CHAPTER 3

AUTOMATA- BASED REPRESENTATION

Concept of state machine has been fundamental to the field of computer science since early 1940s\cite{1}. Formally, a state machine also called automata is a device that can store the significance of something at a given occasion and can work on input to change the significance or reason an action or output to occur in response to a change\cite{2}. Robert Lusser, 1952 \cite{1} first formally defined reliability from an engineer’s point of view as known today. Birnbaum, Esary and Saunders in their 1961 paper on coherent structures \cite{2} first represented a coherent system as a directed graph \cite{1}. Following this 1980’s was the era of network reliability \cite{1}. This phase marked the beginning of a sophisticated graph theory oriented line of research for system reliability calculation \cite{1, 3}. The above discussion points towards the fact that mathematical graph theory has been used in system reliability estimation since the beginning of mathematical theory of reliability \cite{1, 2}. The reason for choice of graph theory can be attributed to many diverse factors like simple representation, formal model of representation etc. Extensions of graph theory especially finite state machines have formed the basis of considerable reliability research especially in the area of network reliability estimation \cite{3, 4}.

State Machines are a fundamental architecture used to assemble applications quickly. State Machine architecture can be utilised to realize any compound decision-making algorithms where some action is performed for each state in the machine. State machines are also used for software architecture representation\cite{5, 6}. Architecture of software gives a holistic view of software \cite{7}. We argue that software during execution is an automaton. Hence the reliability of software is analogous to the reliability of the automata.

On basis of this fact we propose an automata-based software reliability control framework. Before we discuss our approach in the next section we discuss the mathematical concepts and notations of automata in this section.
3.1 STATE MACHINE CONCEPT

In late 1960’s structured methods emerged as a method to help manage large, complex software \[^4\]. A state machine is a mathematical representation of calculation used to devise computer programs as well as logic circuits \[^5, 6\]. **A state machine operates through states which are actually data structures (non-empty sets with a number of operations (operations over the set and relations))**\[^6, 8\]. The major advantage of the state machine formalism is the fact that it can be expressed using three different conventions \[^9\]:

i) State-Transition matrix or table

ii) State-Transition Functions

iii) State-Transition Diagram or Statecharts

State machines have found popular use in applications where distinguishable states exist. These state machines and diagrams are related to the arrangement and creation of compound, discrete-event systems, like real-time computer systems, digital control units and communication protocols\[^5\]. Software is a manifestation of its statechart design. Hence, Statechart notations are popular as compact and expressive diagrams for expressing complex behaviour of reactive real-time systems in a modular fashion \[^10, 11\]. Real-world reactive systems are event-driven, for e.g. telephones, automobiles, operating system, GUI interface of common software applications etc\[^5\]. To describe the reactive behaviour of these systems in a clear, realistic, formal and rigorous way automata are the best formalism.

**Automata express reactive system behaviour as the set of approved sequences of input as well as output events, conditions as well as actions with additional constraints (e.g. timing constraints) if any** \[^5\]. For e.g. a state transition can be expressed as, event \(\alpha\) occurs in state \(q_1\) when condition A is true and the system transforms to state \(q_2\), else the system remains in \(q_1\)

\[
[A]
\]

\[
q_1 \xrightarrow{\alpha} q_2
\]

Hence state machines and their extensions called finite state machines (FSM) or finite automata are the formal mechanism for expressing real-time...
reactive systems. At its simplest state diagrams are directed graphs with nodes denoting states, and labelled arrows (with triggering events and guard conditions if any) denoting transitions \cite{5, 6}. However, it is also agreed that the above concept if not managed may suffer from the state explosion problem \cite{5}. Hence to be useful, state machines must be applied as a modular, hierarchical and well-structured approach.

Technically the kernel of the state-chart approach is a directed graph. Hence, before moving on to the FSM extension of state machine we end this section with a formal definition of a graph and directed graph \cite{6}.

A graph $G(V,E)$ is a construct of two finite sets, the set $V = \{v_1, v_2, ..., v_n\}$ of vertices and the set $E = \{e_1, e_2, ..., e_n\}$ of edges. Each edge here is a pair of vertices from $V$, for e.g. $e_i = (v_j, v_k)$.

Hence $e_i$ is an edge from $v_j$ to $v_k$. If we say that edge $e_i$ is outgoing edge for $v_j$ and incoming edge for $v_k$, such a construct is formally called a directed graph.

3.2 TYPES OF STATE MACHINES

Each real-world problem is classified as computable or incomputable \cite{13}. Similarly, state machines can also be classified as Finite State Machines (FSM) or Infinite State Machines. Infinite State Machine is a state machine that doesn’t have a finite number of states \cite{13}. The Infinite here cannot be determined. Such state machines can be conceived but are not practical \cite{13}. Finite State Machines popularly called finite automata as defined in Section 3.2.1 are a device that can acquire one of a finite number of states at a given time. In certain conditions it can transit to another state. Such action is called as a transition.

Finite volume of space can contain only finite volume of information with a maximum that is in fact a function of that volume \cite{13}. In case of software, the volume implies memory which is finite, hence the amount of information is also finite. As a result, even quantum computers cannot handle infinite calculations in a finite time with finite memory.

All automata share the following characteristics \cite{6, 14}:
i. **Inputs:** Assumed as chain of symbols taken from a finite set $I$ of input signals. Namely, set $I$ is the set $\{x_1, x_2, ..., x_n\}$ where $n$ is the quantity of inputs.

ii. **Outputs:** Series of symbols taken from a finite set $Z$. Namely set $Z$ is the set $\{y_1, y_2, ..., y_m\}$ where $m$ is the quantity of outputs.

iii. **States:** Finite set $Q$, whose characterization varies with the type of state machine.

Discussion about Infinite State Machines in the software context is irrelevant. Hence, we limit our discussion here to the different kinds of Finite Automata:

### 3.2.1 Finite Automata

The first people to think about the conception of finite automata incorporated a panel of biologists, engineers, mathematicians, psychologists and several of the first computer scientists. All of them shared a general curiosity: to represent the human thinking process, either inside brain or in a computer. Warren McCulloch and Walter Pitts, neurophysiologists, were the first ones to give an explanation of finite automata in 1943\[^1\]. Their manuscript, entitled, "A Logical Calculus Immanent in Nervous Activity", added noteworthy assistance to the learning of neural network theory, theory of automata, the theory of computation, and cybernetics.

The concept of Finite Automata (FA) is fundamental to the theory of computer science. It is a specification based on the concept of state machines \[^9\]. This standard model has been utilised in the mathematical basis of computer science for e.g. in the proper design of programming languages \[^9\]. Popularly known as the deterministic finite automaton (DFA), deterministic finite state machine (DFSM), finite automata or simply automata (singular: automaton), it can be formally defined as below \[^6, 9\]:

**Definition 3.1:** A FSM or automata is a five tuple $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ where: \[^13\]

- $Q$ a finite set of states,
- $\Sigma$ an alphabet or a finite set of alphabets,
- $\delta: Q \times \Sigma \rightarrow Q$ a transition function,
- $q_0 \in Q$ a start state,
- $F \subseteq Q$ a set of final or accepting states.

In following discussion we shall quote FSM as automata. The automata $A$ is said to accept a subset of $\Sigma^*$, defining the language $L_A$:

$$L_A = \{ w \in \Sigma^* | \delta(q_0, w) \in F \} \quad (3.2)$$

State machines were known and used much before the beginning of software [9]. However, in this work we shall majorly stress on their use in describing software. An automaton is a conceptual model of a digital computer [6]. It is a well-tested framework for generating reliable software for even the most complex and complicated real-life problems [9]. An automaton is a powerful, decision-making control model that can be used to describe software under all circumstances: acceptable or unacceptable. Automaton applies the idea of state as information about its ancient history. States of an automaton symbolize the circumstances in which a state machine may possibly be in. However the possibility of an automata being in a given state is governed by a conditional probability. Because the number of situations for a finite automata are finite, hence the number of possible states is also finite and thus the name FSM. We may also call a FSM as a control machine as it determines its output along with some input by applying some logic onto it.

For a complete preface to the theory of automata, the reader is referred to [14].

3.2.1.1 Types of Finite Automata
Realistic software applications necessitate a more complicated control model. Finite Automata are powerful models that can be used to describe software behaviour in all imaginable situations [12, 15-17]. States symbolize all probable situations in which the state machine can be. Hence, it contains some type of memory: how the state machine may have acquired its current state. As software runs the state varies in a time-bound fashion, and outputs are decided considering the current state as well as the inputs.
As the amount of discernible circumstances for a given automata is finite, the quantity of states is also finite. This self-explains the word “finite” in the name\cite{6}. There exist numerous sorts of finite state machines, which can be separated into four main types \cite{6, 12}:

- **Acceptors**: Also known as recognizers and sequence detectors, these machines generate a binary output when answering whether the input can be accepted or not (deterministic state machines). All states of such an FSM are either accepting or non-accepting. When all input is processed, if the present state is an accepting state, the input is accepted; otherwise it is rejected. Notably, in an acceptor type FSM the input are symbols; actions are never used.

- **Classifiers**: A generalization comparable with acceptor generates single output on termination but can possess greater than two terminal states. Either recognize the input or do not (deterministic state machines)

- **Transducers**: Produce output based on a specified input and/or state using actions. They can be used for controlling applications. Two main categories of transducers are Moore and Mealy machines as discussed in Section 3.2.1.1a below.

- **Sequencers**: Sequencers also known as generators are a subclass that accept single-letter input alphabet. They produce only single sequence, understood as output sequence of transducer or classifier outputs.

### 3.2.1.1a Moore and Mealy Machines

A state can be represented in different forms as a state-transition diagram, a state-transition table or through transition functions. The diverse forms simply define its capability to illustrate object behaviour. Many different kinds of Finite State Machines are available to model computing machines \cite{6, 12}. However, the most popular types of state machines have been the Moore and Mealy machines\cite{6, 12}.

A computer scientist, G.H. Mealy in 1955 introduced a finite state machine where outputs correspond to transitions between states. Machines of this type became popular as Mealy machines. A Mealy machine hence is a state machine that produces only Input Actions. Another computer scientist, E.F. Moore, in his paper
in 1956 generalized the theory to greatly more powerful machines where output is determined only by the state. This machine is known as the **Moore machine**. Hence, a Moore machine can be described as a state machine that produces only entry actions. The finite-state machines, the Mealy machine and the Moore machine, were named in respect of their work. While the Mealy machine decides its outputs through the present state and the input, the Moore machine's output is based upon the present state alone. There are no suggestions as to which model is better. However, the model chosen will influence design. However, **hardware systems are better realized as Moore models whereas software is best described as a Mealy model**. The **proposed automata-based software reliability model in this work is also an extension of the Mealy model of Finite State Machines**.

From the mathematical clarification above, it can be understood that a finite-state machine contains a finite amount of states. Each distinct state recognizes a finite number of inputs, and has its own rules that depict the action of the machine for each input, characterized through the state transition mapping function.

### 3.2.1.1b Deterministic and Non-Deterministic Finite Automata

For hardware design, a state machine can serve as a vehicle for system design. In software, an automaton can act as a mechanism to establish theorems. For **hardware designs only a deterministic finite automata is required and non-deterministic finite automata is irrelevant**. However, **from the software point of view a non-deterministic finite state automata is useful from the point of proving theorems describing software behaviour**. A finite automaton where there is only single transition from each state for a specified input is deterministic state machine. However, if for a specified input a state machine recognizes several transitions from at least one of its states it is nondeterministic. **Non-deterministic finite automata are a generalization of deterministic finite state automata**. For modelling reactive systems, the class of non-deterministic finite automata offers more flexibility.
Definition 3.2: A non-deterministic finite state machine is a 5-tuple $M$ where:

$$M = (Q, \Sigma, q_0, F, \Delta) \quad (3.3)$$

Such that:

- $Q$: A set of finite states
- $\Sigma$: A finite alphabet
- $q_0$: A distinguished start state $q_0 \in Q$
- $F$: A set $F \subseteq Q$
- $\Delta$: All the possible transitions of the FSM, where $\Delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$, where $P(Q)$ is the power set of $Q$.

The set of NDFA includes, as a subset, the set of all DFA (because every DFA automatically satisfies the definition of an NDFA). This means that the following definition holds for both DFA and NDFA.

In contrast to deterministic finite automata, **nondeterministic finite automata have the advantage that they can easily help model a complex system.** Such machines can easily represent the non-determinism where some state $q$ and some symbol $a \in \Sigma$ such that the set $\Delta(q, a)$ contains more than one element. This models the software case where we may have at least two arrows leaving the state $q$ labelled with $a$.

Our automata-based reliability model utilizes probabilistic non-deterministic finite automata to model runtime software behaviour.

At the same time, an input can direct the automata to alter states. For each distinct input symbol, there is precisely single transition through each state. In addition, any 5-tuple set that is chosen by nondeterministic finite automata is also chosen by deterministic finite automata.
On considering finite automata, it is necessary to note that the mechanical process within the automata that leads to the calculation of outputs and alteration of states is not highlighted or probed into detail; it is instead considered a "black box".

Applications of finite automata are instituted in a range of subjects. They can run on languages containing Bërche automata, a finite number of words (standard case), an infinite number of words (Rabin automata), in hardware circuits and various types of trees, where input, state as well as the output is a bit vector of a fixed size\textsuperscript{[15-16]}.

3.2.2 Misconceptions Regarding Finite Automata

Finite Automata though more than 50 years old formal model for sequential behaviour is still misunderstood and misinterpreted in software despite widespread applications in hardware design \textsuperscript{[9]}. In this section we examine certain misinterpretations regarding state machines that have limited their use.

State in automata is used to store past history (last change of variables in the program modelled using the automata) of variables of the program through their current values \textsuperscript{[9]}. The most popular limitation attributed to automata usage is the state explosion phenomenon. The state explosion problem is likely in a state machine in case a machine represents a complex, concurrent or distributed software. In this case the numbers of states in the model of a system share an exponential relation to the number of components that make up the system \textsuperscript{[31]}. However, many methods have been planned to significantly avoid the state space explosion problem from hindering usage of automata for software design and model checking. Table 3.1 below highlights some notable techniques that are implemented to deal with the state explosion problem of state machines used to describe concurrent or distributed systems.
### Table 3.1: Techniques to Control State Explosion Problem in State Machines

<table>
<thead>
<tr>
<th>S.No</th>
<th>Technique</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Symbolic Model Checking using Binary Decision Diagrams (BDD)(^{[19-20]})</td>
<td>Instead of individual representation for each state, set of states represented as a BDD using Fixed point Algorithm. BDD representation was found to be exponentially smaller.</td>
</tr>
<tr>
<td>2</td>
<td>Partial Order Reduction (^{[21-22]})</td>
<td>Exploits action independence in a system using asynchronous process composition.</td>
</tr>
<tr>
<td>3</td>
<td>Bounded Model Checking (^{[23]})</td>
<td>Exploits fast Boolean Satisfactory solvers (SAT) to search counterexamples of bounded length.</td>
</tr>
<tr>
<td>4</td>
<td>SDL (^{[24]})</td>
<td>A CCITT Visual specification language, apt for communication protocols has been used as the basis for exhaustive analysis of systems with over 250 million reachable composite system states. An effective technique for validating systems of extended finite state machines.</td>
</tr>
<tr>
<td>5</td>
<td>LOTOS (Language of Temporal Ordering Specification) (^{[25]})</td>
<td>A Specification Language developed for the official explanation of OSI (Open Systems Interconnection) architecture. Is also appropriate to distributed and concurrent systems.</td>
</tr>
<tr>
<td>6</td>
<td>Spin (^{[26]})</td>
<td>Open-source software verification tool for the formal verification of multi-threaded software applications. Developed at Bell Labs in the Unix Group, 1980. It can be applied as a simulator, exhaustive verifier, proof of approximation system as well as a driver for swarm verification.</td>
</tr>
<tr>
<td>7</td>
<td>Counter-Example Guided Abstraction</td>
<td>An automatic, iterative, abstraction-refinement methodology applied to Symbolic Model Checking of large, complex software systems.</td>
</tr>
</tbody>
</table>
Refinement (CEGAR) [27]

<table>
<thead>
<tr>
<th>8</th>
<th>VFSM-Valid[28]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Implemented by Bell Labs to locate errors in network of communicating processes. The fundamental assumption of the model is to run in an implementation loop using a replica of the initial virtual input (VI) as it is when entering the loop.</td>
</tr>
</tbody>
</table>

Table 3.1 above highlights artifacts that provide high-level descriptions of complex systems (with multiple parallel processed and complex data types)[19-28]. These techniques can be used to deal with several issues related to the application of model checking like construction of state spaces and state explosion. Detailed analysis of the above techniques reveals that, concurrency is the main provider to state explosion [29]. Concurrency results in huge number of implementation sequences that may all start from the same start state and terminate at the same end state through same transition. However, the ordering of the transitions may differ resulting in different states [29]. Apart from above artifacts, choosing a coarser-level of atomicity has been proven to provide a partial solution to this problem [29]. The work proposed in this thesis is based on the utmost level of atomicity for executable software (each unique assembly language instruction transits to a new state). We argue that as the set of assembly language instructions is finite and limited in number the transitions leading to new states are finite, hence state explosion is not a grave concern as the number of possible states are never potentially infinite. Hence, the model itself achieves statecontrol to some level itself. Though we have manually verified the claim manually on small and medium-scale systems, complete verification awaits automation of the proposed model. However, we do not rule out further extension of this model with additional constraints like time[32, 33] etc to overcome the state explosion phenomenon completely for large-scale concurrent systems [29-31]. In such a scenario the proposed model would be enhanced using a finite set of real-valued clocks.

It should be further noted that the size of automata representing complex software with millions of code can be controlled by modularizing the design into sub-parts
and using platform-independent model directly in the runtime systems [9, 19-22, 29-31]. Hence, the state explosion problem though most-likely to occur in concurrent and distributed real-time systems [19-23, 28-31] does not limit the use of automata as basis of model-checking verification techniques (automatic detection of errors in complex systems) [5,9-12,17-23,29-34].

3.2.3 Finite Automata versus Turing Machine

Finite automaton is the simplest automaton considered for calculation. It can effectively calculate every primordial operations; hence, it is not a satisfactory estimation model. Further a finite automata’s incapability to simplify computations hinders its supremacy.

The following example demonstrates the distinction between a finite automata and a Turing machine:

Envision a contemporary CPU. Each bit of the device can only acquire binary states (0 or 1). Henceforth, there exist only a finite amount of probable states. Further, detailing the components of a computer that a CPU intermingles with, only finite numbers of probable inputs are possible, as of the computer's mouse, different slot cards, hard disk, keyboard etc. As a consequence, we can reason that a CPU can be represented as finite automata.

Now, envision a computer. Though each bit of a machine can only acquire two dissimilar states (0 or 1), there are unlimited number of connections inside the computer in total. It is extremely hard to represent the behaviour of a computer completely within the restriction of finite automata. However, higher-level, infinite and more potent automata would be proficient of executing out such task [29-32].

Well-known computer scientist Alan Turing envisioned the first "unbounded" (or immeasurable) model of computation. In 1936, the Turing machine was proposed, to resolve the Entscheidungs problem [8]. The Turing machine is considered as a control unit or finite automaton capable of infinite storage (memory). The
"memory" is composed of an immeasurable amount of one-dimensional array of cells. Turing's machine is effectively a conceptual model of current-day computer implementation as well as storage, to provide an accurate mathematical definition of a mechanical process or algorithm.

For an automaton to be considered finite, its representationshould contain a finite amount of functions and states with predetermined strings of input and output, infinite automata include an "addition" - any stack that can be moved right or left, and can match up the demands completed on a machine.

**Definition 3.3:** A Turing machine is formally defined by the set \([Q, \Sigma, \Gamma, \delta, q_0, B, F]\) where

- \(Q = \) finite set of states, of which one state \(q_0\) is the initial state
- \(\Sigma = \) a subset of \(\Gamma\) not including \(B\), is the set of input symbols
- \(\Gamma = \) finite set of allowable tape symbols
- \(\delta = \) the next move function, a mapping function from \(Q \times \Gamma\) to \(Q \times \Gamma \times \{L,R\}\), where L and R denote the directions left and right respectively
- \(q_0 = \) in set \(Q\) as the start state
- \(B = \) a symbol of \(\Gamma\), as the blank
- \(F \subseteq Q\) the set of final states

Thus, the main distinction among a Turing machine and finite automata is stated through the reality that the Turing machine is competent of altering symbols given on its tape and replicating program implementation and storage. For the above reason, it can be concluded that the Turing Machine has the authority to model most calculations that can be computed today through contemporary computers.

However, Turing machine has not been taken as the base for this work because Turing Machine Model provides a conceptual representation of an algorithm\[^{22}\]. Real computers are capable of performing much more complex calculations than just moving the head, altering the present symbol and state. Hence, Turing Machine model is a model excellent enough only to be applied as a benchmark to evaluate varied algorithms and computation problems\[^{22}\].
3.3 PROBABILISTIC FINITE AUTOMATA

Probabilistic Finite Automata (PFA) model has been extensively investigated in numerous scientific fields [15-16]. They are an interesting model that has been used in a range of areas like machine learning, pattern recognition, time series analysis, computational biology, computational linguistics etc [15]. Due to this, discussions of the PFA models use widely differing terminology and notation. However, these models are considered as potential models by researchers to design good learning algorithms [15].

We use the machine learning terminology for this discussion. A PFA is formally defined as follows [15]:

**Definition 3.4:** A PFA is a tuple $A = \langle Q_A, \Sigma, \delta_A, I_A, P_A \rangle$ where:

- $Q_A$ is a finite set of states,
- $\Sigma$ is a finite alphabet,
- $\delta_A \subseteq Q_A \times \Sigma \times Q_A \rightarrow Q$ is a transition function,
- $I_A: Q_A \rightarrow [0,1]$ are the initial state probabilities,
- $P_A: \delta_A \rightarrow [0,1]$ are the transition probabilities,
- $F_A: Q_A \rightarrow [0,1]$ are the final state probabilities.

PFA is an extension of FA where the probabilistic information is incorporated at the suitable places. For e.g. a transition $q_i \stackrel{i}{\rightarrow} \hat{q}_i$ signifies that if the machine is in state $q_i$ and receives input $i$ then the machine moves to the next state $\hat{q}_i$ with a probability including the interval $\hat{p}$. Further there may also be certain constraints on such transitions. For e.g.:

- A probability $\hat{p} = [p, 1]$ can emerge in a transition $q \stackrel{i}{\rightarrow} \hat{q}, \hat{q} \in \delta$ only if it is a unique transition between $q$ and $i$. If there would be present two possible two transitions say $q_i \stackrel{i}{\rightarrow} \hat{p}_i \hat{q}_i$ and $q_i \stackrel{i}{\rightarrow} \hat{p'} \hat{q}_i$ and probability $\hat{p}$ be 1, then the probability associated with $\hat{p'}$ will be 0, which is forbidden.
• Real probabilities for each state $q \in Q$ should add to 1. This will only be possible if 1 lies between the lower and upper bound of the symbolic probabilities.

Success of the PFA in various fields has made them valuable indeed \cite{15, 16}. We further propose that PFA is the most suitable model for representing runtime software. In the following sections we propose a PFA based reliability control and monitoring tool we call Automata-Based Software Reliability Estimation Model.

3.3.1 Approaches for Probability Calculation in PFSM

There exist many approaches for probability assignment and calculation in PFSM. However, Markov Chain approach is most suitable to represent runtime software \cite{34} instance.

A Markov chain is a special type of mathematical system that uses a stochastic process to undergo transitions from one state to another, among a finite quantity of possible states \cite{35}. It is a random and memoryless process where the next state depends on the current state and not the sequence of events preceding it. This is called the Markov property.

Formally, a markov chain can be defined as a random evolution in a state space $Q$ such that \cite{35}:

i. At time 0 the state of the chain is distributed according to $\mu$.

ii. If at time $t$ the state of the chain is $q_i$, then at time $t+1$ it will make a random jump which will end up at $q_j$ with probability $\pi(q_i, q_j)$ (for every $q_j \in Q$).

Named after the Russian mathematician A.A. Markov (1856-1922), a Markov Chain can be easily maintained using a transition matrix, $P$ with the following characteristics:

i. A Transition Matrix is a square matrix, as all possible states are used as both rows and columns.

ii. All entries are between 0 and 1 (inclusive) and represent transition probability from the state at left to a state indicated across the top.
iii. Sum of entries in any row must be 1.

Markov Chains have widespread applications in varied domains and have also been applied as a basis for many architecture-based network reliability estimation models [3,36-38].

As part of this work we have used the properties of a Markov Chain to represent the software structure at runtime and track software state to state transition. Notably, each runtime software instance can be represented as a Markov chain if:

i. Each transition is one of a set of discrete states of the Markov Chain

ii. The outcome of a transition depends only on the current state of the software and not on any of its past states.

Preceding sections discuss the use of a Markov-chain as the foundation to achieve a automata-based software reliability control model.

3.4 SUMMARY

This chapter summarises the fundamental concept of an automata and its types. The chapter delves into the basic foundations of our automata-based reliability model.
REFERENCES