CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

The amount, size and complexity of data accumulated and created by humans is increasing exponentially each day\[^1\]. In such situations non-performance metrics such as security, reliability, availability etc have gained more importance\[^2\]. Ensuring instant, failure-free access to data has become a major challenge of the current century\[^3\].

Software Reliability can be regarded as an important software quality attribute that quantitatively assesses the continuity of acceptable service delivery\[^3-4\]. However, since its inception accurate software reliability prediction has been a difficult problem for the software engineer. Fundamentally, software reliability can be defined in two diverse ways:

i) The probability of failure free software operation in a specified environment for a specified period of time\[^3, 5\].

However, Software reliability actually denotes the trustworthiness of software in terms of the customer service it delivers\[^4\]. Hence, it is better defined as:

ii) The continuity of correct service delivery\[^3\].

The accuracy of the first definition is debatable as software failure is not a function of time but the side effect of the user input, operational environment or changes made to software\[^3\]. When software executes in real-time and produces desirable output it is 100% reliable, else it is 0% reliable. Considering this fact of software operation, Software Reliability is found to be more realistically defined with definition (ii) above. We shall thus follow the second definition through this thesis, as it is independent of time and is more generic to software\[^3\].

Current software research is aiming at autonomic, fault-tolerant and self-healing software. In this scenario reliable software operation at each runtime instance is of prime importance. To predict reliable software operation, traditionally many software reliability estimation methods and models \[^2-3\] were
proposed and applied. These software reliability models provide a general form for describing a random process called failure as function of number of failures experienced or time for next failure\cite{5}. However, recurring instances and losses of software failures have put the validity of the above models under scrutiny\cite{6-9}. Varied issues associated with software reliability estimation using these models have been raised\cite{3, 6-9}. We highlight some of the important ones below:

i) Existing software reliability estimation models use parameters like failure rate ($\lambda (t)$), Hazard Rate ($h (t)$), MTBF, MTTF etc for estimating software reliability. Analysis of the history of these parameters\cite{3,6} reveals that all these parameters were initially suggested for lifecycle reliability prediction of hardware products \cite{10}. Overlooking the underlying differences between hardware and software, conventional software reliability models use these parameters along with certain assumptions about the software failure process.

ii) Software unlike hardware is not visible unless it executes\cite{3}. Hence, software has distinctive characteristics that do not occur for physical hardware systems\cite{2-3, 8} like varied nature of faults, unique failure mechanism etc.

iii) Most models require post-failure data for reliability estimation of software under consideration\cite{3,6}. However, not many standard real-time data sets of software failure data are available in public domain.

iv) Further, a mechanism to collect failure data of software implemented in real world is required. This may be a difficult task as all customers may not report their problems or for a new system this data may be completely unavailable\cite{9}.

v) For accurate reliability estimation, the real handling time of the product by the customer should be determined to calculate failure rates\cite{2}. The failure rate parameter for estimating software reliability is data dependent and does not give accurate estimates.

vi) Though all reliability estimation methods rely on the postulation that reliability is the nonexistence of failures, they compute reliability using some sort of failure data\cite{3,6} (brute force).

vii) Modelling the software failure process is a challenging exercise \cite{11}. In sharp contrast to hardware, numerous other factors also manipulate software
failure rate, similar to\(^3\):

- Failure of any individual software elements.
- Failure caused through human factors/ operating documentation.
- Failure through environmental factors
- Common mode failure, here redundancy is routed by factors common to replicated units.

Though helpful in organizational decision making, software reliability theory itself suffers due to its misassumptions\(^{6, 12-14}\). **Software is itself a dynamic entity as its nature and behaviour continuously change depending upon user input and operational environment driving software execution**\(^2\). Software is designed to be used forever\(^3\); hence it should be 100% reliable.

Software does not suffer fatigue like mechanical and other physical devices\(^{15}\). Hence, advance prediction of system reliability like software behaviour is difficult to achieve. In current practice conventional software reliability models are close imitations of counterpart hardware reliability models\(^{16}\). These models as such are not representative of the nature and behaviour of target software at runtime. Although efforts are being made towards production of defect free software, such software cannot be guaranteed\(^{3, 9}\). In times when we are progressing towards **self-healing, autonomic, fault-tolerant, ubiquitous systems** reliable software operation is not a negotiable option\(^{17, 18}\).

The existing conventional models of software reliability estimation simply make predictions about software reliability and number of errors in the software underneath consideration\(^{12}\). **None of the existing models attempt to control software reliability throughout the software life cycle.** As per records, more than two-hundred software reliability estimation models are available\(^{6, 19}\). However, despite such large volumes of work, much work is still required in development of **a generic reliability control model that helps in failure-free software execution at each runtime instance.**

All the conventional models treat software as a black box and estimate software reliability using complex statistical or mathematical functions which
have been borrowed from hardware reliability models. Table 1.1 below depicts some common software reliability estimation parameters and functions directly borrowed from hardware reliability engineering.

**Table 1.1: Hardware/Software Reliability Parameters/Functions**

<table>
<thead>
<tr>
<th>S.No</th>
<th>Parameter/ Function</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Failure Rate ($\lambda$)$^{[20]}$</td>
<td>Standard data for hardware components and assembly types provided by MIL-HDBK-217$^{[21]}$. Ex: Jelinski-Moranda Model$^{[22]}$</td>
</tr>
<tr>
<td>2.</td>
<td>Hazard Function/ Failure Density Function ($\lambda(t)$)$^{[20]}$</td>
<td>First used as Force of Mortality for Actuarial Science. Represented as $\lambda(t)$ in Reliability Theory$^{[21,23]}$. Ex: Jelinski-Moranda Model$^{[22]}$</td>
</tr>
<tr>
<td>3.</td>
<td>MTBF (Mean-Time Between Failure)$^{[20]}$</td>
<td>Applied to constant failure rate models for electronic parts made to military standards$^{[21]}$. Standardised by MIL-HDBK-217. Represents average time among failures of hardware modules. Ex: Jelinski-Moranda Model$^{[22]}$</td>
</tr>
<tr>
<td>4.</td>
<td>MTTR (Mean-Time To Repair)$^{[20]}$</td>
<td>Time taken for repairing a failed hardware module. For software module is the mean time to reboot since software fault has been identified.</td>
</tr>
<tr>
<td>5.</td>
<td>MTTF (Mean Time To Failure)$^{[20]}$</td>
<td>Basic statistical unit for reliability (non-repairable systems). Mean time expected till first failure of a product. Ex: Jelinski-Moranda Model$^{[22]}$.</td>
</tr>
</tbody>
</table>

Detailed analysis of the history of the reliability estimation parameters listed in Table 1.1 reveals that all the parameters and functions were initially suggested as methods for lifecycle reliability prediction of hardware products$^{[24]}$. With inception of software reliability theory, these methods were simply fitted over to the software process. These functions and parameters fundamental to the conventional software reliability models formulatesuppositions about the software failure process to simplify estimates$^{[25-26]}$. In contrast to hardware, modelling
software failure can be a challenging exercise due to diverse, interlocking nature of software faults\textsuperscript{[3, 11]};

An amalgamation of one or more of the above factors united with factors like failure rate, probability of incidence, time etc work on almost all software systems underneath test and operation, hence leading to imprecision of software reliability calculations and estimates made using the conventional models and techniques.

If all software applications would comprise of the simple class of non-reactive, deterministic, closed and finite-state systems, complicated program analysis would have entirely substituted ad hoc testing techniques and software reliability estimation would have been easy\textsuperscript{[27]}. However, the non-determinism of a multi-threaded system can augment the difficulty of program verification by an exponential amount, potentially utilising all the gains made by the likewise exponential enhancement in computing power based brute force Moore’s law\textsuperscript{[28]}.

Reliability of software is highly dependent upon the user input \textsuperscript{[19, 29]} and operating environment of the software as there is great deal more feedback in software as compared to traditional engineered hardware systems. Software has its own problems in addition to being calculated. Thus, to determine software reliability we need a pure software-based approach.

A detailed analysis of the software reliability problem revealed that, the problem is actually far too complex for easy solutions \textsuperscript{[29]}. Separate from their classification, reliability techniques basically fall into two categories: \textbf{a priori techniques} (build the software right) and \textbf{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 \textsuperscript{[29]}. By shifting some of the balance towards a priori efforts; we go a long way towards correcting some of the most serious problems. Though we need both cure and prevention, but alittlebit of prevention saves a lot of cure.
1.2 RESEARCH OBJECTIVE

Aim of this study is to develop an automata-based formal reliability model for ensuring reliable software operation. We further establish the applicability of the model for accurate software reliability at runtime.

Our Detailed objectives are:

i. To prove that the major reason for the inaccurate estimates of traditional reliability models is due to the fact that all of them use failure data fitted with brute force techniques to make reliability estimates for given software\(^6\). Hence, our basic treatment applied for reliability estimates is faulty.

ii. An automata-model is a model with a strong theoretical basis. It was an obvious choice as the basis for a software reliability model because it has its roots in state-based approach which can be mathematically verified using Hoare’s law and predicate calculus\(^{28}\).

iii. This work will result into the development of a theory and harness that in tools that can be applied in the software development process to eliminate the brute force nature of software development methodologies in order to produce operationally reliable software.

1.3 CONTRIBUTION

This study examines why existing software reliability models fail to accurately estimate software reliability. It addresses holes in existing literature in terms of how the existing reliability models are designed and what are the limitations of this design. Aiming at addressing the problems of software reliability, this work contributes a novel and practical automata-based reliability control model that leverages operational reliability of software at runtime by taking into account:
i) The runtime structure of software

ii) The concurrent nature of software governed by user input and present software state.

The model is based on the fact that all software during execution is an automaton. Hence, its reliability is equivalent to the reliability of the automata. The research presented in this thesis is an application of the state machine concept to the probabilistic framework \[^{30}\]. The Probabilistic Finite Automata have been widely used and studied in a number of theoretical as well as practical areas \[^{31}, 32\]. Here we use a probabilistic finite automata representation of runtime software to monitor software path of execution at each runtime instance. Using this software execution path information, we detect faulty situations and use it for healing software. With this contribution we contribute to the task of systematically integrating fault free or reliable software operation in software execution engine (compiler/interpreter) itself.

As a major contribution, this work shall identify two programming idioms that can be used to enhance software reliability:

- The use of monitoring automaton to track a program’s execution in order to control both its correct and incorrect behaviour, and
- The use of self-healing, intelligent logic to safely recover from detected failures.

The approach laid in this thesis is a first attempt to describe software reliability taking into account the actual nature of software at runtime. The framework has been laid carefully enough to be applied to real-time software. The advantage of the proposed framework is that it will not simply be a tool to estimate software reliability; instead it can be extended into the following utilities:

i. Realize a trusted system, which can maintain itself to changes like if a new hardware/software component is added \[^{33}\].

ii. Develop a self-learning software control system that can learn its own faults and prevent their execution at the next instance.
1.4 INSPIRATION FOR AN AUTOMATA-BASED MODEL

An automata-based model was an obvious choice as the basis for a runtime software reliability model because:

i. The model has its roots in state-based approach which as stated above, can be mathematically verified using Hoare’s law and predicate calculus\[^{[28]}\].

ii. Similar work in network reliability estimation studies \[^{[15,33-36]}\] and developing software reliability estimation models for component-based software systems\[^{[37-39]}\] has also been attempted. However, detailed work for runtime software reliability control is lacking.

This work will result into the development of a theory that can be applied in the software development process to eliminate the brute force nature of software development methodologies in order to produce operationally reliable software.

We hypothesize that automata-based software reliability model is the best model for reliability estimation that will help us in estimating the reliability of a system accurately as well as detect conditions under which such an action can be invoked.

We further argue that the above hypothesis is in accordance with Hoare’s logic which is an axiomatic method of proving program correctness\[^{[28]}\] as discussed in next section.

1.4.1 Hoare’s Logic of Program Correctness

Hoare logic is a formal system for reasoning about Hoare style correctness formulae\[^{[28]}\]. The approach is partially based on the so-called intermediate assertion method of Floyd\[^{[28]}\] and has a significant impact upon the methods of designing and verifying programs. The approach owes its success to three basic factors, of which we outline the first two below:

i) The first factor, which it shares with the inductive assertion method\[^{[28]}\], is its universality: it is state-based, characterizes programming constructs as transformers of states, and therefore applies in principle to every construct.
Central feature of Hoare logic is **Hoare triple** that describes how the execution of a piece of code changes the state of the computation. It is of the form:

\[ \{P'\} \mathcal{C} \{Q'\} \tag{1.1} \]

*Where* \( P' \) = Assertion Precondition

\( Q' \) = Assertion Postcondition

\( \mathcal{C} \) = *command that establishes the postcondition* \( Q \)

Assertions in the above case are formulas in predicate logic

ii) The **second factor** in its success is its **syntax directedness**: The result of every rule for a composed programming construct is embedded into the rule itself. This reduces proving properties of that construct to proving properties of its constituent constructs. The latter are also formulated as **Hoare style correctness formulae** for proving programs correct. Hoare style correctness formulae for their constituent constructs can be used without any additional knowledge about the latter’s implementation.

Compositional Hoare logics can also be regarded as **design calculi**. Here a **proof rule is interpreted as a design rule**. Design rule implies a specific design goal of developing a program satisfying certain properties, is reduced to certain subgoals (and, in general, the satisfaction of certain verification conditions), obtained by reading that rule upside-down. **When applied to software, reliability of the software is reliability of the rules driving software execution.** The above properties of Hoare’s logic can be captured through automata.

From both the above factors we hypothesise 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 tracing how a particular piece of code will change the state of software computation and hence result in correct or incorrect system state.
To prove the **validity of the above hypothesis**, we aim at the development of a **formal model that can help in guiding and monitoring the implementation of software to control system reliability at any point in its life**. We propose an intelligent, self-learning software reliability model that can easily detect software error state, register the particular state and the transition that led to such a state in its memory. Further, the model can be trained to ensure that the software never repeats the transition that lead to the particular error state. We propose inculcation of self-healing properties in the framework, such that whenever software is executing to error state next (due to any operational or design anomalies), the system should be capable enough to halt software operation. At this point the framework may recall the previous correct state and should retreat back to the same in order to resume its alternate operation. **To achieve the above framework, we require representation of runtime program code and not source code.** Fig.1.1 below depicts an example of the type of software representation required as an input by our automata-based reliability model.
To realize the above design we propose that software executable should be represented using state-based approach as suggested by Hoare [28]. Each software component/module/block of software (smallest independently compilable and logically independent unit of code) should be represented as a cluster of nodes as represented above in Fig. 1.1. If each node in the cluster marked as the Learning Cluster (every node is achievable from the initial node and may result in either the correct node or error node) is assigned a probability, then the reliability of the whole cluster can be defined as the sum of probabilities of the correct nodes traversed. Further if the whole system is represented as a combination of such integrating and interacting clusters then the reliability of the whole system can also be calculated as a product of the individual reliabilities of each of these clusters.
1.5 OUTLINE OF THE THESIS

The thesis is organized into six chapters, which are structured as follows:

- Chapter 1 rigorously defines the different conventions and concepts used in software reliability modelling. It also briefly discusses the problem with existing software reliability estimation models and the major objective and contributions of this research.

- Chapter 2 discusses the nature of reliability as a concept. It also examines different models for estimating hardware as well as software reliability. Focus here is on highlighting the need for reliability estimation, how reliability of hardware and software components has been handled till date, differences between hardware and software reliability models and why a new approach to software reliability estimation is warranted. The chapter concludes with a detailed classification of existing software reliability models.

- Chapter 3 describes the basic fundamentals of the well-known state machine concept. Fundamental concepts of the finite state automata have been explained as they form the basis of the proposed model. Discussion of how automata are appropriate for describing software also follows. The chapter concludes with discussion of probabilistic finite state automata, an extension of the finite state automata used as the basis for monitoring runtime software execution.

- Chapter 4 discusses novel automata-based software reliability paradigm. The major characteristic of this framework is that it treats software during execution as an automaton and utilizes the characteristics of the automaton to control software reliability. The model is not just another mathematical or statistical function that estimates software reliability using post failure data. Instead it is a coherent framework for generating reliable software for even the most complex and difficult situations.
• Chapter 5 summarizes the manual simulation of the automata-based software reliability model on a Java-Based software application.

• In Chapter 6 discusses conclusions of this work as well as the possibilities of further extension and research.

• Appendix A summarizes some well-known conventional software reliability estimation models.

• Appendix B presents the automata-based representation of other modules of software application used for testing the automata-based model.

• Appendix C lists the publications during the course of this research. However publications relating to this work but under submission at time of this writing have been omitted from the list.

1.6 SUMMARY

This chapter discusses the motivation behind selecting the problem addressed by this research. Further with statement of the problem an overview of the proposed solution has been introduced along with discussion of novelty and contribution of the developed solution.
REFERENCES


