CHAPTER 4

AUTOMATA-BASED SOFTWARE RELIABILITY MODEL

The chapter proposes an algorithm based on probabilistic finite state automata (markov chain) for software reliability monitoring and control during software execution. The automata-based software representation in this case is obtained using opcode instructions from executable software code.

All software during execution is an automaton. It contains discrete sets of input, along with possibly extra parameters in input to retain state information. If the software executes as an automaton then its reliability is analogous to the reliability of the automata representing the software at runtime. Software Reliability here implies the continuity of correct service delivery as expected by the customer. It is notable here that, as long as software executes in a particular operational profile with no changes to the software or its execution environment it is expected to have identical outcomes. Hence instead of using the term reliability for software we are actually looking for probabilistic correctness. In the following discussion we propose an automata-based software reliability model that can ensure probabilistic correct software operation during each execution instance.

4.1 WHY AUTOMATA-BASED REPRESENTATION

A state machine or automata is not a heuristic model that can be interpreted in different ways according to user choice. Instead the model has a sound mathematical and theoretical basis. There are many available verification methods to prove correctness of software built as a system of state machines. To support our claim Table 4.1 below lists some realistic examples of the same.
<table>
<thead>
<tr>
<th>S.No</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hoare’s Communicating Sequential Processes (CSP) [5]</td>
<td>Formal Language to describe interaction patterns for concurrent systems. First described by C.A.R Hoare in 1978 was influential in designing of programming languages like Occam, Limbo and Go [5]. It has also been applied to specification and verification of a number of different concurrent systems such as T9000 Transputer, Secure Ecommerce System etc [5].</td>
</tr>
<tr>
<td>2</td>
<td>Replicated State Machines (RSMs) [6]</td>
<td>General approach for implementing fault tolerant service via server replication. In this case, client interacts with server replicas enabling physical and electrical processor isolation leading to independent server failures. Approach has been widely used for implementing fault tolerant services by replicating state machines in multiple distributed servers [6].</td>
</tr>
<tr>
<td>3</td>
<td>Interacting State Machines (ISM) [7]</td>
<td>High-level variant of Input/Output Automata utilized in abstract modeling and verification of reactive systems. Popularly used to describe system model and represent them graphically with AutoFocus tool [7]. Has been applied to formal system analysis of variety of applications like the LKW model of Infineon SLE 66 smart card chip, Needham-Schroeder Public Key Protocol [7].</td>
</tr>
<tr>
<td>4</td>
<td>Extended State Machines (ESM) [8]</td>
<td>Extension of traditional Finite State Machine that allows addition of both new states and events to have new behaviour.</td>
</tr>
<tr>
<td>5</td>
<td>Real-Time Operating Systems (RTOS) [9]</td>
<td>Introduced to handle the complexity of real-time control systems through use of APIs to describe desirable system behaviour.</td>
</tr>
</tbody>
</table>
Perhaps state machines are the oldest known formal model that allows the designer to authenticate a control system and produce reliable control software [3]. However, the state machines are not the only available model [11-12]. Other models like petrinets and temporal logic are also popular alternatives for control system verification [3-4,11]. Though all the available models describe the same problem, they are not similar and each have their own limitations. Table 4.2 below compares state machines with petrinets and temporal logic.

Table 4.2: State Machines versus Petrinets and Temporal Logic

<table>
<thead>
<tr>
<th>S.No</th>
<th>State Machine</th>
<th>PetriNet</th>
<th>Temporal Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Oldest recognized formal model to describe sequential behaviour and verify control systems [3]. Also known as Finite State Machine or Finite State Automaton (plural automata).</td>
<td>Popularly called place/transition net or P/T net, one of numerous mathematical modelling languages for representing distributed systems [11-12].</td>
<td>Popular mechanism for specifying and reasoning a concurrent Program [4]. A mathematical way to express properties of a system over time. Hence, also called linear time logic.</td>
</tr>
<tr>
<td>2</td>
<td>Mathematical model of computation for designing computer</td>
<td>Visual mathematical model to illustrate information flow</td>
<td>Simple extension of classical propositional logic to describe</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>programs as well as sequential logic circuits.</td>
<td>proposed by Carl Adam Petri in 1962(^{[11]}).</td>
<td>temporal behaviour of a program through hierarchical specification.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Helps visualize behaviour of sequential control systems</td>
<td>Helps visualize and analyse behaviour of asynchronous concurrent systems.</td>
<td>Developed initially to reason about ordering of events in time for concurrent programs</td>
</tr>
<tr>
<td>4</td>
<td>Any particular FSM is consists of a finite number of states and the triggering condition that results on transition between states.</td>
<td>Bi-partite graph with two classes of non-zero, finite nodes of places and transitions.</td>
<td>Is a system of rules and symbolism to represent and reason for propositions in terms of time.</td>
</tr>
</tbody>
</table>
| 5 | 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. | Represented as, tuple \( S= (P, T, I, O, u) \) with  
- \( P \): Finite set of places  
- \( T \): Finite set of Transitions  
- \( I \): Finite set of arcs from places to transitions  
- \( O \): Finite set of arcs from transitions to places  
- \( u \): Integer vector representing the current marking | Let \( T=(T, \prec) \) be a flow of time, a valuation on \( T \) is a map \( \pi : (T \rightarrow (\Phi = \{0,1\})) \), here \( \Phi \) denotes set of propositional variables. A model \( M \) is a pair \( M=(T, \pi) \) consisting of a flow of time and valuation. |
<p>| 6 | Performs well for small and medium scale systems, can | Representation power greater than a Finite State Machine for | Applied in formal verification to state the requirements of |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>prove to be computationally expensive for large-scale complex systems.</td>
<td>complex systems</td>
</tr>
<tr>
<td>7</td>
<td>State space may get exponentially large in case of complex, concurrent systems leading to state explosion problem.</td>
<td>Presents a much more compressed state space as compared to finite automata, exploits hierarchical structure of a complex system to reduce states at any level of abstraction [11].</td>
</tr>
<tr>
<td>8</td>
<td>Common component of VLSI circuits, network protocols etc.</td>
<td>Commonly applied to concurrent programming, process modelling, reliability engineering, simulation etc.</td>
</tr>
<tr>
<td>9</td>
<td>Many varied FSM models exist. Especially Buchi automata, Timed Automata and Zielonka automata have found wide use in distributed, concurrent systems[11-13].</td>
<td>Many variations to the standard petrinet model are also available like ColoredPetrinet, Timed Petrinets, PrioritisedPetrinetsetc[13].</td>
</tr>
<tr>
<td>10</td>
<td>Variants of FSM like Variants of Petrinets</td>
<td>LTL formulas must be</td>
</tr>
</tbody>
</table>
Timed automata have found successful use in modelling real-time, industrial systems have been found to be undecidable for timed language equivalence and universality problem[13].

Likewise, Timed Petri Nets suitable for analysis of real-time systems are also undecidable for verification problems like reachability and boundedness[13].

Known before the reduced state space generation for any software is commenced, also use of temporal operators “next state” is prohibited in many cases.

To explain the relation between the above three popular alternatives for control system verification, we can state that: A concurrent program can be viewed as a set of communicating FSMs which can be converted to a Petri Net through available techniques and correctness can be verified using Temporal Logic specifications[11-12]. However, it may be noted that the effectiveness of any of the above formal method may diminish as the size of the software object increases as in current concurrent and distributed systems[11]. Thus it is difficult to apply a single formal method to a complete software system. Therefore, engineers look towards development of compositional methods that verify different parts of the code separately, and then make conclusions about the software as a whole.

The state machine however remains the preferred choice for modelling the behaviour of any software control system[4]. As software contains some control task or the other, state machine models are apt for representation of modern software. There however exists no reasonable proof for the belief that coding software without any formal model would produce better software[3]. In the present day application software are event-driven. This implies that in principle, software simply waits for an event to occur. In case event occurs software reacts to it, otherwise it continues in the same state. In this case an automata model is the most appropriate to represent software behaviour. However, this class evades software that continuously checks for some user input[12]. Creating large-scale
software working in a well-organized fashion can be correctly managed with finite state machines and its different variants [13].

In this work, we have modelled finite state machines using executable software opcode to achieve reliable execution of runtime software. This would definitely be a significant achievement in reliable software production as this would allow invoking self-learning and self-healing modules in software.

4.2 NEED OF RUNTIME SOFTWARE REPRESENTATION

Current software reliability modelling approaches makes their own set of assumptions for simplifying software reliability estimation [14-18]. However, in this age of embedded, mobile, pervasive, distributed software characterized by their indefinite operational profiles and erratic runtime conditions the traditional models are inappropriate [19-21]. To meet the challenges of the expanding software industry and avoid losses from failure, a reliability estimation framework incorporating inherent complexity and uncertainty of modern software in a dynamic setting is required. A runtime architecture-centric reliability estimation approach to ensure reliability monitoring at runtime shall serve the above purpose. Traditional reliability estimation models rely primarily on past-failure history data to predict future software reliability [18]. In contrast, runtime reliability estimation uses information like software execution profile, software architecture etc to estimate software reliability.

To overcome software failure and the damage caused by it, a proactive approach for software reliability estimation is required. Such a model should be capable of forecasting failure and take appropriate actions to overcome the same. Proactive software reliability estimation requires proactive action by evaluating software performance during execution. Traditional models estimate reliability statically in the development phase using post-failure data fitted to certain assumptions. To overcome limitations of traditional reliability estimation we propose a state-based, path-based software reliability estimation model. This model shall help combine modelling power with operational data representativeness.

As discussed in chapter 2, none of the conventional reliability model assumptions actually represent the actual nature of software execution. As a result accurate
software reliability estimation still remains an open challenge. To solve this problem we first need to understand that software execution is actually input \(\rightarrow\) program \(\rightarrow\) output \([22]\). This implies that for some defined input the program processes the input to produce the desirable output. Hence, software reliability is basically a function of correct I/O Pair \(<i, o>\) at runtime. Thus runtime reliability estimation is the true estimate for software reliability. Formally, **runtime reliability estimation is the use of runtime software representation for reliability estimation** \([19]\). As software reliability is a dynamic system attribute that alters at runtime. Runtime reliability estimation focuses on software behavior at runtime. This kind of software analysis allows changes to be made to execution and thus allows prevention of fault execution. The technique implies enhancing runtime software to perform tasks like software state investigation and reliability estimation \([20]\).

**4.3 AUTOMATA-BASED SOFTWARE REPRESENTATION USING OPCODE**

Modern software aid us in a variety of different tasks ranging from weather monitoring, air traffic control, banking transactions to computer gaming and photograph retouching \([20-27]\). But amazingly the **mechanics of software execution for any purpose is similar in base functioning** \([23-28]\). All software code is executed as a bunch of bytes. This set of bytes is **machine language instructions** (binary codes) that the computer executes. The general format of each machine language instruction is constituted using **opcode and operands** \([30-31]\). The **operands can be a memory address, a register or a value** \([30-31]\). **Opcode is short for operational code.** At its simplest it is a **portion of machine language instruction that specifies the operation to be performed** \([31]\). Opcode are the heart of machine language instruction set. Interestingly opcodes can also be found in bytecodes of Java class files, bytecodes of compiled LISP code, .NET common Intermediate Language and many other programming languages \([32-36]\). Further for each of these programming languages the opcode set is a small, finite instruction set. For Ex- Java bytecode is constituted of approximately 147 opcode instructions \([21]\). Hence it may be noted that the opcode instruction set can be considered finite and manageable.
We propose that as opcode controls software execution, it can also be used as the basis to ensure fault-free software execution. The opcode sequences in program code drive software transition from one state to another. Utilizing this basic attribute of software operation we propose to control software reliability. For implementation of the same we propose our automata-based software reliability model in the next section.

4.4 PROPOSED AUTOMATA-BASED SOFTWARE RELIABILITY MODEL

Software executes as a system of finite states. Every software state has a probability of transition either to the next correct state or incorrect state. Hence software is a probabilistic system. To ensure reliable operation, we need a formal model to analyze such asynchronous programs with discrete probabilistic choices. To accurately control reliable or fault-free operation of such a system we propose the use of probabilistic automata.

In theoretical computer science, the automaton or finite state machine (FSM) is a mathematical model of computation [15]. It has been used to design both computer programs and sequential logic circuits for long [16, 22-24]. It is actually an abstract machine that can be in any one of a finite number of states at any given time.

The above formalism can be used as a basis to monitor software at runtime. Executable software is an automaton which on receiving an input string ‘a’ may transit from its current state to the next state. If the next state, q’ belongs to Q and the software finally terminates in some state q belongs to F (q_f ∈ F) then the software is 100% reliable. However, if at any point during its execution the next state q’ does not belong to F (q_f ∉ F) then the software is executing a fault and is 0% reliable. Hence the runtime model of software can be represented through a probabilistic finite automata model. The probabilistic automaton is also a quintuple like an ordinary automaton [15]. However, it is different as here the transition function δ is defined as [15]:

\[ δ: Q × Σ → P(Q) \]
Here, \(P(Q)\) denotes power set of \(Q\). The above transition function can be expressed as a membership function \[15\]

\[
\delta : Q \times \Sigma \times Q \rightarrow \{0,1\}(4.2)
\]

such that

\[
\delta(q,a,q') = 1 \text{ if } q' \in \delta(q,a)(4.3)
\]

and

\[
\delta(q,a,q') = 0 \text{ if } q' \notin \delta(q,a)(4.4)
\]

Hence in a probabilistic automaton the target of a transition is a probabilistic choice over several next states. For instance, a transition may reach the correct next state with probability of \(\frac{1}{2}\) and incorrect state with probability \(\frac{1}{2}\) too \[15\]. Thus in probabilistic automaton a transition relates a state and an action to a probability distribution over a set of states.

On basis of the above discussion, we now derive an approach to software reliability estimation based on probabilistic automata. The approach provides a model that is simple, formally sound and practically useful. The model permits the tracing and control of next possible software state. The information can then be used to ensure failure-free software execution and analyze software feasibility.

The conventional models for software reliability estimation quantify reliability as the absence of failures from a system. Contrastingly they compute reliability using some kind of failure data (brute force) \[23\].

Software Architecture is a key means for achieving understandability of the complex, real-time software systems \[24\]. In present times when real-time software is expected to perform despite faults, a finite state-based software representation scheme is used to represent and control software execution. This state-based software representation is built using actual executable code. Hence, it showcases the actual states software can acquire during its operation along with the possible paths to the final desirable as well undesirable states.

While the total input space of a software system can be represented as an automaton, at no particular execution will software traverse all states of the automaton. During each execution software shall execute a subset of connected states from the set of states constituting the automata. This subset of connected states is what is technically called a “path” in software engineering. A path can be formally defined as \[1\].
**Definition 4.1:**
A **path** is a sequence of states through software, from software entry to successful termination or failure.

**Definition 4.2:**
A **state** here may represent a single statement or a sequence of statements with a single entry point and a single exit point.

It is important to note here that automata representation for software may contain ‘n’ possible paths (software code may contain loops and recurring blocks of code). However we are not interested in estimating the number of paths in the software. For reliability calculation we are only interested in monitoring the software path of execution.

Real-time software can be considered as a large program which is a collection of logically independent modules/classes. A module in this case is a compilable unit that can perform a specific function. Reliability of such a module can be defined as the probability that the module performs its functions correctly, i.e. produces desired output and transfers control of execution to the next module. For each specific user input, set of modules will be executed. This interaction between modules may also lead to introduction of the state explosion problem. However, may techniques as discussed in Section 4.1 exist to reduce the number of states to a manageable number. However, Reliability of output will depend on the reliability of individual modules.

**Assumptions**
- **Reliabilities of modules are independent:** Errors do not compensate each other; incorrect output of a module will not be corrected later by subsequent modules being executed[25].
- **Transfer of controls among program modules is a Markov Process:** The next module to be executed will only depend on the present module and is completely independent of the past history[26]

**4.4.1 Automata-based Software Reliability Model**
The runtime structure of a program (each software module) is represented as automata where each node \( q_i \) represents a software state and a directed arc
\((q_i, q_j)\) represents a possible transfer of control from \(q_i\) to \(q_j\) upon execution of software opcode which constitutes the transition function. To every directed arc \((q_i, q_j)\) we attach probability \(P_{ij}\) as the probability that transitions \((q_i, q_j)\) will occur when software is at node \(q_i\). The transition probability represents the characteristic of decision branch at exit point of state \(q_i\) which shall decide whether the transition is to \(q_j\) or Failure state \(e\).

Let \(R_i\) be the reliability of node \(q_i\).

Without loss of generality we assume that the program has a single entry node and a single exit node. Each node of the graph is considered as a state of the Markov process, with initial state corresponding to the entry node of the automata \(^{[26]}\). We add two other states, namely, \(F\) as the terminal state representing the state of correct output and \(e\) as the state of failure.

For each node a directed arc \((q_i, e)\) is created with a transition probability \(1 - R_i\), representing occurrence of an error in execution of state \(q_i\). As errors do not compensate each other, failure at any \(q_i\) will lead to an incorrect system output, regardless of the sequence of opcodes executed later. This is marked as transition to the terminal state \(e\). In such a case the original transition probabilities between nodes \(q_i\) and \(q_j\) is modified into \(R_iP_{ij}\), which represents the probability that the execution of module under analysis produces the correct result and control is transferred to next module. For the exit node \(F\) a directed branch \((q_n, F)\) is created with transition probability \(R_n\) to represent correct transition at exit node.

To achieve this software representation, we propose an automata-based software reliability model.

The notations used in the model are described in Table 4.3 below:
Table 4.3: Notations for the Automata-Based Software Reliability Model

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G(V,E)</td>
<td>Graph Representation of executable software as a set of V nodes and E links.</td>
</tr>
<tr>
<td>R(Q)</td>
<td>Reliability of a software as a set of Q components or nodes</td>
</tr>
<tr>
<td>F(I,N)</td>
<td>Function that calculates next software state using previous state information (N) and assembly opcode (I)</td>
</tr>
<tr>
<td>x_i</td>
<td>Input node i</td>
</tr>
<tr>
<td>P_ij</td>
<td>Weight b/w node x_i and x_j</td>
</tr>
<tr>
<td>Q(i)</td>
<td>A distinct collection of nodes through the FSM from the start node to the final node.</td>
</tr>
</tbody>
</table>

The primary goal of this automata-based software reliability model is to provide automated support for model construction and next state knowledge base generation. The model controls software execution using the fact that future or next software state depend on the present state and input instruction\(^{[26]}\). The model uses the above model and data to maximize runtime software reliability.

The algorithm extends the usage of stochastic finite state automata (Markov chain) formalism for runtime software representation and control. The stochastic finite automaton model obtained utilizes the rules laid in eqn. (1-4) above, to monitor the next software transition.

Table 4.4: Phases of the Automata-Based Software Reliability Model

<table>
<thead>
<tr>
<th>Phase I:</th>
<th>FSM Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Start</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Step 0:</strong></td>
<td>Preprocess all executable code of software under scrutiny by transforming their values to equivalent assembly code using disassembly tool.</td>
</tr>
<tr>
<td><strong>Step 1:</strong></td>
<td>Extract opcode from each assembly instruction. Repeat steps 2-4 for each assembly opcode until end of executable code file</td>
</tr>
<tr>
<td><strong>Step 2:</strong></td>
<td>For each unique assembly opcode instruction, record the opcode instruction and its corresponding node to the next state transition table.</td>
</tr>
<tr>
<td><strong>Step 3:</strong></td>
<td>Represent each unique assembly opcode as a new, unique automata node,</td>
</tr>
<tr>
<td>Step 4:</td>
<td>Check for end of file; assign it as the final node of the FSM.</td>
</tr>
<tr>
<td>Step 5:</td>
<td>Introduce an error node in the FSM, linked to all existing nodes to error node.</td>
</tr>
<tr>
<td>Step 6:</td>
<td>Assign each node of automata an equal probability of execution $V_{ij}$ where $V_{ij} \rightarrow {0,1}$ according to eqn. (4.2)</td>
</tr>
<tr>
<td>Step 7:</td>
<td>Using the Probabilistic Automata, obtain a connection matrix for each software module. The square connection matrix represents each node as row as well as column and represents a direct transition between two nodes with a 1.</td>
</tr>
<tr>
<td>Step 8:</td>
<td>Replace the 1’s in the transition matrix with the transition probability node values as depicted in probabilistic automata. This reliability matrix can now be used for reliability control as well as maintenance during software execution.</td>
</tr>
</tbody>
</table>

**Phase II: Software Implementation**

*(Feed Forward)*

| Step 9: | Receive input node $q_i$ and assembly opcode, $a_i$. |
| Step 10: | Validate next node from next_state transition table using eqn. (3) and (4). If (3) is true allow transition to next node. |
| Step 11: | Increase the probability of execution of last traversed node by a unit (smallest, atomic probability value assigned to each node). |
| Step 12: | If eqn. (4) is true, halt system execution. Set the probability of execution, $V_{ij}$ of node $q_i$ resulting in error node as 0 and record it to faulty_node table. |

**Phase III: Fault Avoidance**

| Step 13: | Repeat steps 12-13 till last executable instruction. |
| Step 14: | If step 10 executes, check for alternate next node using Dijkstra’s algorithm [20]. Else, let the software execute. |
| Step 15: | Continue software execution using the next alternate node. |

**Phase IV: Software Maintenance**

| Step 16: | Update reliability matrix values for each node traversed during execution. |

### 4.5 RELIABILITY CALCULATION

The final step in this model would be computation of software reliability by the software at any point during its life. In order to do so, we require a reliability model that can extract software reliability from the Probabilistic automata model. For this purpose, we found the Cheung model [28] to be an appropriate approach for our aim. This model computes the software reliability as a function of the deterministic properties of the structure of the program and the properties of component utilization and failure following a Discrete Time Markov Chain Model. The Cheung model [28] uses software specifications for early reliability detection of software. The model takes different possible software modules using software specification as the basis and uses the program structure obtained for
reliability estimation. Taking clue from the proposed approach, we used our probabilistic automata software representation to construct a reliability matrix for runtime software reliability estimation. We now discuss usage of this matrix for reliability calculation at any runtime instance.

The reliability of a program is the probability of reaching the terminal state \( F \) from the initial state \( q_0 \) of the automata. We calculate program reliability using the following procedure.

Let \( \{q_0, q_1, \ldots, q_n, F, e\} \) be the set of nodes in the automata with \( q_0 \) as the entry node and \( P_{ij} \) as the transition probability of the branch \((q_i, q_j)\). \( P_{ij} = 0 \) if the branch \((q_i, q_j)\) does not exist. The states of the Markov model are \( \{q_0, q_1, \ldots, q_n, F, e\} \). Let the transition matrix be \( \bar{P} \), as shown below:

\[ \bar{P} = \begin{pmatrix} q_0 & q_1 & \ldots & q_n & F & e \\ q_0 & 1 & 0 & \ldots & 0 & 0 \\ q_1 & 0 & 1 & \ldots & 0 & 1-R_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ q_n & 0 & 1-R_{n-1} & \ldots & 0 & 1-R_{n-1} \\ F & 0 & 1-R_n & \ldots & R_i P_{i(n-1,n)} & 1-R_n \\ e & R_n & 0 & \ldots & 0 & 0 \end{pmatrix} \]

**Fig 4.1: Transition Matrix, \( \bar{P} \)**

In the above transition matrix \( \bar{P} \), \( \bar{P}(i,j) \) will represents the probability of transition from state \( q_i \) to state \( q_j \) of the software. Let \( M \) be the matrix obtained from \( \bar{P} \), by deleting all the rows and columns corresponding to the final states \( F \) and \( e \). For any positive integer \( n \) let the \( n \)th power of \( \bar{P} \) be \( \bar{P}^n \).

Evidently, \( \bar{P}^n(i,j) \) is the probability that starting from \( i \), the chain enters the final state \( q_j \in \{F, e\} \) at or before the \( n \)th step. Hence, the reliability of reaching state \( F \) from the initial state \( q_0 \) can be represented as:

\[ R = \bar{P}^n(q_0, F) (4.5) \]

Let \( S \) be an \( n \) by \( n \) matrix such that
$$S = I + M + M^2 + M^3 + \cdots = \sum_{k=0}^{\infty} M^k$$  (4.6)

If $Q$ is finite, which is the case here, and $W=I-M$, it can be shown that

$$S = W^{-1} = (I - M)^{-1}$$  (4.7)

From the above it can be easily shown that:

$$R = S(1,n)R_n$$  (4.8)

4.6 SUMMARY

This chapter summarizes our automata-based reliability model for software reliability control at runtime.
REFERENCES