A Tool for Runtime Reliability Estimation and Control Using Automata Based Software Reliability Model

RITIKA WASON*, P. AHMED* AND M. QASIM RAFIG**

*School of Engineering and Technology, Sharda University, Greater Noida
**Department of Computer Science, Aligarh Muslim University, Aligarh
INDIA
rit_2282@yahoo.co.in, pervez.ahmed@rediffmail.com, mqrafiq@hotmail.com

Abstract: -The paper presents a novel approach for runtime reliability estimation of executable software during software execution. The proposed model does not depend on post failure data analysis whose accuracy depends on assumptions about some dubious statistical distribution. Instead the model captures required estimates during runtime from state to state transition. Thus, the model computes software reliability from the actual path taken by the software at runtime.


1 Introduction

Faults due to environmental constraints are inevitable in any software [1]. The process of software development and implementation is itself prone to errors of different nature [1]. In such status quo, software reliability has always been a problematic issue for software developers. Despite numerous reliability estimation and prediction models, software continues to operate unreliably. This proves the inaccuracy and inappropriateness of the existing reliability estimation and prediction models [2]. Today when critical, real-time systems have grown exponentially in size and complexity, its reliability is no longer an expectation but a necessity. Hence, researchers are looking for some reliable software reliability estimation models.

Continuous software operation despite the presence of faults (fault tolerance) is the new age expectation from what we call reliable software. However all conventional software reliability estimation models have failed to overcome software failure [1]. The major reason for this being the absence of a rigid formal form of software representation that can be tested and verified during software execution. Further all existing reliability estimation models quantitatively estimate how many times a system would perform correctly or fail during execution. We argue that for each execution, software shall either fail or perform as expected. When software shall execute a fault, software reliability becomes zero, as the system failed to produce its expected output. Similarly if the software terminates with the desired output, it is 100% reliable. In real world these are the only two possibilities for software systems. Then, why should we juggle with complex mathematical and statistical parameters to estimate numbers which are useless in reality.

This paper proposes an automata-based software reliability model that uses finite state machine (FSM) representation of executable software to estimate system reliability at any point of time during software life cycle. All software execute as a finite state machine (FSM). Hence, an FSM representation of executable software is the most appropriate representation of the various system states and the transitions resulting in state change.

We propose that as software executes as an automata, hence, we should find the reliability of the automata itself. The broad goal of our research is to make available an automata-based reliability estimation model that can be used across the software life cycle to ensure that the software performs reliably at any point of time despite errors in the system. We also aim to control the system from executing such faults.

The rest of the paper is organized as follows: Section II discusses how runtime reliability estimation is the more appropriate estimate for software system reliability. Section III proposes and explains the automata-based reliability model that uses FSM-representation of executable software for...
reliability estimation. Section IV elaborates the application of the above model as a control tool to ensure uninhibited software operation despite the occurrence of faults. Section V evaluates the pros and cons of the proposed model.

2 Runtime Reliability Estimation
Runtime reliability estimation refers to the technique of using the runtime software representation for estimation of system reliability. Software Reliability is dynamic; hence, runtime software is the most appropriate source for software reliability estimation to ensure continuous software performance.

Runtime reliability estimation implies runtime system analysis that only focuses on program behavior exhibited at runtime and is hence more accurate. Further dynamic analysis allows runtime changes to be made during execution and hence enables sophisticated tasks like prevention of fault execution. For Software there can be no constant reliability estimate for a system. To accurately estimate and maintain software reliability runtime support is essential [5]. Runtime support refers to the technique to extend the runtime software system with more functionality for reliability-oriented tasks such as reliability estimation and maintenance, runtime state analysis and bug prevention during the software development and deployment cycle [5].

The fundamental reason behind the profound software reliability challenge is the fact that all existing software reliability estimation measures lack a rigid formal representation that can be verified during runtime. Different formal methods that apply theoretical computer science fundamentals to solve difficult software problems have surfaced to address our software reliability concerns. It is a well-known fact that formal methods for software verification can surely help improve software reliability [6].

Reliability is the probability of software failure during a specified period of time in a specified environment [1, 2]. For long software reliability has been an important area of research. Several models for software reliability estimation have been known to exist for long [7]. However, most of these models are time-based models, i.e. they ignore program structure and consider time-dependent failure data for estimating software reliability. Inaccuracy of all above models in most real-time situations is well-demonstrated. Further there is no generic model that can give accurate reliability estimates on all software systems at all instances of time.

3 Automata-Based Software Reliability Model
Existing reliability estimation models attempt to predict software reliability without any regard to the actual software structure during execution. Though the conventional models estimate reliability using certain statistical distributions, however, they lack any distributions to ensure fault free software operation. All the existing reliability estimation models define reliability as the absence of failures from a system. However, contrastingly they quantify reliability using some kind of failure data (brute force) [8].

We hypothesize that in present times when fault tolerance is the expectation from most real-life software, a finite state-based software representation scheme can easily be used to represent as well as control software execution. Such a state-based software representation shall be a more meaningful representation of actual software structure if it’s built using actual executable code. In such case, the software representation shall showcase the actual states software can acquire during its operation along with the possible paths to the final desirable as well undesirable states.

In order to achieve this, we propose an automata-based software reliability model. The various phases of this model are enlisted in table 1 below:
Table 1: Automata-Based Software Reliability Model

<table>
<thead>
<tr>
<th>Phase</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>FSM Model</td>
<td>Input: assembly code of executable software; Output: i. Finite state machine representation. ii. Next_State Transition Table iii. Convert finite state machine representation as obtained in (i), to that of a stochastic FSM by initially assigning an equal probability of occurrence to each node of the FSM (uniform probability of execution).</td>
</tr>
<tr>
<td>II</td>
<td>Software Implementation</td>
<td>Allow software execution in reference to Next_State Transition Table. For every node the software traverses; increase the probability of occurrence of the node by a unit. If the next transition is resulting in an error node, halt system execution, mark the next node with a probability 0 and write it to the Faulty_Node table.</td>
</tr>
<tr>
<td>III</td>
<td>Fault Tolerance</td>
<td>Select an alternate node accessible from the previous node where the system was halted (use Dijkstra’s Algorithm). Continue software execution at the alternate path chosen. Repeat this step till the system does not terminate at the final output state.</td>
</tr>
<tr>
<td>IV</td>
<td>Reliability Maintenance</td>
<td>Write the path taken by the software from initial to final node to the Alternate_Path Table for future software re-runs.</td>
</tr>
</tbody>
</table>

The phases for the automata-based reliability model as suggested in Table 1, if implemented as a software can help maintain reliable software operation. However, we are still working with the actual implementation of this model. To realize the above model, we consider a class menu.java, which is a part of Java-based Chart Generator Application.

```java
public class menu extends JFrame implements ActionListener {
    public menu() {
        setTitle("Menu");
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public static void main(String[] args) {
        new menu();
    }

    JButton button1 = new JButton("Button 1");
    JButton button2 = new JButton("Button 2");
    JButton button3 = new JButton("Button 3");
    JButton button4 = new JButton("Button 4");

    button1.addActionListener(this);
    button2.addActionListener(this);
    button3.addActionListener(this);
    button4.addActionListener(this);

    menu menuFrame = new menu();
    menuFrame.setVisible(true);
}

public void actionPerformed(ActionEvent ae) {
    if (ae.getSource() == button1) {
        menuFrame.setTitle("Button 1 Pressed");
        menuFrame.setVisible(true);
    } else if (ae.getSource() == button2) {
        menuFrame.setTitle("Button 2 Pressed");
        menuFrame.setVisible(true);
    } else if (ae.getSource() == button3) {
        menuFrame.setTitle("Button 3 Pressed");
        menuFrame.setVisible(true);
    } else if (ae.getSource() == button4) {
        menuFrame.setTitle("Button 4 Pressed");
        menuFrame.setVisible(true);
    }
}
```

Fig 1: Source Code for menu (menu.java)

Compiling the above code results in menu.class. This file contains the actual executable code in bytecode form. We disassemble this bytecode to its equivalent assembly instruction set. Using the above set we construct an FSM for menu.class. Figure 2 below depicts the same.
A careful look at Figure 2 reveals that each assembly instruction results in software transition to a specific state of the FSM. This is further maintained for menu class in Table 2 below:

Table 2: Next State Transition Table of menu

<table>
<thead>
<tr>
<th>S.No</th>
<th>Transition Instruction</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aload</td>
<td>q_0</td>
</tr>
<tr>
<td>2</td>
<td>invokespecial</td>
<td>q_1</td>
</tr>
<tr>
<td>3</td>
<td>ldc</td>
<td>q_2</td>
</tr>
<tr>
<td>4</td>
<td>putfield</td>
<td>q_3</td>
</tr>
<tr>
<td>5</td>
<td>iconst</td>
<td>q_4</td>
</tr>
<tr>
<td>6</td>
<td>new</td>
<td>q_5</td>
</tr>
<tr>
<td>7</td>
<td>dup</td>
<td>q_6</td>
</tr>
<tr>
<td>8</td>
<td>sipush</td>
<td>q_7</td>
</tr>
<tr>
<td>9</td>
<td>invokevirtual</td>
<td>q_8</td>
</tr>
<tr>
<td>10</td>
<td>getfield</td>
<td>q_9</td>
</tr>
<tr>
<td>11</td>
<td>aconst_null</td>
<td>q_10</td>
</tr>
<tr>
<td>12</td>
<td>pop</td>
<td>q_11</td>
</tr>
<tr>
<td>13</td>
<td>bipush</td>
<td>q_12</td>
</tr>
<tr>
<td>14</td>
<td>if_acmpne</td>
<td>q_13</td>
</tr>
<tr>
<td>15</td>
<td>goto</td>
<td>q_14</td>
</tr>
<tr>
<td>16</td>
<td>astore</td>
<td>q_15</td>
</tr>
<tr>
<td>17</td>
<td>invokestatic</td>
<td>q_16</td>
</tr>
<tr>
<td>18</td>
<td>return</td>
<td>final state</td>
</tr>
</tbody>
</table>

Table 2 depicts each possible transition on the FSM for the menu class. From the point of view of ensuring reliable system operation, we are only interested in the above. Table 2 is also used to monitor software execution in phase 2 of the model. Hence, on encountering a particular assembly instruction if the software does not transit to the predicted state, the software should be halted at the previous node as the transition would result in error.

During Phase II, let us assume that software execution begins and takes the following path:
Start Node → q₀
In this case because the software has transited to node q₀, we increase the probability of execution of q₀ by one unit. However, if the next assembly instruction leads to error node instead of node q₂ as per Table 2, then system execution is halted at q₀. Register node q₃ and the assembly instruction in the Faulty_Node table. At this point as per phase III, the automata-based model shall track the shortest possible path from the last previous node (q₀) to the final node using Dijkstra’s algorithm. After this software execution can be resumed from the previous node where it was halted using the shortest possible alternative path tracked previously. If this software execution terminates at the final node then the actual path taken by the software can be written to the Alternative_Path Table.

4 Automata-Based Reliability Model as a System Control Tool

Software executes as a finite state machine. This runtime architecture strongly influences software runtime properties like performance and reliability [5]. Taking formal runtime software architecture model as the basis, one can use the same to estimate actual operational system reliability at any point of time during the system life cycle. Growing software complexity coupled with varied errors due to runtime environment or erroneous input, may result in critical system failures. However, with the increasing dependence of human society on software systems, their failure is no longer tolerable. In times when the science of computing is migrating towards Fault-tolerant, autonomic and self-healing paradigms, models for estimating accurate system reliability have become a necessity. The reliability challenge does not terminate with a model helping determine actual system reliability. The current times require systems that can prevent failures and maintain given values for reliability.

The proposed automata-based reliability model does not confuse software reliability as a quantitative estimate of some parameter that predicts fault-free system operation. Instead this model holds the key to reliable system operation by avoiding errors and directing the system to alternate paths in order to ensure desirable output. The model can be further trained to act as a generic middleware tool that can prevent software system failure and self-adapting the system to user input.

Self-adaptation in real world can be complex and costly to implement for computer systems. However it is a necessity for domains where systems must provide continuous operation or guaranteed dependability. Using architecture-centric approach we can use architecture models at runtime to allow systems to reason about and control operation [9].

The automata-based reliability model can be trained to serve as a generic middleware that can be trained to perform the following functions:

i) Maintain record of current system state, denoted as Sᵢ,
ii) Detect User Input, denoted as Iᵢ,
iii) Compare next state with Next_State Transition Table, if next state is correct or desirable allow the system acquire it.
iv) Else if next state is incorrect, reject user input and let the system remain in its previous state.
v) Maintain probabilistic FSM representation of software under control by maintaining probability of occurrence for each correct and incorrect state.

The proposed automata-based reliability model will serve as a model for reliability estimation of all variably-sized heterogeneous software. Further, this model can be developed as a software control tool to ensure reliable software operation at each runtime instance. This model will super-impose itself on the software under execution and shall ensure error-free software execution.

5 Conclusion

The paper clears the misconception that software reliability modeling can only be applied as a statistical and mathematical concept. By defining, software reliability as the probability of failure-free software operation we have for long dwindled between unworthy figures and estimates. In this human-generated maze of software reliability experts have always tried to model software reliability as what it is actually not.

Software execution is dynamic as it is controlled by a number of environmental factors. As software reliability implies failure-free software operation, hence it is also dynamic. Error at any point during software execution implies that the software has zero reliability. Considering this important fact we conclude that software reliability at any point of software life will be zero or one. This implies that
software may either perform as expected or may fail to produce the expected output. Working on the above points, this paper proposes an automata-based reliability model which can be trained as a software control tool to ensure reliable system operation. This model is sound as it is based on the well-founded automata theory.

The proposed model borrows its basic idea from path testing technique of software testing [3]. However, the similarity is only marginal as path testing involves finding all possible paths in a control flow graph and testing each path individually. We argue that finding all possible paths in automata with loops is impossible and not required. Instead as demonstrated in the discussion above with this model the system shall only require tracing the next alternate node from the previous node incase the resulting node is an error node. The main advantage of the proposed model is the fact that it estimates software reliability through actual runtime software. This major characteristic of the model is also an edge over conventional and complex software reliability models that involve complicated assumptions and statistical calculations.

References: