Automata-Based Software Reliability Model: The Key to Reliable Software

Ritika Wason$^{1,*}$, P. Ahmed$^2$ and M. Qasim Rafiq$^3$

$^1$ School of Engineering and Technology, Sharda University, Greater Noida, India
$^2$ School of Engineering and Technology, Sharda University, Greater Noida, India
$^3$ Department of Computer Engineering, Aligarh Muslim University, Aligarh, India

$^1$rit_2282@yahoo.co.in, $^2$pervez.ahmed@rediffmail.com, $^3$mqrafiq@hotmail.com

$^*$Corresponding Author

Abstract

In this paper, we critically analyze the success of the traditional reliability models built to measure and estimate software reliability. We further propose that a Finite State Automata (FSA) based reliability model can serve as a befitting solution to all existing software reliability challenges. The proposed model estimates actual system reliability at runtime. The main advantage of this model is that it allows authentic or real-time reliability estimation, prediction and can also be trained towards dynamic learning of the evolving behavior of software, and fault tolerance.

Keywords: Finite State Machines (FSMs); Software Reliability; Automata-Based Software Reliability Model; Finite State Automata

1. Introduction

Technically, reliability is the probability of failure-free software operation for a specified period of time in a specified environment (ANSI, 1991). The term has gained momentum recently due to the increasing penetration of software-based utilities in every human endeavor. This escalating dependence of human society on software has fuelled the demand for reliable software to ensure the smooth functioning of modern society.

It has been observed that effective software measurement has the potential to play an important role in risk management during software development [17]. However, software development being a human centric task is very much prone to errors. Despite decades of research and numerous reliability estimation metrics, software failures remain inevitable. The exploration for better models for software reliability estimation remains an open research area [20]. Some standard coding practices like extensive error handling facilities contribute towards rendering systems fault tolerant to certain types of errors and exceptions [20, 31]. However, these methods are tightly coupled with software codes and are highly application specific. Therefore, software testing and error handling processes are being increasingly supplemented by different formal verification techniques [1, 14, 23, 10]. Out of these, Model Checking [14], Finite State Machines [1], State Based Models [23] and Operational Profiles [10] are notable. In addition, Self-Healing systems which heal themselves from system faults are also being developed [18]. Formally, a self-healing system performs even in the presence of faults by overcoming faults or recovering from them with minimal human intervention.

Software during execution is a Finite State Machine (FSM). Hence, Finite State Machine (FSM) based techniques are the most suitable to represent an executable software. FSMs represent software as a collection of nodes representing states and the edges transitions...
between states. In this paper we propose and advocate an automata-based reliability model as an accurate reliability estimation tool for executable software systems. The remainder of this paper is organized as follows. Section 2 discusses the problems and issues cursing software reliability modeling science. Section 3 introduces the formal finite state machine model and discusses its applications in software systems. Section 4 analyzes the traditional reliability models and enlists the major causes for their inaccuracy. Section 5 traces the different extensions and applications of finite state machine (FSM) models in designing and testing reliable software systems. Section 6 compares the traditional reliability models over automata-based alternatives and establishes the effectiveness of the latter. Section 7 discusses the automata-based reliability model by explaining the basics of the proposed model. The section also makes some future predictions regarding the growth and acceptability of automata-based models as an absolute technique to model reliable systems. Section 8 concludes the discussion by suggesting some future research trends in this domain.

2. Software Reliability Problems

No human activity can enjoy zero risk and no equipment a zero failure rate [29]. Despite the present day advancements, the ordeal for software still remains the domain of reliable software. This inter-relationship between the number of bugs and errors in software and the escalating costs can be demonstrated by the instance where a single defect may cause lower in diagnosis and repair if detected early in development, while the same may cost astronomically higher to rectify after deployment [4]. Similarly, if a failure transpires to be a design fault the cost of redesign, documentation and retesting may well be in tens or hundreds or thousands of rupees. The above situation demonstrates the importance of using effective reliability assessment techniques to control and prevent potential failures before they execute. The resulting estimate, though an imprecise measurement, is still a valuable exercise.

Completely reliable software despite the introduction of so many reliability estimation and statistical models still remains a distant dream. More than quarter of a century has passed since the first software reliability model appeared [43]. Many more dozens have since been developed [19, 29, 36, 42]. However when probed it has been well-established that the current dissatisfaction stems from the manner in which reliability as a concept has been applied and how the related models have evolved. We now highlight the reasons behind the inaccurate estimates of the existing software reliability estimation models.

2.1. Limitations of Reliability Estimation Models

Traditional theory of software reliability has its roots in hardware reliability [32]. However, for software systems the concept of reliability has its own unique characteristics which are strikingly different from hardware. In contrast to hardware, software reliability is complicated by several factors like human operator, hardware subsystem and operating environment. Hence, any of documentation, training and interface problems can (directly or indirectly) induce software failure and affect the final reliability of a software system.

Further, to evaluate reliability, traditional reliability models are arguably the customary approach. Notably, all these models are parametric as they derive an abstract mathematical model that uses several parameters determined from empirical data. The precision and estimates offered by these models are dependent on the accuracy of these parameter values [17]. To estimate these values all models make some assumptions.
Incorrect model assumptions may result in incorrect estimates. However, none of the existing models account for the effects of incorrect model assumptions. Thus traditional models for reliability modeling provide inaccurate estimates of system quality and also lack the ability to quantify their inaccuracy. In support of the above claim we consider one of the oldest models for reliability estimation, the Jelinski-Moranda Model. This model constructs its reliability function based on the following assumptions:

- The initial number of faults in a program is finite;
- Each fault in the program has the same probability of detection;
- Time between failures is exponentially distributed and faults are removed as soon as they are detected.
- A careful observation of the given assumptions reveals that most of them are contrasting to the actual real-time environment.

Clearly, the above assumptions do not model real-life situations. Hence, the inaccuracy of these assumptions is bound to affect the accuracy of the estimation. Further a computer is an electrical machine that consists of both hardware and software; hence both software and hardware contribute to the overall system reliability. However, hardware reliability concepts had been successfully developed and understood long before the inception of software reliability [16]. Hence it was quite natural that the same concepts would be applied for developing software reliability models. But hardware modeling techniques do not work in the software environment as there are fundamental differences in the two environments, like:

- Hardware is a physical entity, whereas software is a logical entity which remains invisible till it runs.
- Hardware is generally machine-produced and its faults are mostly wear-out faults whereas software is designed and reused instead of being machine-produced and its faults are generally design errors.

The ever-increasing complexity of modern engineering products and systems further ensures that system failure may not always be a result of component part failure [43]. Many other factors may also influence the system failure rate. Some of them are:

- Failure of individual software elements or components.
- Failure due to human factors.
- Failure due to environmental factors
- Common mode failure, where redundancy is defeated by factors common to replicated units.
- Failure due to internal system factors like hardware failure.

Combination of one or more of the above factors combined with factors like failure rate, probability of occurrence, time etc, work on almost all software systems under test and operation [32, 49]. Any combination of one or more of the above factors may lead to inaccurate software reliability predictions and estimates done using traditional models and techniques.

3. Software is a Finite State Machine

Section 2, establishes that traditional reliability estimation methodologies are based on certain misassumptions and have thus failed to provide accurate reliability estimates. We argue that as software in execution is a Finite State Machine (FSM) or Finite State
Automata (FSA), we should use a finite state machine representation of an executable software system to accurately estimate system reliability during runtime. Finite State Machines are a popular formal model used to handle several complex problems in a variety of domains [11]. Formally, a Finite State Machine is defined as:

\[
A = (Q, \Sigma, q_0, \delta, F, h, R)
\]

where:
- \(Q\): Finite set of states,
- \(\Sigma\): Alphabet of input symbols,
- \(R\): Output alphabet
- \(q_0\): One start (initial) state where \(q_0 \in Q\),
- \(F\): Zero or more final (accepting) states where \(F \subseteq Q\),
- \(\delta\): A transition function \(\Sigma \times Q \rightarrow Q\),
- \(h\): Output function \(h: Q \rightarrow R\)

For Software, a Finite State Machine is commonly used as a behavioral model composed of a finite number of states, \(k\) and transitions, \(\delta\) between those states to give an abstract description of the behavior of a software system.

It is important to note here that such an automaton can also be induced with learning ability so that it can modify its behavior dynamically depending upon its operating environment, current state and transition condition. A Finite State Machine is thus a mathematically defined object that can easily provide structured and precise understanding of what is going on in systems represented as state machines. Contrastingly, state machine models for real-life systems may not be as complex as theory as here states are strings and events are methods that cause transition from one state to another.

3.1. Advantages of Finite State Machine Based Software Representation

The major advantage of this formal automata-based model for software representation is that system behavior can be studied as a finite set of states in the FSM. Notably, hardware designs have also been modeled as some kind of finite state machines, such as \(\omega\)-automata or Büchi Automata (BA) [30]. A Finite State Machine based model can further allow easy identification of an error state caused due to factors like incorrect input, misrepresentation of values, incorrect bounds checking, memory overflow etc. Software can further be trained to become self-learning by tracking back from an error state to say, its most recent correct state and then identifying the cause of the incorrect state, indexing the cause in its knowledge base (learning) and tracing an optimal solution to result in some correct state (reasoning). The above design if realized can serve as the foundation of a future software engineering paradigm for ensuring optimal quality software which prevents errors in the first place and learns from them in the second place.

The use of Finite state Machines (FSM) has also been advocated for the solution of common design problems [27]. For system design, an FSM can be easily used to represent all system components passive or active like the controller or even the data store [27]. The advantage of using FSM design here is that it permits programmers to define multiple instances of Finite State Machine (FSM) and their associated behavior without actually changing code. However such an FSM model should satisfy two basic constraints [27]:

1) Every state must have a defined transition for every possible input and
2) Every state must have a defined transition to handle cases where a timeout occurs (recognize failure). FSMs are an important, formal and mathematically sound model of software representation because:

1) FSMs have decades of mathematical and computer science research behind them.
2) FSM allows parsing a problem into states, where each state may produce different results.
3) An unreachable state in this model directly implies a flaw in requirements or design.
4) FSMs allow association of software or hardware actions with a current state and an input combination (Mealy State Machine) or just the current state alone (Moore State Machine).
5) FSMs allow controlling how a component of software flows, how the user interacts with the system and how the system responds to certain stimuli.

3.2. Finite State Machine Applications

Modeling with state machines has been a well-researched design model [10, 14, 47]. The extended finite state machine (EFSM), an improvement of Finite State Machine that contains associated triggers with transition conditions for data flow representations has also been employed for a wide range of systems [15, 28]. A common choice in this case can be the use of transition coverage, which can be easily expressed as transition paths through an EFSM. An integrated search-based approach to automate testing from an EFSM has also been proposed in [28]. The proposed approach works in two phases. The first phase produces a feasible transition path (FTP) using a Genetic Algorithm with a TP (transition path) feasibility metric based fitness function. The second phase further searches for an input sequence to trigger this TP using a combination of branch distance function and approach level. The paper empirically validates its approach by applying it on five EFSMs. The experiments for the work demonstrated 96.6% effectiveness in the first phase and 100% success rate in the second phase. The results of the work are noteworthy and establish the effectiveness of EFSMs in automating testing beyond doubt.

FSM-based models and its extensions have also found extensive use in designing and testing different kinds of systems. Some of which are highlighted in the table below:

<table>
<thead>
<tr>
<th>S.No</th>
<th>FSM Application</th>
<th>FSM Type</th>
<th>Related Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Realistic Component Protocol Modeling</td>
<td>Extended FSM (EFSM)</td>
<td>Wehrheim and Reussner [47]</td>
</tr>
<tr>
<td>2.</td>
<td>Testing of Interactive Systems</td>
<td>Probabilistic and Stochastic time FSM (PFSM, PSFSM)</td>
<td>Fantinato and Jino [15] and Madani and Parissis [31]</td>
</tr>
<tr>
<td>3.</td>
<td>Integrated Search-Based Approach for Automatic Testing</td>
<td>EFSM</td>
<td>Kalaji et al. [28]</td>
</tr>
<tr>
<td>4.</td>
<td>Partial order reduction for state/event LTL</td>
<td>Component interaction automata</td>
<td>Benes et al. [6]</td>
</tr>
<tr>
<td>5.</td>
<td>In-stent restenosis: Two-dimensional multi scale modeling and simulations</td>
<td>Complex Automata</td>
<td>Ciauzzo et al. [8]</td>
</tr>
<tr>
<td>6.</td>
<td>Learning from Queries</td>
<td>Deterministic finite cover automata</td>
<td>Ipate [25]</td>
</tr>
<tr>
<td>7.</td>
<td>Bottom-up automata on ranked tree applications</td>
<td>Rigid Tree automata (RTA)</td>
<td>Jacquemard et al. [26]</td>
</tr>
<tr>
<td>9.</td>
<td>Reactive Automata</td>
<td>Reactive Automata</td>
<td>Crochemore et al. [13]</td>
</tr>
</tbody>
</table>
Table 1 establishes that FSM-based models are being increasingly utilized to develop different software systems. Testing of such systems has proven them to be more reliable as compared to others. If FSM-based systems can prove to be better in varied real-life applications, we argue that an automata-based reliability model can also serve as the basis of accurate reliability estimation and can further be simulated as a control tool for intelligent system operation.

Despite the increasing acceptability of the finite state machine model for designing systems and achieving a more reliable system, absolute acceptability of this model still remains debatable. The applicability of such techniques remains limited to only a few operational domains and only under some operational assumptions [40]. Also, despite the popularity of EFSMs, testing from an EFSM can be difficult due to: path feasibility and path input sequence generation [1, 15, 28]. Path feasibility problem relates to generating feasible paths, whereas the path input sequence generation problem is to find an input sequence that can traverse a feasible path. Further, it has also been noted that the current techniques of FSM testing are statistical and rely on simulation. As a result these techniques often fail to identify critical cases in FSMs.

4. Traditional Approaches for Software Reliability Estimation

The commonly used approach for estimating software reliability has been the use of some software reliability model applying varied software metrics [16]. In spite of the above, ensuring reliability remains one of the most prominent problems of software development. With the current outburst of technology, any entity is produced in order to execute failure free its anticipated tasks at a desired performance level for some given period. This set of quality features have been united under the common notion of reliability [32]. In general terms, reliability of a product is the probability that the underlying item will perform its intended functions for at least a given time. Hence, if T variable represents the time-to-failure, then the reliability function associated with T at time instance T₀, can be represented mathematically as given below in Eq.1:

\[ R(T_0) = \text{Prob} \{ T \geq T_0 \} \]  

(1)

Consequently, the complement of reliability stated as non-survival or non-reliability can be represented as in Eq. 2:

\[ F(T) = 1 - R(t) \]  

(2)

From Eq. 1, it is clear that at:

\[ T = 0, R(0) = 1, F(0) = 0 \]  

(3)

and at

\[ T=\infty, R(\infty) = 0, F(\infty) = 1 \]  

(4)

Eq. 3 and 4 above imply that any human engineered product is bound to fail after a certain period of time. Hence, completely reliable software which works for an indefinite time period is too high an expectation. However, accurate reliability predictions for all software created is something one can strive for. The reason for the inaccuracy of the traditional reliability models is the fact that all of them use experimental data collected in terms of testing time between failures, using two or more parameters. The two most important parameters being, the mean value, µ (t) and the failure intensity function, λ (t) respectively. Table 2 below highlights these values for some basic models:
Table 2. Mean Value and Failure Intensity Functions of some basic Reliability Models

<table>
<thead>
<tr>
<th>S.No</th>
<th>Model</th>
<th>Mean Value Function ($µ(t)$)</th>
<th>Failure Intensity Function ($λ(t)$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Musa’s Basic Execution Time</td>
<td>$V_0(1-exp(-\lambda_0 t))/V_0$</td>
<td>$\lambda_0 exp(-\lambda_0 t)/ V_0$</td>
</tr>
<tr>
<td>2.</td>
<td>Logarithmic Poisson Execution Time</td>
<td>$1/θ Ln(\lambda_0 θ t+1)$</td>
<td>$\lambda_0 exp(-θ µ)$</td>
</tr>
<tr>
<td>3.</td>
<td>Jelinski – Moranda NHPP Model</td>
<td>$N(1-exp(-Φ t))$</td>
<td>$NΦ exp(-Φ t)$</td>
</tr>
<tr>
<td>4.</td>
<td>Goel-Okumoto NHPP Model</td>
<td>$a(1-e^{-bt})$</td>
<td>$abe^{-bt}$</td>
</tr>
</tbody>
</table>

Interestingly, all the above models estimate parameters by fitting the mathematical model in question to the available failure data. Further, none of the existing reliability estimation models can be applied generically under all situations to all data sets.

All the traditional reliability models suffer from the problem of unsatisfactory evaluation quality, as a result many new models are being developed continuously. According to IEEE standard 982.1 (1998), the various factors related to the software product, process and resource in the whole software life cycle have great influence on reliability of any software system. [50] analyzed and ranked 32 parameters affecting software reliability in terms of their impact by devising an evaluation method based on influencing factors. However, the work is more of a guiding process reinstating the need for improvement in the reliability estimation process.

Though much previous work exists on software reliability models, [38] observe that their credibility in practice suffers from a number of challenges.

1) The operational profile during testing may differ from the actual operational profile in use.
2) It becomes difficult to make statistically meaningful reliability predictions with limited failure data collected in a project.
3) Software reliability models may make some unrealistic assumptions regarding independent occurrence of faults, fault occurrence probability and absolute fault correction.

Traditional reliability models are also ineffective in estimating software reliability of modern, heterogeneous, component-based software systems [19]. These models treat the system as a black box and model its input/output behavior without any consideration of its internal structure. The inherent limitations of traditional models and their ineffectiveness in estimating reliability of systems developed under the component-based paradigm have shifted attention towards structure-based reliability assessment techniques [19]. Predominant research in the area of structure-based software reliability assessment is focused on the development of state-space models for reliability prediction [19]. State-space models represent the structure of the software system as a control flow graph and estimate reliability analytically [23]. These models have been widely used and implemented in research focusing on development of automata-based software systems as well as automata-based reliability evaluation models like fault tree analysis (FTA) models, DA Charts [34] (Dependability Assessment) and many others. The models have also found successful application in describing dynamic aspects of system behavior [49] irrespective of the complexity of the system under study.
5. Reliable Computing: Present State of Affairs

Software Testing and Software Reliability are two grey areas of software engineering where despite much research, failure has recurred and unreliable software is still being designed and delivered. Computer programs under execution are finite state machines. The very idea of modeling software as a finite state machine holds the potential of solving a large number of problems in automaton, electronic design, parsing and many other engineering applications. However, a major area where the application of this idea can prove to be a great success is in the design of self-learning, self-healing, error recovery software which are expected to perform reliably [18, 33].

Automating the task of fixing faults has been identified as the next natural step in full automated software engineering [4]. Though there has been some previous work on fixing code automatically [4], it suffers from heavy constraints on type modifications that can be automatically done on the code. As a result only limited classes of errors can thus be identified and handled. A software system is a combination of both functional and non functional characteristics. Although the functionality of software is important for its adoption, the critical prerequisite for a system’s success is the exhibition of non-functional quality properties like performance, security, usability and reliability [5, 20]. Traditional formal methods concentrated on the representation of functional behavior or properties of the system under test. However, such methods do not allow expression of non-functional system properties like time, probabilities etc. Different formalisms have been extended to deal with such properties. New languages allow for explicit representation of both probability (for shared resources) and time for performing a particular task [20]. To specify systems with probabilities, probabilistic finite state machines, PFSMs (machines with real number probabilities assigned to initial states, transitions and final states) have been used. The concept of specification mutation (mutating a specification) has also been applied with finite state based models containing non-functional properties [20]. Here the mutation testing technique has been extended to probabilistic finite state machines (PSFSM) which have stochastic time attached to its transitions. In such cases mutation testing involves applying a test sequence multiple times to observe the resultant behavior among the model and its mutants. The work developed test sequences for mutants of PFSMs (FSMs with probabilities attached to their transition) and PSFSMs by using mutation operators to distinguish any PFSM say $M$ from its mutants $M’$. State-based embedded systems also use probabilities; hence the application of the proposed technique is possible to many critical systems with real-time constraints.

Another approach that has recently gained a lot of attention in software design and development is model-based testing (MBT). Informally, MBT is a general term that signifies an approach that bases common testing tasks like test case generation and test result evaluation on a model of the application under test [14]. Model-based techniques have substantial potential. Many studies using these techniques for testing a variety of applications [50] has met with success, like testing of Graphical User Interfaces [48]; Testing Embedded Controller Software [14]; Trustworthiness Testing of Phishing Websites, [41] and many more. Finite State Machines (FSMs) are one of the most common modelling techniques for MBT. The increasing complexity in software development has stimulated the application of new techniques that help deliver systems in shorter time and lower cost. One such technique is component-based development (CBD) that creates software from pre-fabricated autonomous components as discussed in [14]. Software systems assembled from a large number of autonomous components need to be formally verified in order to ensure the correct interplay or interaction among components. This is an important issue controlling the effectiveness and reliability of such systems. Previous studies [40] have incorporated both
states and events to express important properties of component-based software. A partial order reduction technique for the verification of state/event LTL (a combination of state-based/action-based linear temporal logic) properties has also been proposed [6].

Component-based development is currently gaining momentum in the software engineering community. An approach for determining the reliability of component-based software architectures using rich architecture definition language (RADL) oriented towards platforms like Microsoft’s .NET and Sun’s EJB has also been worked upon in [40]. The proposed method uses parameterised contractual specifications based on state machines for efficient static analysis. The researchers argue that as the reliability of a component most strongly depends on its environment, hence reliability models can be parameterised by the required component reliability in a deployment context [40]. The work further re-establishes the fact that fault tolerance also requires a systematic and formal approach to reliability [35]. The notable findings of the study being the fact that each component is expected to render certain services, at the concrete implementation level these services can be realised by method executions. These method executions consist of a number of sub steps, each characterizing relevant transitions for specific states. For a failure-free execution each run should traverse all these states and thus the reliability of the component can be modelled as the product of separate reliability factors which may be numerous in case of a real-time software.

5.1. Reliable Autonomous Computing

An important research area which has garnered lot of interest and effort in the previous decade has been the area of self-healing and autonomic systems [2, 18]. With the basic purpose of managing complexity, Autonomic Computing deals with the idea of self-managed systems that embody the visionary notion of self-CHOP systems that can self-configure, self-heal, self-optimise, and self-protect themselves. Autonomic systems attempt to automate the management of hardware, software and network infrastructures, hence alleviating the need of human intervention. A relatively naive domain with a far-sighted mission, autonomic systems can be broadly classified into two main categories of Autonomic Computing and Autonomic Networking [4]. However, the realization of the purpose of this paradigm lies in the merging of these two domains to give rise to a new theory of Autonomic Systems. Many large IT organizations have and are investing hundreds and millions of dollars to bring autonomic capabilities to many of their products (like HP 3 PAR software family Data sheet [22] Tivoli Software Products [21] Ahuja [2], IBM WebSphere Virtual Enterprise Version 7.0 [24] and Oracle Database 11g versus IBM DB2 UDB v9.7 Manageability Overview [37]. Though a complete autonomic system does not exist yet, autonomic technologies have already become a vital part of many important systems today. Table 3 below lists down a few notable applications enabled with autonomic capabilities:

<table>
<thead>
<tr>
<th>S.No</th>
<th>Applications</th>
<th>Autonomic Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>IBM WebSphere Virtual Enterprise</td>
<td>Autonomic Management [24]</td>
</tr>
<tr>
<td>2.</td>
<td>IBM Tivoli Change and Configuration Management Database</td>
<td>Tracking IT information about relationships and dependencies between products and their component [21].</td>
</tr>
<tr>
<td>3.</td>
<td>IBM DB2</td>
<td>Self-Configuring and Self-Maintaining Features [2].</td>
</tr>
<tr>
<td>4.</td>
<td>IBM Tivoli Usage and Accounting Manager</td>
<td>Automatic Resource Accounting, Cost Allocation, Charge-back Billing [21]</td>
</tr>
<tr>
<td>5.</td>
<td>IBM Tivoli Security Operations Manager</td>
<td>Autonomous Data Centre Information Analysis for Threat Detection and Response [21].</td>
</tr>
<tr>
<td>6.</td>
<td>HP’s 3PAR Adaptive Optimization Software</td>
<td>Autonomic Storage Solution for virtual and cloud data centres [22].</td>
</tr>
<tr>
<td>7.</td>
<td>Oracle 11g</td>
<td>SQL Performance Analyzer, Database Replay, Automatic Database Diagnostic Monitor (ADDM) [37].</td>
</tr>
</tbody>
</table>

Copyright © 2013 SERSC 119
Table 3 depicts that autonomic systems are coming up in a big way; however, the release or even design of completely autonomic products is still distant as it requires consideration of many practical issues, challenges and problems like accurate reliability designing and estimation techniques and models for such systems.

Another research endeavour worth mentioning is the domain of Self-Healing Systems. Self-Healing Systems attempt to heal themselves in order to recover from faults and regain normative performance levels independently without any human intervention [18]. Notable research has been done for Self-Healing or Survivable systems under the domain of Fault Tolerant or Recovery Oriented Computing. Many different architectural models have also been suggested to realize a self-healing system that can recover from abnormal (“unhealthy”) state and return to the normative (“healthy”) state it was before disruption. A detailed literature review in the area of self-healing systems and a classification based on similarities and relationships was suggested in [18]. Though most of the research on self-healing systems is still in its infancy, varied prospective application areas for such systems have already been suggested in [17]. Table 4 below highlights some of these promising applications.

### Table 4. Self-Healing System Applications [17]

<table>
<thead>
<tr>
<th>S.No</th>
<th>Self-Healing Applications</th>
<th>Self-Healing Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Grid Computing</td>
<td>Self-adaptation to heterogeneous environments; error detection; System Monitoring.</td>
</tr>
<tr>
<td>2.</td>
<td>Software Agent-Based Systems</td>
<td>Multi-agent redundancy for software adaptation.</td>
</tr>
<tr>
<td>3.</td>
<td>Service Discovery Systems</td>
<td>Reliable views of distributed systems in different network topologies.</td>
</tr>
<tr>
<td>4.</td>
<td>Reflective Middleware Model</td>
<td>Reconfigurable compilation of different components.</td>
</tr>
<tr>
<td>5.</td>
<td>GRACE</td>
<td>Integrated cross-layer adaptive system for resource constrained systems.</td>
</tr>
<tr>
<td>6.</td>
<td>Clustering</td>
<td>Application inter-dependency and resource allocation management without service interruption.</td>
</tr>
</tbody>
</table>

The basic goal of these self-healing systems is to design a survivable system that is reliable and dependable. The area has also generated enough commercial interest like Microsoft MSI files for fixing Windows XP by creating a system restore point; self-healing capabilities in IBM’s WebSphere initiative; Microsoft’s Dynamic Systems Initiative etc., [18]. However, its realization into a reliable software engineering paradigm with a well-defined model for construction of autonomous self-healing software systems remains to be realized.

### 6. Automata-Based versus Traditional Approaches: A Comparison

The goal of estimating the reliability of a software system is to anticipate possible software failures once the software is under operation. Designing reliability into software is not easy since increasing performance demands on real-time software amalgamated with recent advances in technology have made way for more complexity in software. However, engineering reliability in software systems is still considered difficult as calculating reliability
is expensive as well as challenging. Software being a logical entity, the metrics involved in calculating reliability is different from other disciplines of engineering and thus difficult to determine. The traditional software reliability growth models as discussed in preceding sections though a popular tool for estimating reliability since many decades have failed miserably in delivering precise and accurate reliability predictions for most software systems. As a result, research for alternative models for accurate reliability estimates of complex, real-time software has gained momentum in the present scenario.

Automata-based reliability prediction models have emerged as efficient models for overcoming prediction inaccuracy of traditional reliability estimation models. From above discussion it can well-be reasoned that the simple fact that automata-based software can be easily modeled as a state-based system makes it ideal for representing real-world, entity-based problems. As a result of this state-space software representation reliability estimation of software represented using FSM approaches are definitely more accurate and hence effective in producing better quality software. The state-space approach is better said a generative approach that can allow modeling of dynamic, complex systems [23]. The approach is already finding varied implementations in many different domains like Stochastic Petri nets and other Petrinet-based models, weighted automata, fault trees, hierarchical state-based architectures, composite state-models etc. The comparison can be concluded with an effective argument that all software can be best represented as a state-space model. This finite state machine representation is provably a better model for software reliability estimation. In comparison the traditional reliability estimation approaches are brute-force models that use an exhaustive number of failure data with comparative ease to predict reliability of a software system.

7. Automata-Based Reliability Model

Consider a common scenario: software receives an input and then generates some output. Depending on the prior result the software again reappraises the next set of possible inputs. From above we can say that software is always in some specific state. Hence, state-based models are a logical fit for software system representation. State-based models treat software as a state machine that begins from a start state and can then acquire any of a set of well-defined states or an error state depending upon user input before it terminates at the final state or fails. A Finite State Machine-based model is thus appropriate for design of any software that is dependable, reliable and can be trained to be self-maintainable.

Automata or FSM-models have been around even before the inception of Software Engineering. Their use in design and testing of computer hardware components is very well-established and considered a standard practice today. Similarly state machines are an ideal model for describing sequences of input linked by various transitions to result in some output. Hence, development of a software engineering paradigm using finite automata is a viable solution for realizing intelligent software systems whose field reliability matches their estimated reliability. FSM-based models are already being successfully used for network reliability estimation and representation [7, 12]. In a very large network there is a high probability that at least one device will be in a failed state at any given time, yet the network should provide uninterrupted service to the vast majority of its users [7]. An interesting model based on stochastic graphs for assessing and comparing the reliability of Local Area Networks was suggested by [7]. The advantage of the proposed model is that it considers individual reliability of each component part of a network in assessing the complete network reliability of the whole network. To model the effects of user activities on the network the model introduces a weighing factor \( w_i \) to measure the importance of a particular parameter failure. Using the above priority weighting scheme the model further computes weighted-average-network reliability \( R \) for each network.
Though the above model applies to network reliability computations, we can use the basic idea of the same for reliability estimation of other software also. Using the above model by [7] as basis, we propose an automata-based software reliability model that represents any software system as a stochastic graph where nodes (system states) of each cluster (part of the software system that can be represented as a single block or module). In this model each node can be assigned a priority. Cluster or module reliability can be expressed as the product of weights of each traversed nodes of the cluster and the final system reliability can be expressed as a summation of the relative priority or weight of each cluster. The model is depicted in Figure 2 below:

![Figure 2. Representation of a Software Block/Module as a Set of Distinct States [46]](image)

In Figure 1 above if all nodes of the learning cluster are equally accessible from the initial node i, the priority of each node can be set to 1. Then \( w_i = 1/5 \) for all nodes and

\[
R = \frac{1}{5} R_1 + \frac{1}{5} R_2 + \frac{1}{5} R_3 + \frac{1}{5} R_4 + \frac{1}{5} R_5
\]  

(5)

The final system reliability using the above approach can be further calculated using an extension of approach suggested in Bowles [7] using probabilistic estimates. The proposed model can not only allow reliability estimation and prediction but can also be trained for dynamic learning of the evolving behavior of software, and fault tolerance as it can easily retract from erroneous state to safe state during software operation.

8. Conclusion and Future Scope

Finite State Machines are a formal, mathematically-sound approach applied to both hardware and software design. The use of an FSM model to many software applications is fairly common nowadays. This paper surveys and evaluates the three basics: what, why and how, concerning the representation of software as automata. The highlight of this study can be the fact that it is a first of its kind attempt on feasibility analysis for FSM-based software’s. One important conclusion from this work is the fact that most of the work in this domain has concentrated on designing or testing different kinds of systems for different domains of human life using different types of FSMs. However, the actual power of the FSM-based model and inherent reliability of such software has not been explored completely. Hence, though the FSM model has found numerous applications in software, the concept is still open.
for research. A few major areas in which work on FSM-based software systems can be further extended are:

i) Different types of FSMs have been used to design and test different systems. Major types among these which have surfaced in this paper and form the basis of many experimental studies have been the probabilistic finite state machine (PFSM), probabilistic stochastic finite state machine (PSFSM), extended FSM and asynchronous automata to name a few. However, performance comparisons and reliability estimates of these FSMs can yield important results for software reliability and FSM effectiveness.

ii) Upon the launch of its Autonomic Computing initiative, IBM defined its four major properties as: self-Configuring, self-Healing, self-Optimizing and self-Protecting [6]. The above properties if used as the basis to design a FSM-based software design model can result in the creation of an investigative (investigates its error state), intelligent (learns from its own faults) and reliable (backtrack from its error state to a correct state) system.

iii) FSM research can also be directed towards generation of automata with explanation facility for its own actions (self-clarifying system).

iv) The design of reliable systems has also garnered sufficient commercial interest; as a result, many IT giants are fostering ideas like autonomic computing and self-healing systems. However, if the concept of autonomic systems and self-healing systems are merged together and coupled with industry applications like grid computing, software agents, middleware components etc., will substantially decrease maintenance time and enhance decision-making capabilities.

v) Another important research area for FSM-based software models can be in improving the speed of recovery. Recovery for any system should be quick enough to restore a malfunctioning system to normalcy as soon as possible. This idea can be realized if we make efforts in designing intelligent self-managing and self-healing software systems.

vi) [35] suggest that modern computers use brute-force statistical methods to execute an exhaustive number of iterations with comparative ease. They have proved Monte-Carlo methods to be another class of brute-force computer intensive methods of simulating the behavior of representative data, using massive sets of random numbers. The same principle can be used to prove the inaccuracy of the traditional SRGMs over automata-based models as listed in Section 6.

The very idea of treating software as a FSM holds immense power in reducing the failure rate of software system. With the help of appropriate research and experiments this is one area of software that holds the promise of changing our software engineering practices and overhauling our traditional software design model to a design model that shall guarantee near about absolute system reliability. However, at the time of this writing we would say that the idea is still in its infancy and requires efficient, timely and bias-free research before its actual implementation.

References


