Automata-Based Software Reliability Paradigm for Ubiquitous Systems

Ritika Wason [1], P. Ahmed [2], M. Qasim Rafiq [3]
[1&2] School of Engineering and Technology, Sharda University
Greater Noida, India
[1] rit_2282@yahoo.co.in
[2] pervez.ahmed@rediffmail.com
[3] Department of Computer Science, Aligarh Muslim Uiversity
Aligarh, India
[3] mqrafiq@hotmail.com

Abstract—The paper proposes a novel paradigm for accurate reliability estimation of ubiquitous systems. The proposed prototype differs from all conventional software reliability models as it does not rely on any kind of post failure data, questionable assumptions or statistical distributions. The paradigm instead is based on software state to state transition at runtime. Using this state to state transition information, the model ensures fault free software operation.

Keyword-Software Reliability, Finite State Automata, Automata-Based Software Reliability Model, Self-Learning Automata.

I. INTRODUCTION

Accurate failure-free operation under different environmental conditions is a necessity for profitable ubiquitous software. The vision of ubiquitous software is to support humans in their daily tasks while remaining practically invisible [1].Ubiquitous systems aim at configuring themselves without any need for human intervention. Before realizing the paradigm of ubiquitous computing completely many different challenges like scalability, interoperability, reliability etc need to be addressed. Reliability of ubiquitous systems is an important, difficult to ignore attribute for all software especially ubiquitous systems.

A ubiquitous system is meant for heterogeneous execution with different new nodes of failure [6]. In such a scenario ensuring failure free operation of ubiquitous software is a challenge in itself. Conventional approaches to reliability estimation will fail badly for ubiquitous software. Reliable ubiquitous software is expected to be fault-free, high-performance and self-healing [1]. Hence faulty operation for ubiquitous software is not tolerable. Science of software reliability till date does not have a reliability estimation model to estimate software reliability with 100% accuracy. Neither do any of the conventional models attempt to handle or control software failure [10]. The major reason behind these limitations of conventional software reliability models can be attributed to a number of factors, like:

- Similar treatment of software and hardware reliability despite fundamental differences in both entities.
- Use of misrepresented assumptions to quantify software reliability.
- Analysis of post-failure data for model parameter estimation.
- Dependency of software reliability models on statistical distributions that do not address the dynamic nature of software.

The above limitations of conventional software reliability models have been pointed in the past by many researchers [2,7,10]. However, reliability engineers continue to design reliability estimation models using faulty assumptions along with statistical distributions in an attempt to accurately estimate software reliability.

It should be noted that dependence of human society on software has grown manifold. To meet real-world expectations software systems have also grown exponentially both in terms of size and complexity. Software today is no longer a structured program which will be developed, tested, repaired, implemented and debugged throughout its useful life. The present age expects intelligent software to be developed in a manner that software can continue to work despite occurrence of errors (fault-tolerant) or software may take corrective measures in case of failure (self-healing). This complex state of affairs shows that though we have advanced software systems we do not have appropriate models to ensure confidence of fault free operation. Hence, to completely realize our fault-free, autonomic, self-healing or ubiquitous systems we need to simultaneously develop a reliability paradigm to ensure smooth operation of ubiquitous systems.

Reliable software is expected to perform continuously despite the presence of faults (fault tolerance) [4]. Till date there does not exist a software reliability estimation model that can overcome software failure [2]. There exists no formal model of software representation that can be tested and verified to ensure reliable software execution. Existing software reliability estimation models quantitatively estimate number of times software would perform correctly. However, for each execution, software is an automaton for converting distinct set of inputs into expected set of outputs. These automata, transits from state to state using some transition instruction. In its execution life, if a software executes ten million times, then every time the software executes along a correct path it will terminate correctly. If we track this execution path of the software then we can direct the software in any mode (self-healing or fault-tolerant). The above discussion demonstrates the actual nature of software reliability. The problem with the conventional software reliability estimation models is that they all treat software reliability as an extension of hardware reliability. None of the existing software reliability estimation models treat software reliability as discussed above.

Software especially ubiquitous, shall have zero reliability each time it executes a fault, similarly software is 100% reliable if it performs as expected. Contrastingly, conventional software reliability modelling approaches do not address software reliability as above. Instead they estimate software reliability using some statistical model. Parameters required by the model are estimated using existing data on software failures [10]. Practicality of such models and their underlying assumptions is thus questionable [10]. We argue that a ubiquitous system is a combination of many possibly globally distributed components. The components may fail individually but the system needs to perform reliably. To ensure this conventional software reliability models using complex mathematics and statistics are of no help. The paper proposes automata-based software reliability model that utilizes finite state machine (FSM) representation of executable software to estimate software reliability. Software at runtime is an automaton. An FSM representation of executable software depicts different system states along with the transitions ensuing state change. Hence, reliability of software is equivalent to the reliability of the automata being executed. The broad goal of this research is to develop an automata-based reliability estimation model that can be applied to control the operation of different, life-critical software like autonomic, self-healing, ubiquitous etc.

The paper is organized as follows: Section II discusses the importance of reliability for ubiquitous software. Section III describes the automata-based software reliability model. Section IV elaborates the advantages of the proposed model over conventional software reliability estimation models. Section V evaluates the proposed model.

II. RELIABILITY REQUIREMENT FOR UBIQUITOUS SYSTEMS

Mark Weiser, XEROX coined the phrase ubiquitous computing in 1988 [1, 8]. The paradigm was developed with a concern to make computers so pervasive that they invisibly assist our daily tasks. To achieve this highest ideal of ubiquitous systems is a challenge in itself. The very idea of software available globally, pervasively involves numerous issues like availability, interoperability and reliability. Reliability of ubiquitous services and devices is a crucial requirement. To construct reliable ubiquitous systems we need to have a reliability paradigm that can ensure characteristics like self-monitoring, self-regulating and self-healing into the available ubiquitous systems. Programming has always relied on models for ensuring reliable software operation despite the occurrence of errors. However, reliable software execution still remains a problem. However, solution of this problem has been crucial in the present times when computing has grown to the level of ubiquitous or pervasive computing. To solve this issue we first define software reliability as a function of correct I/O (input/output) pair at runtime. This can be expressed as follows:

Input: I/O Pair < i, o>

Output: If o= f (i), TRUE; else, FALSE.

Reliability: For all <i, o>: on input <i, o>, the software must return the correct output with probability p, where $0 \geq p \leq 1$.

Software reliability at any point of time is a function of the I/O pair at runtime [5]. Hence, runtime software is the most appropriate source for software reliability estimation. Contrastingly, none of the conventional software reliability models give any consideration to runtime software structure. Hence, to meet the reliability challenge of ubiquitous systems, we require an efficient software reliability estimation model based on the runtime structure of ubiquitous software to control system execution to ensure reliable operation.

III. AUTOMATA-BASED SOFTWARE RELIABILITY MODEL

Conventional reliability estimation approaches estimate software reliability using some assumptions alongwith post-failure data fitted over some statistical parameters. These approaches give no due consideration to the actual software structure at runtime. Ubiquitous systems are meant to be so embedded and so natural that we use it without even thinking about it [8]. To realize this, ubiquitous software should ensure failure free operation under all operating environments. Conventional software reliability estimation models cannot guarantee this [10]. What we require is a rigorous understanding of how ubiquitous software shall execute at runtime. We need a reliability model capable of analyzing, correcting, diagnosing, documenting, enhancing, evaluating and implementing software execution. Complete realization of this control tool will require an amalgamation of many different features of computation like automata theory, Hoare logic, Formal Language theory etc [1]. We now propose an automata-based software reliability estimation model that can control reliable software operation by detecting software state-to-state transition at runtime.

We hypothesize that software during execution is an automata. Hence, a finite state machine-based software representation model should be used to control software execution. This automata-based software representation symbolizes states software can acquire during its function. Hence, we can trace the path along which software executes. Once we trace the software path, we can control the system and stop its progress before it executes failure. The proposed automata-based software reliability is explained in Figure 1 below:

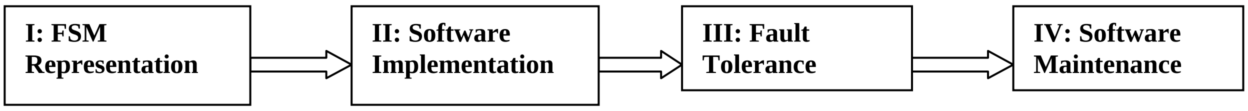

Figure 1: Phases of Automata-Based Software Reliability Model

Figure 1 above depicts the various phases of the proposed automata-based software reliability model. The functions of the model during each phase are elaborated upon in Table 1 below.

TABLE I CHARACTERISTICS OF EACH PHASE OF AUTOMATA-BASED SOFTWARE RELIABILITY MODEL

Phase	Name	Description
I	FSM Representation	**Input**: Assembly code of executable software under analysis; **Output**: i) Finite state machine representation. ii) Next_State Transition Table iii) Stochastic FSM of finite state machine representation as obtained in (i).
II	Software Implementation	Use Next_State Transition Table to monitor software execution. Increase the probability of execution of each node software traverses by a unit. If next transition results in error node halt system execution, mark the next node with a probability zero and record it to Faulty_Node table.
III	Fault Tolerance	From the node where the system has halted, find the next possible node using Djikstra’s Algorithm. Continue software execution and repeat this step till the software does not terminate.
IV	Software Maintenance	Record the path taken by the software to the Alternate_Path Table.

The above automata-based reliability model, when implemented will help ensure reliable software operation. At the time of this writing, we are working with the software implementation of this model. We demonstrate the working of the above model using a Java class named Login.java. This class is a component of Java-based Chart Generator Application developed by post-graduate students. Compiling the above code produces the executable file Login.class. .class file contains executable code in its bytecode notation. On disassembling this bytecode to its assembly instruction set an FSM for Login.class as shown in Figure 2 was obtained:

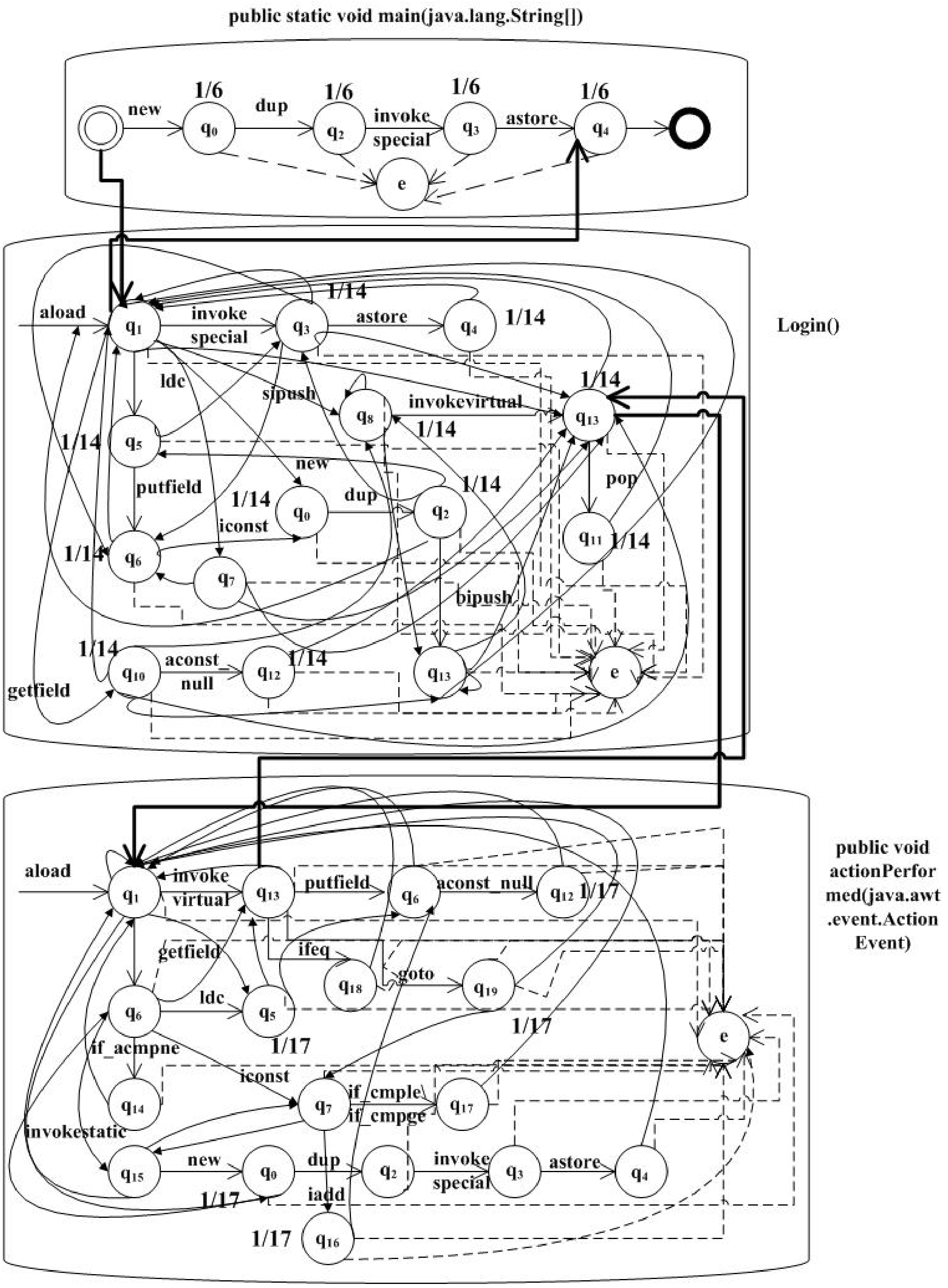

Figure 2: Equivalent FSM for Login.class

Figure 2 depicts how different assembly instructions result in software transition from one FSM state to another. Each of these transition instructions alongwith the possible next state will be recorded in Next_State Transition Table. The Next_State Transition Table obtained for Login class is shown as Table 2 below:

TABLE 2 NEXT_STATE TRANSITION TABLE OF LOGIN

S.No	Transition Instruction	Next State
1	new	q_0
2	aload	q_1
3	dup	q_2
4	invokespecial	q_3
5	astore	q_4
6	ldc	q_5
7	putfield	q_6
8	iconst	q_7
9	sipush	q_8
10	bipush	q_9
11	getfield	q_{10}
12	pop	q_{11}
13	aconst_null	q_{12}
14	invokevirtual	q_{13}
15	if_acmpne	q_{14}
16	invokestatic	q_{15}
17	iadd	q_{16}
18	if_cmple/if_cmpge	q_{17}
19	ifeq	q_{18}
20	goto	q_{19}
21	return	final state

Table 2 can be used to supervise software execution during phase 2. If after encountering an assembly instruction the software does not transit to the next recorded state, the software can be halted at the previous state i. For example, if the program Login starts execution along the following path:

Start Node$\rightarrow$ q_1

When the software arrives at q_1, we increase the probability of execution of q_1 by one unit. However, if 'ldc' leads to error node instead of q_5 as per Table 2, then system execution is halted at $q_{1.}$ q_1 and 'ldc' will be recorded in Faulty_Node table. In phase III, the software can track the shortest possible path from the node (q_1) to the final node using Djikstra's algorithm. After this software execution can be resumed from q_1 using the shortest possible alternative path. Once software terminates then the actual path treaded by the software can be recorded in the Alternative_Path Table.

IV. AUTOMATA-BASED SOFTWARE RELIABILITY MODELS VERSUS CONVENTIONAL SOFTWARE RELIABILITY ESTIMATION MODELS

The proposed automata-based software reliability model scores over the limitations of its conventional counterparts as follows:

- The model does not use any assumptions about software execution; instead it uses actual software execution data to ensure fault-free software operation.
- Theory of software reliability has been retrofitted to software [2]. Many different hardware reliability models are also being applied to software (ex: Power Model, Crow, 1974). The proposed automata-based software reliability model is unique as it is based on the true nature of software execution.

- The model does not try to predict or assume the nature of processes applied to debug software (imperfect/perfect debugging) [7]. Instead the model spends effort on analyzing, diagnosing, correcting and documenting software path of execution at runtime.
- Conventional models attempt to estimate reliability through testing [2]. However, the accuracy of the estimates is governed by the thoroughness of test coverage. Automata-Based Software reliability model is not dependent on the testing process which may vary significantly from the actual system usage and may not be done exhaustively.
- Conventional software reliability estimation models are statistical models that accept some kind of failure data as input and produce system reliability estimates as output [10]. These models estimate software reliability completely ignoring software architecture during actual software execution [9]. The automata-based software reliability model is based on the fact that software during execution is an automata. The model uses actual assembly instruction set to trace software execution.

Unlike its conventional counterparts the automata-based reliability model is an extension to the concept of effective path selection [3] from a control flow graph representing software. However, the model does not attempt to estimate all possible paths in automata. This is impossible for software systems as software-based automata representations may have loops.

V. CONCLUSION

Ubiquitous computing systems are the next generation of computing. However, they also articulate the current technical challenges for software researchers. Despite being coined over two decades ago, the vision of ubiquitous computing seems as futuristic today as it was earlier. The major reason for this is the fact that ensuring failure-free continuous software operation under all environments and platforms is a big challenge. Ubiquitous systems are highly complex due to vast heterogeneity and adhoc interactions. Hence, it is important to understand that though ubiquitous systems prototypes have been implemented. However, mass-production of such systems is currently impossible. The reason for this negative answer is the challenge of failure-free or reliable operation. The proposed automata-based reliability model captures the various system states and transitions resulting in those states. Hence, as compared to its traditional counterparts the model is the accurate solution for reliability control of ubiquitous systems. The major benefit of the model is that it can be used as a control tool to monitor software execution to direct the system towards the correct path of execution, whenever, it takes an incorrect path. We are working on the practical implementation of this model. However, the only area of concern for ubiquitous systems shall be the complexity caused due to heterogeneity at various system levels. To handle the level-wise complexity the tool would have to work with different monitoring points for each level. Further the complex interaction and interconnection patterns in the ubiquitous architecture as well as the nomadic behavior of some nodes may be some issues affecting the successful operation of the proposed control tool.

For a ubiquitous system to be realized completely it needs to realize many generic properties, like reliability, context-awareness (application should comprehend the environment in which it is being used and adapt their operation to provide the best possible user experience. The very nature of ubiquitous computing is very sensitive as it is controlled by the devices and the environment on which it is run and hence involves the integration of many disparate technologies to meet the original design goals.

REFERENCES

[1] M. Satyanarayanan, Pervasive Computing: Vision and Challenges, IEEE Personal Communications, Vol. 8, No. 4, Aug. 2001, pp. 10 – 17.

[2] Bev Littlewood, MTBF is Meaningless in Software Reliability, IEEE Transactions on Reliability, 1975, pp. 82.

[3] B.M. Gouthami and P.Kumar, Effective Path Selection to Estimate Software Reliability, Special Issue of International Journal of Computer Applications, ICCCMIT, 2012.

[4] James Hamilton, Fault Avoidance vs. Fault Tolerance: Testing Doesn’t Scale, HPTS, 1999.

[5] Hal Wasserman, Manuel Blum, Software Reliability via Run-Time Result Checking, Journal of ACM, Vol. 44, No. 6, 1997, pp. 826-849.

[6] Mark Weiser. Some computer science problems in ubiquitous computing, Communications of the ACM, Vol. 36, No. 7, July 1993pp. 75–84.

[7] C.V. Ramamoorthy, Software Reliability- Status and Perspectives, IEEE Transactions on Software Engineering, Vol. 8, No.4, 1982, pp.354-371.

[8] M. Weiser, The Computer for the 21st Century, Scientific American , Vol. 265, No. 3, September, 1991pp. 94-100.

[9] J.Floch, S. Hallsteinsen, E. Stav, F. Eliasen, K.Lund, E. Gjorven, Using Architecture Models for Runtime Adaptability, IEEE Software,vol. 23, No.2, 2006, pp. 62-70.

[10] Amrit L. Goel, Software Reliability Models: Assumptions, Limitations and Applicability, IEEE Transactions on Software Engineering, Vol. SE11, No. 12, 1985, pp. 1411-1423.