Verrall Bayesian Model(1981)	i. Similar to Jelinski-Moranda Model, only assumes that different sized faults may contribute unequally to failures. ii. Larger Sized Faults detected and fixed earlier.	$\Theta(i) = t_i + \psi(i) / \alpha$ $\lambda(t) = \omega_0 e^{-\omega_1 t}$
		Fault Count Models/Non-Homogeneous Markov Models [8],[14],[25]	Goel-Okumoto NHPP Model	i. Random number of faults in software. ii. Independent testing intervals. iii. Homogeneous testing during intervals iv. Number of faults detected per interval independent of each other.	$\mu(t) = a(1-e^{-bt})$ $\lambda(t) = abe^{-bt}$
			Musa's Execution Time Model (1975)	i. Explicitly emphasizes the dependence of hazard function on execution time. ii. Finite failures experienced in infinite time.	$\mu(t) = V_0(1-\exp(\lambda_0 t)/V_0)$ $\lambda(t) = \lambda_0\exp(-\lambda_0 t)/ V_0$
2.	**Data Domain Models [8]** **Basis:** Reliability estimate of a system by exercising all input combinations. **Practice:** Reliability Estimation through Sample data set. **Limitation:** Full input prediction impossible.	**Fault Seeding Models** [8],[14],[25]	Mills Hypergeometric Model	i. Software products with unknown number of faults seeded with known number of faults and tested. ii. Estimation of actual number of indigenous faults obtained through ratio of discovered seeded faults and discovered actual faults.	$\text{Var}(R) = (f+a)(s+b)/(f+s+a+b)^2(f+s+a+b+1)$
		Input Domain Based Model [8],[14],[25]	Nelson Model;	i. Random input testing. ii. Reliability estimation through ratio of discovered seeded and actual faults.	$\mu(t) = E[N(t)]$ for all $t>=0$ OR $MTBF=1/f\ \Sigma\ t_i$

From the above discussion we can now conclude that all the above factors converge to a common point, namely the need for realistic modeling and accurate quantification of the software development process [23].

Despite numerous reliability estimation models and decades of ongoing reliability research all the current software reliability literature is still inconclusive about the fact that which models and techniques are best for reliability estimation [23]. Hence the need of hour is the development a strong theoretical automata-based reliability model that can be mathematically verified and has its roots in state-based approach.

The main reason for suggesting an automata-based reliability model as the model for accurate reliability estimations and

self-learning failure conditions is motivated by the fact that a Finite State Machine can be considered as a mathematically defined object that can provide structured and precise understanding of what is going on in systems represented as complex state machines. The major advantage of this formal model for software system representation is the fact that any system can be easily represented as a control flow graph consisting of a number of states and transitions which may further result in some particular states.

Table 2: Comparison of Key Assumptions of Software Reliability Models versus Reality

S. No	Assumption	Reality
1.	Immediate defect repair.	Defects are not repaired immediately.
2.	Perfect repair process.	Defect repair introduces new defects. (imperfect debugging)
3.	No new code is introduced during testing.	New code may be introduced both during debugging as well as extensions to the software.
4.	Independent times between failures.	The assumption is not universally true as test cases are not always chosen randomly.
5.	Testing is representative of the operational usage.	Software behavior during testing is widely distinct from software behavior during actual operation.
6.	Number of bugs in the program is itself a measure of unreliability.	Program with more bugs in relatively unexercised portions of code will be more reliable than a program with less, frequently encountered bugs.
7.	Equal probability of each fault exposure.	Unequal probabilities of different size fault occurrences.
8.	Input profile distribution is known.	Complete input profile distribution cannot be known.

3. PROPOSED FRAMEWORK FOR AUTOMATA-BASED RELIABILITY MODEL

All software reliability growth models are approximations of the real testing process, thus none of the models can be regarded to be perfect [2]. The traditional reliability models and their underlying assumptions as discussed in the previous section are in no way exhaustive and complete. However, this sample set is enough to establish that none of the current software reliability estimation models can individually suffice to provide the breakthrough that the field of software reliability requires today. To accurately estimate the reliability of critical business applications in the present day we need a realistic, mathematically sound model of reliability estimation. To realize such a model we first need to look for alternate approaches for reliability estimation which can replace the current black box approaches that form the basis of all existing models.

A natural, realistic and mathematically sound replacement to the prevalent black box approaches is the use of structured models [2]. Reliability and availability of a software system can easily be estimated using models of system structure along with failure data regarding the same. The simple philosophy that drives this proposed reliability measurement model is the fact that if we know how the program behaves for every possible input by identifying all the states that a software can acquire based on user inputs, then the reliability of the complete software can easily be estimated as the sum of its component state reliabilities at any point of time during its life cycle [5]. The above approach shall implicitly estimate system reliability in an accurate fashion as it considers the actual system structure of the software instead of failure data.

To realize the above reliability model we further suggest that a novel Finite Automata (FA) based approach based on the realistic assumption that a software system in execution is a Finite State Machine (FSM) is the best choice. The suitability of the proposed FSM approach is further established due to its strong mathematical foundations in the theory of automata.

The proposed Finite Automata-based reliability estimation model scores over the traditional models due to its inherent characteristics. Firstly, the model is based on the realistic assumption that a software system in execution is a Finite State Machine (FSM). Secondly, the proposed model can be realized as a state-space system that can further be enhanced with efficient and well-established problem solving processes required to handle modern, complex, real-time software systems.

We thus hypothesize that a state-based approach for software representation can be universally applied to software and all its component parts. This mode of software representation will help in easily tracing how a particular piece of code changes the state of software computation and hence results in correct or incorrect system state. To prove the validity of the above hypothesis, we further propose the development of an algorithm and a formal model that can help in guiding and monitoring the design and implementation of a software to control system reliability at any point in its life. The proposed model will help realize an intelligent, self-learning software reliability estimation model that can easily detect system error state, register the particular state and the transition that led to such a state in its memory and never repeat the transition that leads to the particular error state.

To realize the above design we first propose that software should be represented using state-based approach as suggested by Hoare [20]. The basic idea behind the proposed model is depicted through a small illustration in Figure 1, Each software module/ block of software if represented as a group of nodes called the Learning Automata. Each node in the group marked as the Learning Automata should be achievable from the initial node and may result in either the correct state

(output node) or error state (error node). Further if each such node is assigned a probability, then the reliability of the whole cluster can be defined as the sum of probabilities of the correct nodes.

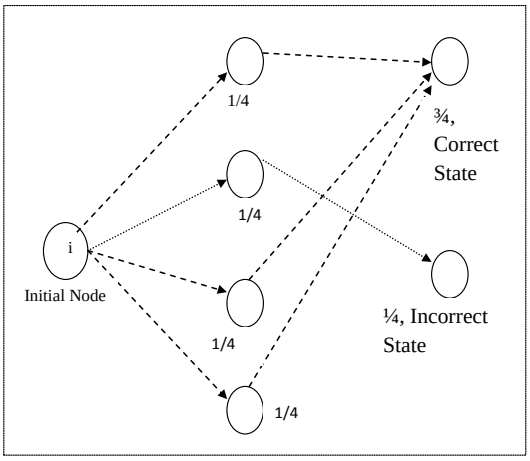

Figure 1: Representation of a Software Block/ Component / Module as a set of distinct states

If every software system is represented as a combination of such mutually interacting nodes, then the reliability of the software can also be estimated through the individual reliabilities of each of these groups.

To realize the proposed model we further outline the component phases required for the same. To initiate, the model shall monitor a software system by parsing it into its component sub-systems which can further be represented as a cluster of inter-connected nodes (State-based Software Representation Phase) depicted in Figure 2. After representation the framework should be able to compute all possible independent paths through the system and also accumulate knowledge regarding which transitions could lead to undesirable failure states (Knowledge-Acquisition Phase). Further, in its knowledge implementation phase the framework should be able to utilize its accumulated knowledge to ensure operationally reliable software at any point of the software life cycle. The various phases discussed above are depicted in Figure 2 below:

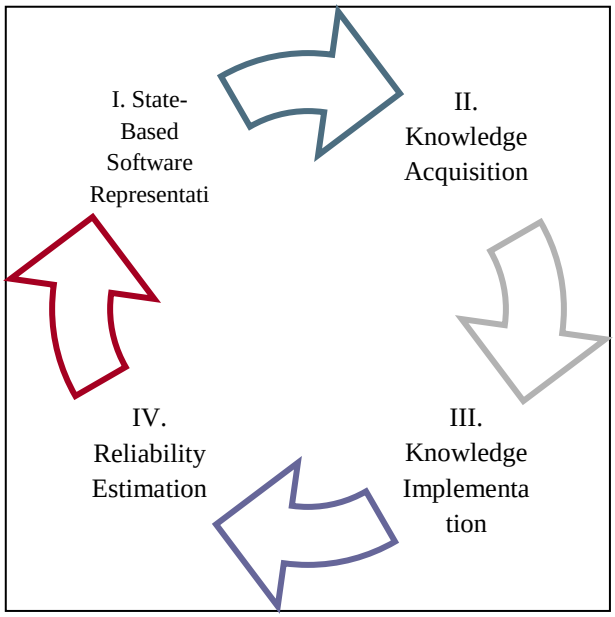

Figure 2: Phases of the Automata-Based Software Reliability Model

Subsequent algorithms for realization of the proposed model are being worked out at the time of this writing and shall be discussed in future work.

4. FUTURE ENHANCEMENTS

Section 3 lays down the basic framework for an automata-based formal reliability model which besides being used for simple reliability predictions can be further trained to perform many other functions. The proposed FSM approach with its strong mathematical foundations in the theory of automata is an ideal choice for a generic, intelligent, self-learning reliability model for complex, ever-evolving software systems. The proposed reliability model can further be extended to function as an intelligent, self-learning and self-correcting model that can dynamically handle failure conditions in real time. The proposed model has its basis in network reliability estimation studies [5] and software reliability estimation models for component-based software systems [22].

The proposed model can also be realized as a state-space system that can further be supported with efficient and well established problem solving processes for embedding intelligence and self-learning capabilities in the proposed reliability model. These capabilities of the automata-based reliability model should prove beneficial in reliability estimation and forecasting of complex, ever-evolving software systems in real-time. We further propose the inculcation of self-healing properties in software system such that whenever system arrives at an error state (actual output differs from expected output) due to any operational or design anomalies, the system should be capable enough to recall its previous correct state and should retreat back to the same in order to resume its operation.

5. CONCLUSION

Inaccurate reliability estimations with the traditional reliability models are no longer an acceptable option. In current times when software rules the mankind and the globe, demand for certified reliable software need to be met. In such a competitive scenario a generic reliability model that can be used for all software systems under all contexts is an essential requirement. The study establishes the fact that traditional software reliability estimation models fail to succeed under all operational contexts due to their dubious and faulty assumptions and misrepresentations of the reality. This paper further ascertains that only a formal automata-based reliability model can be successful in providing accurate reliability estimates for our current and future complex, critical, real-time software systems. However, to realize such a model we require an effective monitoring as well as self-learning model that can learn different states acquired by system components during its life and then apply this knowledge for estimating system reliability at any point on its life or for recovering from a fault.

6. REFERENCES

[1] Faqih, K.M.S. 2009. What is Hampering the Performance of Software Reliability Models? A Literature Review. International MultiConference of Engineers and Computer Scientists.

[2] Chung, D.W. 2007. Quantitative Reliability Assessment for Safety Critical System Software Journal of Electrical Engineering and Technology, Vol 2. No.3, 386-390.

[3] Sharma, V.S. and Trivedi, K.S. 2007. Quantifying software performance, reliability and security: An Architecture-Based Approach the Journal of Systems and Software, vol 80, 493-509.

[4] Dai, Y.S., Marshall, T. and Guan, X. 2006. Autonomic and Dependable Systems: Moving Towards a Model-Driven Approach Journal of Computer Science.

[5] Bowles, J. 1989. A Model for Assessing Computer Network Reliability, IEEE Proceedings.

[6] Chan, H. and Chieu, T. 2003. An approach to monitor application states for self-managing (autonomic) systems. 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications, 312-313.

[7] Dai, Y.S. , Xie M. and Poh, K.L. 2005. Markov renewal models for correlated software failures of multiple types, IEEE Trans. On Reliability, Vol. 54, 100-106.

[8] Gokhale, S.S., Marinos, P.N. and Trivedi, K.S. 1996. Important Milestones in Software Reliability Modeling Communications in Reliability, maintainability and Serviceability, SAE International.

[9] Yang, B. , Li, X., Xie, M. and Tan, F. 2010. A generic data-driven software reliability model with model mining technique, Reliability Engineering and System Safety, Vol. 95, 671-678.

[10] Huth, M. 2007. Some current topics in model checking, Intl. Journal Software Tools Technology Transfer, Vol 9, 25-36.

[11] Littlewood, B. 1879. How to measure Software Reliability and How Not To IEEE Transactions on Reliability, Vol 28 (2), 103-110.

[12] Littlewood, B. 1975. MTBF is meaningless in software reliability, (letter) IEEE Trans. Reliability, vol R-24, 82.

[13] Littlewood, B. 1980. Theories of Software Reliability: How Good Are They and How Can They Be Improved? IEEE Transactions on Software Engineering, SE 6, 489-500.

[14] Goel, A.L. 1985. Software Reliability Models: Assumptions, Limitations and Applicability, IEEE Transactions on Software Engineering, vol SE11, No. 12, 1411-1423.

[15] Li, Juncao 2010. An Automata Theoretic Approach to Hardware/Software Co verification, Doctoral Thesis, Portland State University.

[16] Gokhale, S. Accurate Reliability Prediction Based on Software Structure, http:// www.engr.uconn.edu/ ~ssg/ cse300/ 397-232.pdf.

[17] Fenton, N., Krause, P. and Neil, M. 2002. Software Measurement: Uncertainty and Causal Modeling, IEEE Software, vol. 19(4), 116-122.

[18] Carmely, T. 2010. Using Finite State Machines to Design Software, Embedded Systems Design, vol.23, Ed.6.

[19] Ghosh et. al, D. 2007. Self-Healing Systems- Survey and Synthesis Decision Support Systems, vol.42, 2164-2185.

[20] Hoare, C.A.R 1969. An Axiomatic Basis for Computer Programming Communications of the ACM, vol 12(10), 576-583.

[21] T.S Chow, Testing software design modeled by finite state machines, IEEE Transactions on Software Engineering, 1978, vol.**4** (3), pp. 178-187.

[22] Reusnner, R.H., Schmidt, H.W. and Poernomo, I.H. 2003. Reliability prediction for component-based software architectures The Journal of Systems and Software, vol. 66, 241-252.

[23] Wood, A. 1996. Software Reliability Growth Models in TANDEM.

[24] Yadav, A. and Khan, R.A. 2009. Critical Review on Software Reliability Models International Journal of Recent Trends in Engineering, Vol 2(3), 114-116.

[25] Ramamoorthy, C.V. and Bastani, F.B. 1982. Software Reliability-Status and Perspectives IEEE Transactions on Software Engineering, Vol SE-8, No. 4, 354-369.