I.J. Information Technology and Computer Science, **2015, 09, 23-30**
Published Online August 2015 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2015.09.04

Estimating Software Reliability by Monitoring Software Execution through OpCode

Ritika Wason
Bharati Vidyapeeth's Institute of Computer Applications and Management, Delhi, 110063, India
Email: rit_2282@yahoo.co.in

A. K. Soni
Department of Computer Science and Engineering, Sharda University, Greater Noida, 201306, India
Email: ak.soni@sharda.ac.in

M. Qasim Rafiq
Department of Computer Science and Engineering, Aligarh Muslim University, Aligarh, 202002, India
Email: mqrafiq@rediffmail.com

***Abstract*—**Previous studies on estimating software reliability employed statistical functions for next system failure prediction. These models used parameters based on assumptions regarding the nature of software faults and debugging process. However, none of the existing models, attempted on ensuring reliable runtime system operation. To serve the current demand of autonomous, reliable, service-oriented software, we present a novel approach for runtime reliability estimation of executable software. The approach can help control software execution at runtime by monitoring software state-to-state transition at runtime. The approach involves representing executable software as an automata using opcode extracted from executable code. The extracted opcode is then used to learn stochastic finite state machine (SFSM) representation of executable software which is later employed to trace software state-to-state transition at each runtime instance. An evaluation of our approach on Java-based Chart generator application is also discussed to explain how we can ensure reliable software execution and prevent software failures at runtime with the proposed approach.

***Index Terms*—**Automata-Based Software Reliability Model, Opcode, Software Reliability, Stochastic Finite State Machine, Automata-Based Runtime Execution, Stochastic Automata.

I. Introduction

Modern computing paradigms like Ubiquitous, Autonomic, Self-Healing and Fault-Tolerant computing [1-4] are reliable by definition. Current real-time software applications are also expected to perform their services in a reliable manner [5]. Since its inception, varied models have been applied to software reliability estimation [6]. Most of these software reliability estimation models are analytical models [7]. These models consider software to be a black box and estimate the parameters used from available post-failure data [6-7]. Over two-hundred models already exist for estimating reliability of different software systems. However, the realism of the underlying assumptions of these models and the accuracy of their estimates remains questionable [7].

Reliability is a dominant issue in computing and is the main concern behind the line of research presented in this paper. Modern computing and communication infrastructures demand reliable operation as an essential requirement. In conventional reliability models, statistical functions based on certain unknown parameters were employed [8]. The parameter values were determined using post-failure data [7]. Most of these model estimates are unreliable, far from reality and their accuracy debatable [7]. Further none of these models have tried to reduce or eliminate software failure. All conventional software reliability estimation models attempt at only reliability analysis [9]. None of the available models tries to establish techniques that ensure lifetime utility of software.

To handle the problem of software reliability we propose to monitor runtime software execution. Unexpected software behavior and crashes at runtime can be easily avoided or controlled by monitoring runtime software behavior [10-12]. This will help verify whether the execution was correct. Further the model shall help detect errors and avoid them. All software reliability estimation models have assumptions, functions and parameters in common [7]. However, none of the current models take into account the fact that software during execution is an automaton. Hence its reliability should be analogous to the reliability of the automata representing it. We argue that each software execution is 100% reliable if it produces correct output. Similarly, in case the software generates an undesirable output it is 0% reliable. For customers, this is the only meaning of software reliability. Hence, complex mathematical and statistical parameters to estimate software reliability are useless in reality.

In this paper, we investigate whether software behavior at runtime can be controlled on a per instance basis. If incorrect system states can be identified at runtime, any of the current reliability estimation models can be adapted to calculate software reliability using the above estimates. We propose that an automata-based software reliability estimation framework which extracts finite automaton representation of runtime software using bytecode features. The approach does not use any unrealistic assumptions or statistical parameters for reliability estimation [7]. Instead it utilizes operation codes or opcodes from bytecode at runtime to trace the execution sequence of instructions and next software state at runtime. From this information about software state-to-state transition, the model can avoid fault execution as well as ensure correct software execution at runtime.

The automata-based software reliability model utilizes finite state machine (FSM) representation of executable software to estimate system reliability at any point during the software life cycle. All software execute as a finite state machine (FSM). Hence, a Finite State representation of executable software is the most appropriate representation of the various system states and the transitions resulting in state change.

The broad goal of our research is to make available an automata-based reliability estimation framework 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.

This paper is structured as follows: Section 2 examines how opcodes form the basis of executable code. Section 3 throws light on any related work employing our proposed approach. Section 4 proposes the automata-based reliability approach. Section 5 elaborates application of the above approach to ensure uninhibited software operation despite the occurrence of faults. We close this paper by evaluating the pros and cons of this research in Section 6.

II. Executable Software Representation Using Opcode

The general format of each machine language instruction is constituted using opcode and operands [13]. The operands can be a memory address, a register or a value [13]. Opcode is short for operational code. At its simplest it is a subset of machine language instructions that denote the operation to be performed [13]. Opcodes 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 [13].

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 an automata-based software reliability model in section IV.

III. Related Work

Automata-Based Software Execution is a very naive domain. Recently the importance of formal languages has been used for controlling software implementations. Ref. [14] proposed use of formal languages of finite words between control designs and software implementations. Their work proves that finite automata provides analyzable representations of software implementations and can help capture interesting specifications of switched linear systems [14]. Ref. [15] applied automaton as a basis for recognizing sequences semantically equivalent to a base program sequence during concurrent program execution. As part of their automata-based testing model implemented on Java-based multi-threaded program authors in [15] have designed and implemented key components including automata-generator, program transformer and replay controller. All the components have been designed in accordance to adopt an automaton. Ref. [16] employed automaton as a basis to formalize confidentiality of secret information manipulated by a program. In comparison to static checking, automaton-based monitoring of information flow in a program offers dynamic control of program execution including forbidding dangerous actions. Ref. [17] authors have developed visual software building, verification and validation tool to help in automat-based software development. The tool helps debug software under study in terms of automata. The tool is being continuously upgraded and the current version available is UniMod 4.3. However, the tool functionality is limited in expanse that it can only help in software debugging and verification. Automata have also made its way through in automated robotic systems. Ref. [18] worked upon a novel robotic motion planning model using hierarchical model checking. In this case, notably the robot and its functional environment are all modeled as a timed automaton. System requirements are also formalized using Computational Tree Logic (CTL) formulas. Ref. [19] suggested learning automata based trust model for portioning user-based agents to fair and unfair groups of services available in a service-oriented environment.

The above discussion establishes how slowly automata-based models are penetrating into different software application domains where reliable program execution is of prime concern. Taking note of the above, we have proposed an automata-based software reliability model that can help sustain failure-free software execution at runtime.

IV. Automata-Based Software Reliability Model

Software executes as a system of finite states. Every 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 [20]. It has formed the basis for both computer programs and sequential logic circuits for long. Actually it is an abstract machine that is in any one of a finite number of states at any given time. This abstract machine can be formally defined as a quintuple ($\sum$, Q, q_0, δ, F) where [20]:

- $\sum$ is the finite set of input symbols (a finite, non empty set of symbols)
- Q is a finite, nonempty set of states.
- q_0 is the initial state, an element of Q.
- δ is the state-transition function, δ: Q X $\sum$ → Q
- F is the set of final states, a subset of Q.

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_f belongs to F ($q_f \in$ F), then the software is 100% reliable. However, if at any point during its execution the next state q' does not belong to F ($qf \notin$ 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 [20]. However, it is different as here the transition function δ is defined as [20]:

$$\delta: Q \times \Sigma \rightarrow P(Q) \quad (1)$$

Here, P (Q) denotes power set of Q. The above transition function can be expressed as a membership function [20]

$$\delta: Q \times \Sigma \times Q \rightarrow \{0,1\} \quad (2)$$

such that

$$\delta\ (q, a, q') = 1\ if\ q' \in \delta(q, a) \quad (3)$$

and

$$\delta\ (q, a, q') = 0\ if\ q' \notin \delta(q, a) \quad (4)$$

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 ½ and incorrect state with probability ½ too [20]. 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 develop an approach for software reliability estimation based on probabilistic automata. This 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 failure data (brute force) [21].

Software Architecture is a key means for achieving understandability of the complex, real-time software systems [22]. 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.

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

The notations used in the model are described in Table 1 below:

Table 1. Notations for the Automata-Based Software Reliability Model

G(V,E)	Graph Representation of executable software as a set of V nodes and E links.
R(Q)	Reliability of a software as a set of Q components or nodes
F(I,N)	Function that calculates next software state using previous state information (N) and assembly opcode (I)
x_i	Input node i
V_{ij}	Weight b/w node x_i and x_j
P(i)	A distinct collection of nodes through the FSM from the start node to the final node.

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 [23].The model uses the above model and data to maximize runtime software reliability.

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

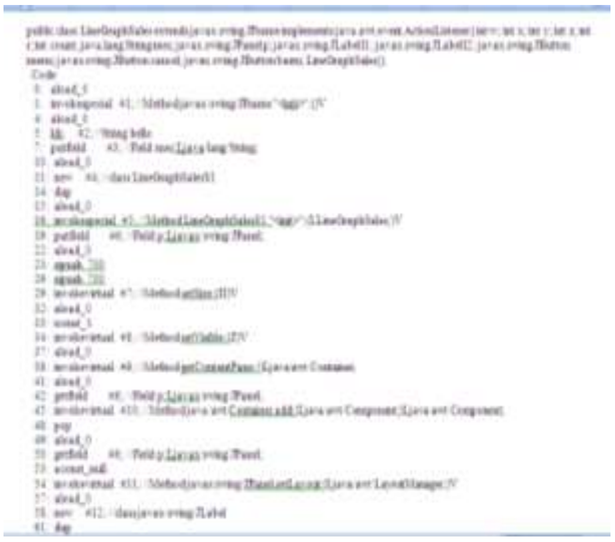
Table 2. Phases of the Automata-Based Software Reliability Model

Phase I:	***FSM Representation***
Start	
Step 0:	*Preprocess all executable code of software under scrutiny by transforming their values to equivalent assembly code.*
Step 1:	*Extract opcode from each assembly instruction. Repeat steps 2-4 for each assembly opcode until end of executable code file*
Step2:	*For each unique assembly opcode instruction, record the opcode instruction and its corresponding node to the next_state transition table.*
Step 3:	*Represent each unique assembly opcode as a new, unique FSM node, linked to its parent node through the opcode transition.*
Step 4:	*Check for end of file; assign it as the final node of the FSM.*
Step 5:	*Introduce an error node in the FSM, linked to all existing nodes.*
Step 6:	*Assign each node of FSM an equal probability of execution V_{ij}. where $V^{ij} \rightarrow \{0,1\}$.according to eqn. (2)*
Phase II:	***Software Implementation***
(Feed Forward)	*Initialize Software Execution*
Step 7:	*Receive input node q_i and assembly opcode, a_i.*
Step 8:	*Validate next node from next_state transition table using eqn. (3) and (4). If (3) is true allow transition to next node.*
Step 9:	*Increase the probability of execution of last traversed node by a unit.*
Step 10:	*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 11:	*Repeat steps 12-13 till last executable instruction.*
Step 12:	*If step 10 executes, check for alternate next node using Djikstra's algorithm [11].* *Else, let the software execute.*
Step 13:	*Continue software execution using the next alternate node.*
Phase IV:	***Software Maintenance***
Step 14:	*Record the complete path of the successful execution to alternate_path table.*

We demonstrate the step by step implementation of the above algorithm on a class of real world Java-Based ChartGenerator application developed by PostGraduate students.

Phase I

Step 0: The equivalent assembly code for the Java class LineGraphSales.class was obtained by disassembling the executable .class file. Figure 1 below depicts a portion of the same.



```
public class LineGraphSales extends javax.swing.JFrame implements java.awt.event.ActionListener{ int w; int x; int y; int z; int [illegible]
[illegible] java.lang.String[illegible] javax.swing.JPanel p; javax.swing.JLabel l1; javax.swing.JLabel l2; javax.swing.JButton
[illegible] javax.swing.JButton cancel; javax.swing.JButton [illegible] LineGraphSales();
 Code:
  0:  aload_0
  1:  invokespecial  #1; //Method javax/swing/JFrame."<init>":()V
  4:  aload_0
  5:  ldc   #2; //String hello
  7:  putfield     #3; //Field [illegible]:Ljava/lang/String;
  10: aload_0
  11: new    #4; //class LineGraphSales$1
  14: dup
  15: aload_0
  16: invokespecial #5; //Method LineGraphSales$1."<init>":(LLineGraphSales;)V
  19: putfield     #6; //Field p:Ljavax/swing/JPanel;
  22: aload_0
  23: sipush 700
  26: sipush 700
  29: invokevirtual #7; //Method setSize:(II)V
  32: aload_0
  33: iconst_1
  34: invokevirtual #8; //Method setVisible:(Z)V
  37: aload_0
  38: invokevirtual #9; //Method getContentPane:()Ljava/awt/Container;
  41: aload_0
  42: getfield     #6; //Field p:Ljavax/swing/JPanel;
  45: invokevirtual #10; //Method java/awt/Container.add:(Ljava/awt/Component;)Ljava/awt/Component;
  48: pop
  49: aload_0
  50: getfield     #6; //Field p:Ljavax/swing/JPanel;
  53: aconst_null
  54: invokevirtual #11; //Method javax/swing/JPanel.setLayout:(Ljava/awt/LayoutManager;)V
  57: aload_0
  58: new    #12; //class javax/swing/JLabel
  61: dup
```

Fig. 1. Equivalent assembly code of LineGraphSales.class

Step1 & 2: Opcode was extracted by parsing each assembly instruction from code obtained in Step 0 and for each unique assembly instruction a new record was added to the Next_State_Transition table. The table is referred as table 3 below:

Table 3. Next_State Transition Table for LineGraphSales

S.No	OpCode Instruction	Next State
1	Aload	q_0
2	invokespecial	q_1
3	Ldc	q_2
4	putfield	q_3
5	New	q_4
6	Dup	q_5
7	sipush	q_6
8	invokevirtual	q_7
9	iconst	q_8
10	getfield	q_9
11	Pop	q_{10}
12	aconst_null	q_{11}
13	bipush	q_{12}
14	astore	q_{13}
15	ifeq	q_{14}
16	goto	q_{15}
17	invokestatic	q_{16}

Step 3, 4 &5: equivalent FSM was generated using the same. The equivalent FSM representation for the executable file LineGraphSales.class is depicted in Figure 2 below:

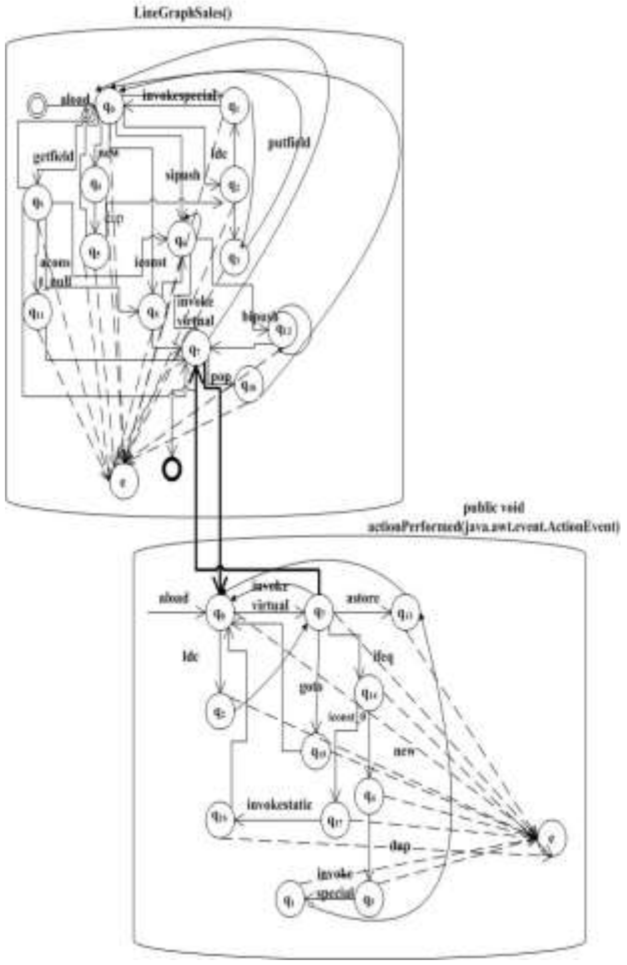


Fig. 2. FSM representation of LineGraphSales.class

Step 6: Initially as each node of the above FSM has equal probability of failure, we assign each node an equal weight V_{ij} as per equation (2). Figure 3 next depicts the Stochastic Finite State Machine obtained at the end of this step.

At the end of phase I, stochastic finite state representation of software is obtained. The automata-based reliability model can now serve as a control tool that can control, monitor and trace each software execution. At this point it is important to note that during each execution, software may not traverse all connected nodes in its automata representation. Instead for each execution, software traverses a set of nodes starting from the initial node till it terminates. Hence, for each execution software traverses a path P

$$where\ P\ (q_i, q_0, \dots, q_f) \qquad (5)$$

is a set of nodes such that $q_i \in Q$, the selection of Q is guided through user input.

We now discuss the case here for correct software execution.

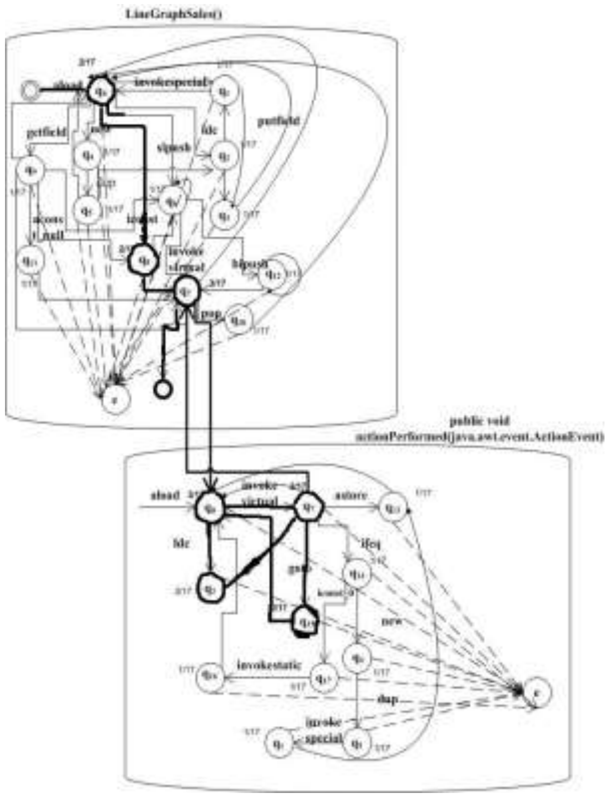


Fig. 3. SFSM Representation for LineGraphSales.class

Phase II

We demonstrate Phase II of the application by taking an execution instance of the LineGraphSales class. Let's say the LineGraphSales class follows the following path (depicted in Figure 3 using bold, solid lines).

$$P(i) = \left(q_i, q_0, q_8, q_7, q_0, q_2, q_7, q_{15}, q_0, q_7, q_f\right) \qquad (6)$$

Step7-9: As the above path starts from the initial node, we keep increasing the probability of each node traversed by the software by a unit. As a result the traversed nodes have a higher probability than the non-traversed nodes.

Phase III and IV

Step 11-14: Software terminates successfully and the path traversed is recorded as a log entry to the alternate_path table.

V. Model Evaluation

Accurate monitoring of reliable software operation has become increasingly important due to the expensive impact of software failures and recovering from them. The proposed automata-based software reliability model is an attempt to achieve the same. The model unlike its conventional counterparts does not stop at simply estimating the reliability value of software. Instead the model is designed to enable early fault detection and avoidance. The model is a novel breakthrough towards solving the software reliability challenge.

All conventional software reliability models depend on either the number of variables related to the software development process, the failure process or the fault object. The proposed automata-based software reliability model does not utilize any such mathematical variables or functions. Instead the model utilizes software state and input information to control software behavior at runtime. As a result the proposed model is simpler and direct in its approach in comparison to its confusing counterparts.

None of the conventional software reliability models can be generically applied to all software systems of any size. Engineers need to make a choice for the appropriate model that suits the data available with the software [24]. The proposed automata-based software reliability model can be generically applied to all software systems of any size, complexity or based on any technology. For application of this model no consideration of the available data from the software is important.

We now compare the model with its conventional counterparts using some comparison criteria identified in [8, 24]. The criteria are the most common criteria used to make a choice between different conventional reliability models.

Table 4. Automata-based Reliability Model Evaluation

S. No	Criteria	Automata-Based Reliability Model
1	***Predictive Validity***: Capability to predict future failure behavior in operational phases from present and past failure behavior	Does not predict failure behavior, instead detects fault and halts the system before executing the same.
2	***Capability:*** Ability of the model to estimate with accuracy the quantities needed in planning and managing software projects.	Irrespective of software size, complexity and structure, the model is capable to monitor software execution and control failure due to input or software related errors.
3	***Quality of Assumptions:*** Degree to which an assumption is supported by the actual data.	Conventional reliability models make assumptions regarding the failure process. Validity of most of the assumptions in real-world is questionable. Automata-Based Model does not rely on any such dubious assumptions.
4	***Applicability:*** Is the degree of model applicability across different software products.	The proposed model unlike its conventional counterparts is applicable across different software products.
5	***Simplicity:*** A model should be simple in collecting data, in concept and readily implementable.	The proposed model is simple as it does not require any post-failure data. The model is based on the formal theory of automata but can be easily understood by software engineers with no mathematical background. Finally the model can be directly implementable on the software at runtime.

The comparison criteria discussed above compares the proposed automata-based software reliability models with its conventional counterparts on a common basis. The framework of comparison criteria and the information gathered in support of the proposed model clearly shows that the proposed model is a better choice as compared to its conventional counterparts.

VI. Conclusion

An approach to software reliability based on theory of automata has been derived. Various methods based on the theory of probability and statistics have been used to assess reliability of elements in hardware structures [1]. This approach establishes a model that is simple, mathematically verifiable and directly implementable on software at runtime. The model permits the monitor and control of software execution at runtime. The state transition information collected on basis of opcode constituting program code can then be linked to actual state transition of the software at runtime. In case the software acquires an allowable transition, then it is performing reliably. However any non-allowable transition indicates that the software shall execute a fault. The model can work as a control tool on top of any software to avoid fault execution. Further if we integrate this model directly at the intermediate code generation phase of a compiler, engineers can decide about the feasibility of a software project.

Use of finite state representation technology suffers from the state explosion problem [25]. The size of a finite state model may increase exponentially as the number of components grows. However, automata-based reliability model controls this problem to a large extent in the following way:

- The model generates a unique state for a unique opcode instruction. The instruction set for every programming language is finite. Hence the next_state transition table for each runtime software code shall be represented through finite state model irrespective of code size.

Our conclusion from this work is rather positive. All conventional reliability models give unacceptably optimistic reliability predictions. The proposed model does not attempt at any kind of data-driven predictions. Instead, the model controls reliability at runtime through the use of a next_state transition knowledgebase. The next_state transition knowledge base uses opcode instructions to calculate the next software state. Preliminary testing of the model has been done. However, its establishment as a generic model requires its conversion to a software tool. We are working on the same at the time of this writing. Though the proposed model is successful in controlling and avoiding all software faults. However, the approach shall be unable to control hardware faults that may result in software failure.

ACKNOWLEDGMENT

The authors wish to thank Late Prof. Pervez Ahmed. This work was conceived under his able guidance and vision. It would have been completely impossible to complete this framework had Prof. Ahmed not provided his able and timely inputs into the work. May God rest his soul in peace.

REFERENCES

[1] M. Krejsa, P. Janas and V. Kresja, " Direct Optimized Probabilistic Calculations," *in Proc. of 3rd WSEAS International Conference on Mathematical Models for Engineering Sciences (MMES'12)*, Paris, pp. 216–221, 2012.

[2] Danial Rahdari, Amir Masoud Rahmani and Afsane Arabshahi, "Fault Tolerant Message Efficient Coordinator Election Algorithm in High Traffic Bidirectional Ring Network," (*IJITCS) International Journal of Information Technology and Computer Science,* vol. 5, no.1, pp. 1-14, December 2012.doi:10.5815/ijitcs.2013.01.02.

[3] Karima Mahdi, Raida Elmansouri and Allaoua Chaoui, "On Transforming Business Patterns to Labeled Petri Nets Using Graph Grammars," (*IJITCS) International Journal of Information Technology and Computer Science,* vol. 5, no.2,pp.15-27, January 2013.doi:10.5815/ijitcs.2013.02.02.

[4] P. Lokesh Kumar Reddy, B. Rama Bhupal Reddy and S. Rama Krishna, "Self-Organized Detection of Relationships in a Network," (*IJITCS) International Journal of Information Technology and Computer Science,* vol. 5, no.2,pp.80-87, January 2013.doi:10.5815/ijitcs.2013.02.02.

[5] M. Viswanathan, "Foundations for the Run-time Analysis of Software Systems," Ph.D. Dissertation, Univ. of Pennysylvania, Philadelphia, PA, USA, AAI9989666, 2000.

[6] Md. Anjum, Md. Asraful Haque and Nesar Ahmed, "Analysis and Ranking of Software Reliability Models Based on Weighted Criteria Value," (*IJITCS) International Journal of Information Technology and Computer Science,* vol.5,no.2,pp. 1-14, January 2013. doi: 10.5815/ ijitcs. 2013. 02.01.

[7] A.L. Goel, "Software Reliability Models: Assumptions, Limitations and Applicability," *IEEE Trans. Software Engineering,* vol. SE-11, no. 12, pp. 1411–1423, 1985.

[8] J.D. Musa, "A Theory of Software Reliability and its Applications," *IEEE Transactions on Software Engineering,* vol.SE-1, no.3, pp. 312–327, 1975.

[9] Pan Jiantao, "Software Reliability", http://users.ece.cmu.edu/~koopman/des_s99/sw_reliability/ (Accessed On: 12-01-2013).

[10] V. Skorpil, L.Cizek, and J. Stastny, "Path Optimization by Graph Algorithms," in Proc. 16th WSEAS International on Computers (CSCC'12), Greece, pp. 73–77, 2012.

[11] "A Machine Learning Based Efficient Software Reusability Prediction Model for Java Based Object Oriented Software"

[12] M. Santhosh Prabhu, A. Hazra, P. Dasgupta, "Reliability Guarantees in Automata-Based Scheduling for Embedded Control Software," *Embedded Systems Letters", IEEE* , vol.5,no.2,pp.17-20,June2013.doi: 10.1109/LES.2013.2250479.

[13] I. Santos, F. Brezo, X. U. Pedrero and P.G. Bringas, "Opcode Sequences as Representation of Executales for data-mining based unknown malware detection," *Information Sciences: an International Journal,* vol 231, pp. 64-82, 2013.

[14] Gera Weiss and Rajeev Alur, "Automata based interfaces for control and scheduling," in *Proc. of the 10th international conference on Hybrid systems: computation and control* (HSCC'07), Alberto Bemporad, Antonio Bicchi, and Giorgio Buttazzo (Eds.). Springer-Verlag, Berlin, Heidelberg, 601-613, 2007.

[15] Heui-Seok Seo, In Sang Chung, Byeong Man Kim and Yong Rae Kwou, "The design and implementation of automata-based testing environment for Java multi-thread programs," *Software Engineering Conference, APSEC 2001. Eighth Asia-Pacific*, pp.221-228, 4-7 Dec. 2001, doi: 10.1109/APSEC.2001.991480.

[16] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. "Automata-based Confidentiality Monitoring," *in Proc. Asian Computing Science Conference (ASIAN'06), Revised Selected Papers*, vol. 4435 *LNCS*, pp. 75–89. Springer-Verlag, January 2008.

[17] D. Kochelaev, B. Khasanzyanov, B. Yaminov and A. Shalyto, "Instrumental Tool for Automata Based Software Development UniMod 2," vol. 1, pp. 55-58. DOI: 10.15514/SYRCOSE-2008-2-11.

[18] Rui Wang et al. , "Timed automata based motion planning for a self-assembly robot system," *Robotics and Automation (ICRA), 2014 IEEE International Conference on* , vol., no., pp.5624,5629, May 31 2014-June 7 2014 doi: 10.1109/ICRA.2014.6907686.

[19] A. Khoshkbarchi, H.R. Shahriari and M. Amjadi, "Comparison-based Agent Partitioning with Learning Automata: A Trust Model for Service-Oriented Environments," *Information Security and Cryptology (ISCISC), 2014 11th International ISC Conference on* , vol., no., pp.109-114, 3-4 Sept. 2014 doi: 10.1109/ISCISC.2014.6994032.

[20] M. O Rabin, "Probabilistic Automata," *Information and Control,* vol.6, pp. 230-245, 1963.

[21] Ritika Wason, P. Ahmed and M. Qasim Rafiq, "A Tool for Runtime Reliability Estimation and Control Using Automata-Based Software Reliability Model," *in Proc. of the 2nd WSEAS International Conference on Computers, Digital Communications and Computing (ICDCC 2013)*, Romania, pp.51-56, 2013.

[22] A. Dimov, "Formalizing Nonfunctional Characteristics in Software Architecture- A Way to Achieve System Dynamism," *in Proc. of 2nd WSEAS International Conference on Computers, Digital Communications and Computing (ICDCC 2013)*, Romania, pp. 14, 2013.

[23] M. Iliescu, V.Ursianu, F. Moldoveanu, R. Ursianu and E. Ursianu, "Mathematical Models to Estimate the Quality of Monitoring Software Systems for Electrical Substantiations," *in Proc. of 2nd WSEAS International Conference on Applied and Computational Mathematics (ICACM'13)*, Greece, pp 112-117, 2013 .

[24] A. Iannino, J. D. Musa, K. Okumoto and B. Littlewood, "Criteria for Software Reliability Model Comparisons," *IEEE Transactions on Software Engineering,* vol. SE-10, no. 6, pp. 687-691, 1984.

[25] J. C. Corbett et al., "Bandera: extracting finite –state models from Java source code," in Proc. of the 22nd *International Conference on Software Engineering,* New York, pp.439-448, 2000.

Authors' Profiles

Ritika Wason did her B.Sc from Delhi University, MCA from Guru Gobind Singh Indraprastha University and is a doctoral student at the School of Engineering and Technology, Sharda University, India. She joined Bharati Vidyapeeth's Institute of Computer Applications and Management as an Assistant Professor in 2013. She has published papers in several National and International conferences proceedings and journals and has also authored books on Software Testing. She is also a life time member of Computer Society of India. Her research interests include formal models and approaches for software quality especially software reliability and software testing.

Dr. A.K Soni has done his Ph.D. & M.S.(Computer Science) both from Bowling Green State University in Ohio, USA. He is the Professor and Head, Department of Information Technology, Sharda University.

He has more than seventeen years of teaching experience. He has published many papers in national and international journals. His research area includes Software engineering, Datamining, Database management systems and object oriented systems.

Prof. Dr. M. Qasim Rafiq did B.Sc Engg (Elect), M.Sc Engg (Inst), and M.Tech (CSE) from A.M.U in 1972, 1979 and 1986 respectively, Did Ph.d in parallel processing in 1996 from UOR Roorkee. Joined the teaching at AMU in 1979 as Lecturer, became Reader in 1987, and Professor in 1997. A founder chairman of the Deptt, served thrice as chairman Dr Rafiq is an expert member of NBA committee, selection committee to various universities in India, Academic council of different universities and resource person to different places for delivering expert lectures. He has also been external examiner for Ph.D and undergraduate courses to various universities. Have guided many research scholars for Ph.D. Published a no of papers in reputed journals/conferences. Fellow IETE, Fellow IE & Senior member CSI also.

How to cite this paper: Ritika Wason, A. K. Soni, M. Qasim Rafiq,"Estimating Software Reliability by Monitoring Software Execution through OpCode", International Journal of Information Technology and Computer Science(IJITCS), vol.7, no.9, pp.23-30, 2015. DOI: 10.5815/ijitcs.2015.09.04