DOI: 10.1007/s11633-019-1214-7

Hidden Markov Model Approach for Software Reliability Estimation with Logic Error

R. Bharathi[1,2] R. Selvarani[3]

[1] Department of Computer Science, PES University, Bangalore 560100, India

[2] Research scholar, Visvesvaraya Technological University, Belagavi 590018, India

[3] Department of Computer Science, Alliance University, Bangalore 562106, India

Abstract: To ensure the safe operation of any software controlled critical systems, quality factors like reliability and safety are given utmost importance. In this paper, we have chosen to analyze the impact of logic error that is one of the contributors to the above factors. In view of this, we propose a novel framework based on a data driven approach known as software failure estimation with logic error (SFELE). Here, the probabilistic nature of software error is explored by observing the operation of a safety critical system by injecting logic fault. The occurrence of error, its propagations and transformations are analyzed from its inception to end of its execution cycle through the hidden Markov model (HMM) technique. We found that the proposed framework SFELE supports in labeling and quantifying the behavioral properties of selected errors in a safety critical system while traversing across its system components in addition to reliability estimation of the system. Our attempt at the design level can help the design engineers to improve their system quality in a cost-effective manner.

Keywords: Hidden Markov model (HMM), reliability, logic error, safety critical, software failure.

1 Introduction

Mostly all recent safety critical systems are driven by software especially for control and monitoring. Since software has become dominant, hence inevitably it has to maintain its quality, cost and time-to-market[1]. Moreover, the software tends to fail when it is least anticipated. It is reported that to ensure that a safety critical software is reliable would cost 7 to 20 times higher than ensuring a conventional software[2]. The controlling software is developed with utmost care expecting it to be an error free product. Is this expectation true for all cases? In reality, there is no error free software product. It doesn't mean that if the software doesn't fail, it would not have any fault or error. Definitely, there would be a hidden fault that might have happened in any one of its phases of the software development life cycle. When a fault gets activated, it does not cause failure instantaneously, rather it causes an internal state in the system called error that deviates from the actual internal state, which usually cannot be perceived by the user[3]. In addition, a particular error type can develop into other error types as a temporal component before an occurrence of final failure.

It is necessary to explore and identify how these errors are likely to propagate across the system components and cause failure, so that reliability can be estimated. To achieve reliability in a cost-effective manner, it is necessary to integrate the knowledge, data and models of any critical system, that aids design engineers to devise for likelihoods, redundancies and hypothetical changes right from the early design stage. The categories of failures due to software errors and bugs are discussed only when disasters happen.

A software error might cause serious consequences in the case of safety critical applications that are present in automotive systems[2]. In recent past, the software has become the most dominant part of automotive systems[4, 5]. The controlling software of these systems is application software that is integrated with the hardware and it is the pivotal component of human safety systems[6]. The main issue in developing automotive software is its quality. The failure of these systems will cause severe damages or loss, so most of the software errors lead directly to car recalls. It is found that one-third of the recalls in the recent past is due to software errors[7, 8]. An error is a part of the system state, which is liable to lead to subsequent failure(s). An error has the property of propagating and transforming into other error types before it encounters a failure. The functional relation between errors and failure is called as error propagation[3].

Errors which are not eluded in the early detection efforts and do not surface until the software is operational

Research Article
Manuscript received July 24, 2019; accepted November 27, 2019; published online February 22, 2020
Recommended by Associate Editor Xun Chen

are called Persistent software errors[9]. Glass in his article had listed out two classes of errors: errors of omission and errors of combinatorics[10]. If necessary logic is left out of the software, it is termed an error of omission and if the software segments are executed not in expected order, it is termed an error of combinatorics. Further, he has mentioned that 35% of errors are due to omission of required logic[10]. These errors are found to be troublesome and costly and even the test coverage analyzer could not help in avoiding. Software errors are classified in three notions: source, severity and cause. The software development phase in which the error was made, namely requirement, design, coding, testing and maintenance falls under the category of "source" of error. Under the "severity" category, there are three errors: minor, major and critical. Under the notion "cause", errors are classified as computational, logic, data definition, data handling, database, interface, operation, and documentation[11]. Moreover, it is found that under this category, 38.5% errors happen due to logic[11].

Software logic errors might happen if the requirements are either incomplete or inconsistent, errors happened during software design stage, or in implementation stage of any software development lifecycle. These logic errors might cause failures like infinite loops, incorrect calculations, abrupt returns, taking a longer time to complete routine execution, etc.[12] Feiler et al.[13] have detailed that logic error might happen either as design errors in the decision logic or algorithm or coding errors such as array index out of bounds. Design errors are often caused by incorrect assumptions about system operation, e.g., input x is always followed by input y.

The error can propagate in the form of no output, bad output, early/late output, etc.[13] Typically fixing a software logic error is not hard but diagnosing these bugs is most challenging[14]. Pressman in his design principles stated that design should be developed iteratively. In the beginning, iterations generally concentrate on refining the design and correcting errors, but later iterations should concentrate on making the design as simple as is possible[6]. System failure and reliability can be predicted by probing into the error events that have occurred.

Reliability models are developed to predict the residual errors that exist even after the acceptance testing[11]. Reliability assessment in the design phase is useful for identifying and correcting the critical zones. This goal is achieved by early determination of the contribution of all parts of the software in system failure. These parts could be components, the connection between components, or structural design blocks like loops, conditions, etc. Software architecture is usually expressed in a semiformal language such as unified modeling language (UML) or architecture definition language (ADL)[15]. However, to prevent ambiguities and for precise verification, it is required to transform semiformal models into formal models which have a strong mathematical basis such as the Markov chain (MC), petri net (PN), and Bayesian model[16].

The behavior of a logic error cannot be unique, it will be probabilistic in nature and depend upon the system. In a component-based system, it is assumed that there are dependencies among system components. Hence, if there is an error in any one of the components, it might transform and propagate into other components[17]. In any system, the fault causing a failure cannot happen immediately, the system has to traverse in various error states called error propagation which are invisible to the user[3]. The fundamental assumption is that the failure can be predicted through detecting errors in the system components and these errors are generally hidden. The various invisible error states can be investigated through observing the physical behavior of any safety critical system having injected with fault and simultaneously monitoring its observable critical parameters. The relationship between the observable critical parameters and the software error states can be mapped and visualized through hidden Markov models (HMM). HMM is a formalism[18] wherein the system being modeled is assumed to possess a Markov process. It can estimate hidden/unobserved parameters in a model, and hence it is suitable for our method in investigating the invisible factor, the software error events which help to predict the failure[19]. Henceforth, we propose a new framework called software failure estimation with logic error (SFELE). It is a data driven approach to determine unobservable software behaviors of safety critical software systems and offers an opportunity to estimate its reliability.

1.1 Paper outline

The following sections of the paper are structured as follows: The related works are presented in Section 2. A basic study on HMM is explained in Section 3. The estimation framework SFELE is introduced in Section 4. The automotive system anti-lock braking system (ABS) is introduced in Section 5. The framework evaluation is described in detail in Section 6. Results analysis is presented in Section 7. Threats to validity are discussed in Section 8. Conclusions on our work are presented in Section 9.

2 Related works

In the recent past, there has been a drastic rise in the maturity of software in safety critical applications. These applications are used in diverse areas, which include aircraft-control, nuclear reactors, real-time sensor networks, industrial automation, automotive systems and health care used by millions of communities across the globe directly or indirectly. The quality attributes like safety and reliability requirements are more challenging. To meet this criterion, the safety critical software systems (SCSS) are developed with rigorous validation and verification techniques to make it an error free product. Although the system has been well designed, coded, thoroughly tested and validated, there are many instances wherein critical and catastrophic failures have occurred[20].

The reason for failure in the Mars Polar Lander in 1999 was due to software error. While investigating this accident, it was found that a breakdown in the design process developed in the software condition, made the specific failure occur. The reason behind the USS Yorktown disaster during 1998 was due to "division by zero" error. This error happened because by mistake a crewmember of the guided-missile cruiser entered a zero. The error propagated and ultimately the ship's propulsion system was shut down. The cruiser was dead for several hours because the software did not check for the validity of inputs[21]. The reason behind all these incidents was due to software impairments: failure, error and fault.

Failures are labeled as content failure, timing failure and content & timing failure. When the content of the information delivered at the service interface is different from the expected is termed as timing failures. When the time of arrival or the duration of the information delivered at the service interface deviates from the expected, this is termed timing failures. When there is deviation both from content and timing, it is termed as content and timing failure[22]. Logic errors might surface in almost any facet of any nomenclature of errors. If the developers are not logically omniscient, we cannot expect to avoid logic errors even in a system having moderate levels of complexity[23]. An error occurs when a fault gets activated. In other words, "a fault is the hypothesized cause of an error, which may lead to a failure"[22]. It is believed that software performs deterministically so that it will produce the same output every time for the same range of input. But in reality, it is not always true. Mostly, the errors are either due to interaction of software with the hardware or response of the system[21].

Software faults are classified into many types. Avizienis et al.[22] listed three different types of fault in dependable systems as development faults, physical faults and interaction faults. Hamill and Popstojanova[24] have classified based on NASA software safety guidebook and ISO/IEC/IEEE 24 765 as "requirement faults", "design faults", "data problems", "coding faults" and "integration faults". Further, Duraes and Madeira[25] have categorized faults on the basis of orthogonal defect classification as assignment faults, checking faults, interface faults, algorithm faults and function faults. Hamill and Goseva-Popstojanova[24] summarized the distribution of different faults for safety-critical failures as 45% of failures are due to coding faults, 24% of failures are due to requirement faults, 15% of faults are due to integration & interface, and design faults contribute 12%. Fig. 1 depicts the taxonomy of error.

The other version of faults leading to failure is called bug and there are many categories available, to name a few Mandelbugs, Bohrbugs, Heisenbugs, etc.[26] Mandelbugs have the error propagation property and it has a phenomenon of causing the error to propagate into partial failures thereby increasing the total time the system is running. This actually happens due to the accumulation of error states[27].

"System reliability is defined as the ability of a system to perform and maintain its required functions under nominal and anomalous conditions for a specified period of time in a given environment" and "System reliability is typically expressed by a failure-probability density function over time"[13]. In the recent past, for software reliability estimation at the design stage, researchers considered the impairment attributes namely fault propagation, error propagation and failure propagation[21].

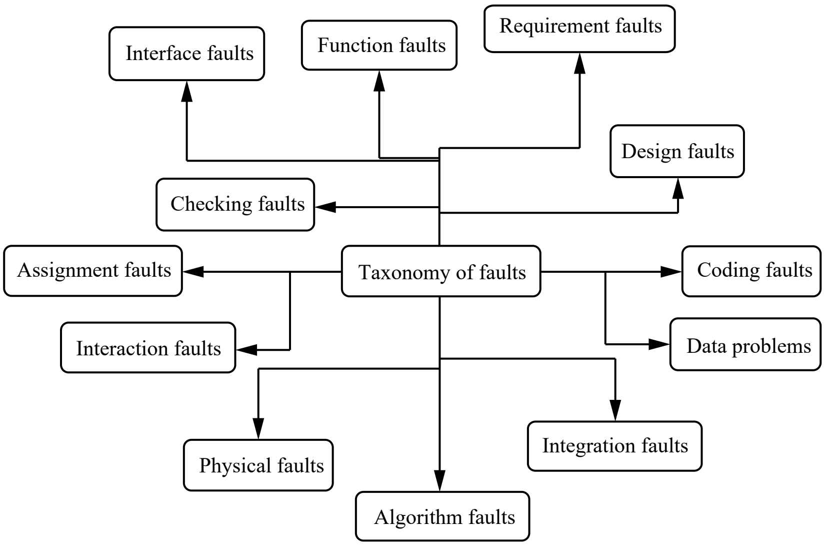


Fig. 1 Taxonomy of faults

Fault propagation is an incorrect or erroneous intermediate result that is passed/propagated to the downstream components[28, 29]. "Error propagations represent failures of a component that potentially impact other system components or the environment through interactions, such as communication of bad data, no data, and early/late data. Such failures represent hazards that if not handled properly can result in potential damage"[13]. Many researchers have worked on error propagation[22, 30–33]. Fiondella and Gokhale[33] in their paper have modeled the error propagation and recovery among components to analyze the software reliability based on its architecture.

A reliability and availability worst-case prediction model for embedded systems through simulation has been proposed[34]. The constraints for that model shows clearly that predicted values might not perfectly match with post-development estimated reliability and availability values achieved during system testing. Since the proposed model does not consider actual system components during prediction process, here in the proposed method system functions are realized through functionally equivalent generic components. Different execution scenarios[35] are used to test the system functions and the worst-case reliability and availability predictions are made based on simulations. The method proposed here is a cost effective method and helps in the reliability improvement process at the end of the development cycle force to considering these ad-hoc solutions thereby reducing time and cost[34].

In general, researchers use simulation models to evaluate software architecture. Formal description is used to represent the executable model of architecture and through this behavior of the software system can be explored before implementing into a real time product. In other words, using the executable model, the architecture can be evaluated. This model can be simulated using the Simulink environment. Simulink is a flexible tool to visualize the system functions at the implementation level and it accommodates in providing the most detailed descriptions of a functional block[7]. Hosseinzadeh-Mokarram et al.[16] proposed a model for software reliability assessment at early design stage using the colored Petrinet. They predicted the reliability during the execution of various scenarios using UML sequence diagram.

The safety critical systems are claimed to be dependable systems, which expect zero error tolerance. Automotive systems fall under the category of safety-critical systems anywhere these failures might cause severe damages or loss. Traction control, stability control, anti-lock braking and cruise control are software driven subsystems in the automotive area. These subsystems read input from sensors, make computations according to its respective control laws, and output a control value to maintain the state of the engine, throttle position, or the brakes as required[15]. The number of lines of code of these automotive controlling software lies somewhere between hundreds and millions, and can make the wrong decision if there are some hidden faults that might trigger a problem which were not identified and mitigated during the testing phase. These situations prompted researchers to predict the reliability of the software systems, in particular the worst-case prediction during the early design stage itself. Automotive systems have a robust software having constituents like methods, practices, and algorithms designed with higher quality of performance, which will prevent the activation of faults into error and eventually into failure. Model based software using Matlab and Simulink is vital in the automotive sector which is highly appropriate to represent the physical characteristics of the system to be controlled and further to look into the behavior of the control system[13].

It is recommended that fault injection can be used in a suitable simulation environment to determine what undesirable outputs the component can produce, and what inputs lead to those outputs, further to find the hidden flaws as well[36]. Researchers used fault injection techniques for reliability prediction and estimation. Markov modeling techniques are recommended for software reliability[36, 37]. A Markov based error propagation model has been built to find the reliability of component based software systems[26], but here the description about the type of error is not mentioned. HMM together with complex event processing has been used to predict on-line failure prediction in safety-critical systems[38]. In our proposed framework, we have used HMM for estimating failures and reliability.

3 Hidden Markov model

Markov models, also named as Markov processes or Markov chains, were introduced by a Russian mathematician Markov in [39] used to construct a stochastic model. Generally, they deal with finite set of states where each of them is associated through transition probability distributions. These distributions describe how a system evolves from one state to another over a period of time. A hidden Markov model is a statistical model having two stochastic processes, wherein the system being modeled will hold the Markov process with hidden/unobserved states. These hidden states are statistically organized through a probability distribution called "transition probability distribution", and assumed as a first order Markov model. The observable variables called "emissions" are probabilistic functions of the hidden states and it is the second stochastic process. One of the most comprehensive explanation is provided by Rabiner[18]. The HMM having double stochastic layers namely hidden states and emissions can be explained in Table 1.

A system at any time will exist in any one of a set of N distinct states: S_1, $\cdots$, S_N. Any state at time t is denoted as q_t, for $t = 1, 2, \cdots, n$, so, e.g., $q_t = S_i$ denotes that at time t, the system remains in state S_i. The sys-

Table 1 HMM symbols

$S = \{S_1, \cdots, S_N\}$; Set of all possible hidden states.

$O = \{O_1, \cdots, O_M\}$; Set of emission symbols.

$q = \{q_1, q_2, \cdots . q_n\}$; Set of values for hidden states.

$v = \{v_1, v_2, v_3, \cdots . v_{\rm n}\}$; Set of values possible in emissions termed as observation vector.

$A = \{a_{ij}\}$ is the hidden state transition probability distribution, where $a_{ij} = P(q_t = S_j\, q_{t-1} = S_i)$; such that $1 \le i,\ j \ge N$ and $a_{ij} \ge 0$.

$B = \{b_j(v_m)\}$, is the emission probability distribution, where $b_j(v_m) = P\{O_t = v_m\, q_t = S_j\}$; such that $1 \le j \le N$ and $1 \le m \le M$.

π_0 is the initial state distribution $\pi_0 = \{\pi_i\}$ for hidden states.

Hidden Markov model is represented as $\lambda = (A, B, \pi_0)$.

tem moves to a state with a given probability, depending on the values of the previous times and it is represented as

$$P\left(q_{t+1} = S_j | q_t = S_i, q_{t-1} = S_k, \cdots\right) = \\ P\left(q_{t+1} = S_j | q_t = S_i\right). \tag{1}$$

Equation (1) conveys that, as per the first order Markov model, given a present state, the future is independent of the previous state. The transition probability "A" as given in Table 1 says that a transition from S_i to S_j has the same probability no matter where it happens or when it happens in the emission sequence. It is represented in a $N \times N$ matrix where the sum of each row elements must be one.

In HMM, the hidden states are not observable, for every state, there will be an emission that is recorded and this will be a probabilistic function of hidden state. "B" is the emission probability distribution and its mathematical equation is depicted in Table 1. The transition probability represented as a stochastic automaton is depicted in Fig. 2.

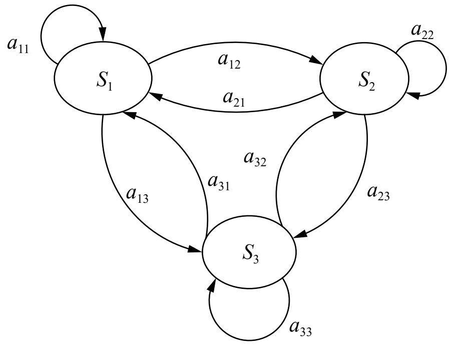


Fig. 2 Hidden state probability distribution

Once the HMM λ is constructed, then the model λ can be implemented through the following approach:

1) The probability of the emission sequence can be computed using the "forward and backward algorithm".

2) Given a sequence of emissions, the most likely states sequence can be searched. This can be found using the "Viterbi algorithm". An HMM can be unfolded in time as a lattice or trellis showing all possible trajectories using this algorithm.

3) Given an emission sequence and on a set of possible models, the HMM model parameters can be adjusted such that the model will fit for the intended application. This is realized through the "Baum-Welch algorithm".

HMMs have been used successfully in many areas, which includes mechanical, electronics and computer science[40]. Honamore and Rath[41] have implemented HMM together with fuzzy logic for web services reliability prediction. HMM was used to diagnose the health of patient for wearable device[42]. Dorj et al.[43] have presented a data-driven approach for anomaly detection in electronic systems using Bayesian HMM classification. Clustering multivariate time series in the healthcare domain has been carried out using HMM[44]. To monitor predictive maintenance of diesel engines, HMM algorithms have been used in [45]. HMM is used to predict the software failures by identifying the special patterns of errors[17]. Durand and Gaudoin[46] have established a framework using HMM for software reliability modeling and prediction. HMM is a classification technique used to estimate the unknown/hidden parameters. Hence, it provides an opportunity to detect the unobservable hidden software behaviors of any system.

4 The proposed SFELE framework

The proposed framework is based on a data driven approach called "software failure estimation with logic error (SFELE)" and it is presented as shown in Fig. 3. It is composed of three different phases namely process phase, analysis phase and end phase. Broadly, our framework uses a stochastic model to estimate the failure and reliability of any safety critical software system. We choose to use a stochastic process namely Markov models for examining the temporal behavior of software. In the Process phase, the architectural model is executed and observable critical parameters (v_1, v_2, v_3, $\cdots$, v_n) are measured. In parallel, the hidden controlling software behavior pattern is explored, and particularly examined for system specific software error patterns.

In analysis phase, the observed critical parameters (v_1, v_2, v_3, $\cdots$, v_n) are quantized into emissions O_1, O_2, $\cdots$,

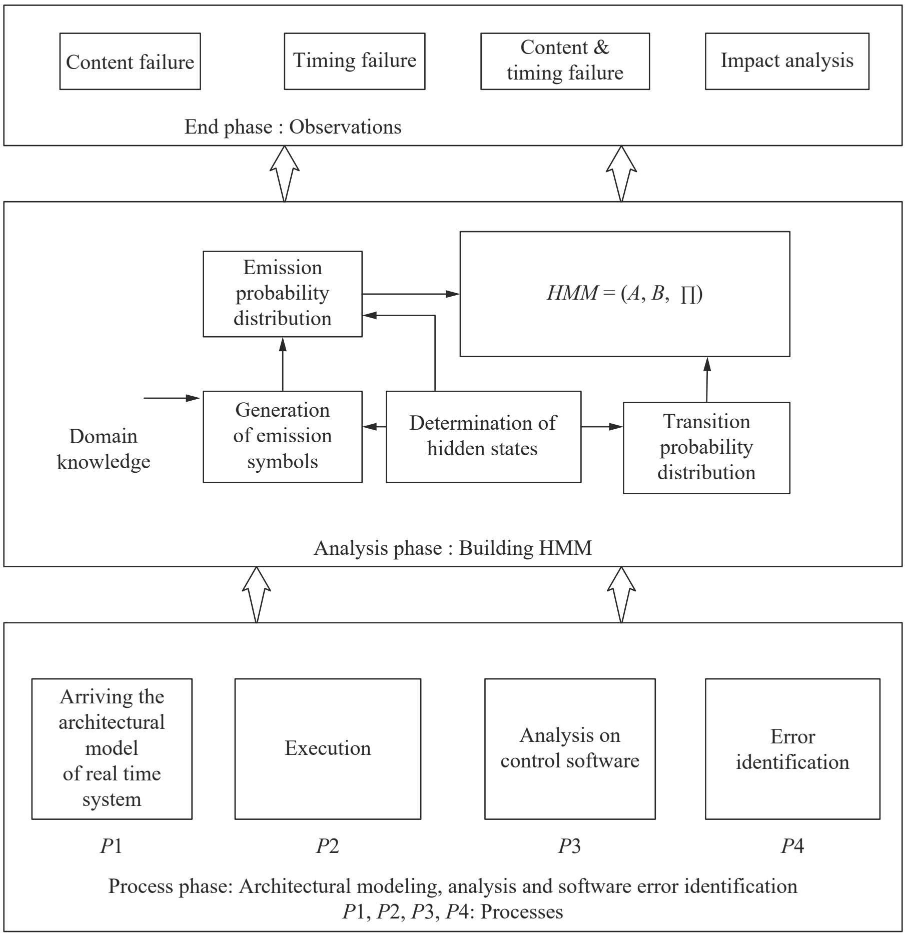

Content failure
Timing failure
Content & timing failure
Impact analysis
End phase : Observations
Emission probability distribution
HMM = (A, B, ∏)
Domain knowledge
Generation of emission symbols
Determination of hidden states
Transition probability distribution
Analysis phase : Building HMM
Arriving the architectural model of real time system
Execution
Analysis on control software
Error identification
P1
P2
P3
P4
Process phase: Architectural modeling, analysis and software error identification
P1, P2, P3, P4: Processes

Fig. 3 SFELE framework

O_M, based on the domain knowledge. The software error patterns are analyzed further and categorized into different software error states namely S_1, S_2, $\cdots$, S_N. These software error states are unknown/hidden operation of the system, which brings the uncertainty if there are any in the software reliability estimation process.

Hence, our framework constructs a machine-learning model called HMM with available information called emissions (O_1, O_2,$\cdots$,O_M) whose readings are associated with its software behavior. The transitions across the software error events are summarized through a matrix called the transition probability distribution. Every software state is associated with specific emissions. The stochastic behavior of any safety critical software, the hidden software events are mapped with emissions and arrived with a probability distribution called the emission probability distribution.

Once the emission states, hidden states, initial state, transition probability matrix and emission probability matrix are available, the HMM λ is re-estimated using the Baum-Welch algorithm. In the end phase, the encountered failures are identified and the impact of these failures is estimated to calculate the system reliability. Once the model is built, it helps to explore types of failure by visualizing the temporal sequence of error and dependencies among components. This is attained through the Viterbi algorithm and it actually supports diagnosing the software behavior by observing the operation of the safety critical system and helps to find out the possible trajectories. The proposed data driven framework is evaluated through an automotive anti-lock braking system called ABS.

5 Case study: ABS

ABS is an automotive system that is designed to control and to maintain the performance of a vehicle. The ABS either shares a single CPU along with the other systems, namely traction control, stability control and cruise control or an individual CPU. The ABS ensures that maximum braking is accomplished even under adversarial conditions such as skidding on rain, snow, or ice. These brakes function by sensing slippage at the wheels during braking and continually adjusting braking pressure to ensure maximum contact between the tires and the road. The system computes relative slip value and based on this relative slip value a control signal is generated which controls the activation/deactivation of the brake pressure valve in accordance with ABS principle, a detailed description can be found in [47].

The component model of ABS is represented in Fig. 4. Functionality of each component in this system contributes to the expected performance. The model is visualized through six components namely Comp-1 to Comp-6. These six components interact with each other during its operational period. Comp-6 is the controller component and it computes the brake pressure according to the ABS principle. The function of Comp-3 is to compute relative slip as per (1), by accepting input ω_v and ω_w from Comp-2. The values ω_v and ω_w are recalculated in Comp-6 and Comp-1 respectively using special integrators. The frictional force F_f is computed by Comp-4. In Comp-1, F_f is divided by the vehicle mass to produce the vehicle deceleration and it integrates to obtain ω_v. The desired relative slip is compared with slip for maximum traction at Comp-5. The value of desirable slip, has to be 0.2, the number of wheel revolutions equals 0.8 times the number of revolutions under non-braking conditions with the same vehicle velocity. The slip is calculated from wheel-speed and vehicle-speed, and it is given by $Slip = 1 - \frac{\omega_w}{\omega_v}$, where ω_v is the vehicle-speed and ω_w the wheel-speed.

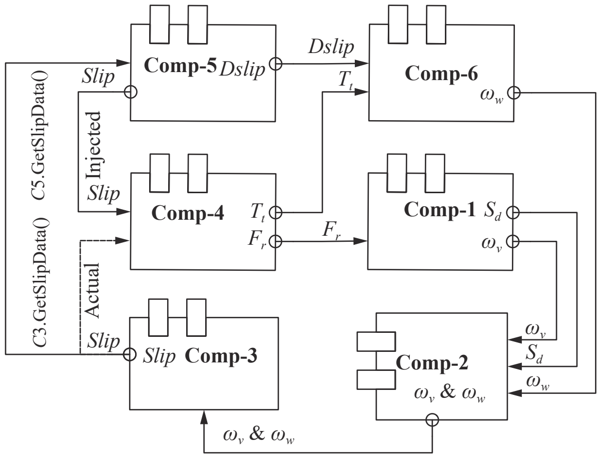


Fig. 4 Component model of ABS

For the failure analysis, four critical parameters of ABS are considered. The first critical parameter is slip and second critical parameter is the stopping-distance (S_d), which is defined as the distance covered by the vehicle from the instant the brake is applied, to the instant the vehicle is stopped completely. The third parameter is tire-torque (T_t). The fourth observation is for comparing wheel-speed ω_w and vehicle-speed ω_v.

6 SFELE framework evaluation

6.1 Process phase

ABS is executed with a running speed of 88 km/h, and the system is assumed to be absolute. The actual data flow for slip is from Comp-3 to Comp-4 and it is shown by the dotted line in Fig. 4. The operational scenario is illustrated through a sequence diagram in Fig. 5, which depicts the ABS functionality for time instants $t = 12.832$ s, 12.854 s and 14.008 s. For an absolute system, the vehicle stops absolutely at time $t = 14.008$ s. The four critical parameters S_d, T_t, $Slip$ and ω_v & ω_w are observed over the complete execution time and sample readings are listed in Appendix Table A1.

To explore the behavior of ABS when there is a logic error, the system is injected with a logic fault. A logic error might happen either as design errors in the decision logic or algorithm or coding errors. The logic fault is injected altering the dataflow path for slip from Comp-5 to Comp-4 as shown in the solid line labeled as injected in Fig. 4. This causes undefined program behavior. As mentioned in Section 1, the system took more than the expected time to complete its operation resulting in wrong and late output. Software logic errors might lead to failure conditions for example infinite loops, abrupt returns, taking a longer time to complete routine execution and incorrect calculations[13]. This error can be mapped into externally observed error propagations in the form of no output, bad output, early/late output, etc. The critical parameters S_d, T_t, $Slip$ and ω_v & ω_w are observed over the execution period and it is expressed through the sequence diagram as shown in Fig. 6 and the respective dataset is tabulated in Appendix Table A2. The statistics of this error scenario with respect to the absolute scenario is tabulated at time $t = 12.832$ s, 12.854 s and $t = 14.008$ s, $t = 14.3668$ s and it is shown in Table 2.

The performance of the ABS system with no error and with injected logic faults over the critical parameters S_d, T_t, Normalized $Slip$ and ω_v & ω_w are depicted in the graph as shown in Figs. 7–10. Fig. 10 shows the value of expected stopping distance S_d as 720.8 in dotted line and the stopping distance as 722 when associated with logic fault. Moreover, it shows that the time taken for stopping the vehicle is 14.4 s, which is more than the normal operation of 14.008 s. This is an abnormal situation and the failure can be called as content and timing failure. This is an anomaly, which is defined as a condition that is not anticipated.

Fig. 8 shows the comparison between the wheel speed and vehicle speed ω_v & ω_w as per vehicle dynamics, the wheel speed tends to be lower than vehicle speed. Instead in the time duration 12.832 s to 12.854 s, wheel speed is slightly greater than vehicle speed. The difference is found to be small when compared with the ABS having a timing fault[48]. The failure here is found to be marginal. Fig. 9 shows the tire torque (T_t) values. When the system is absolute, the value of the tire torque never goes negative. But, when it has a timing fault, it is found that it possesses negative values during few execution cycles[48]. After a few executions the tire torque returns to habitual values, but the influence is propagated as a Mandelbug and timing error, disturbing the performance

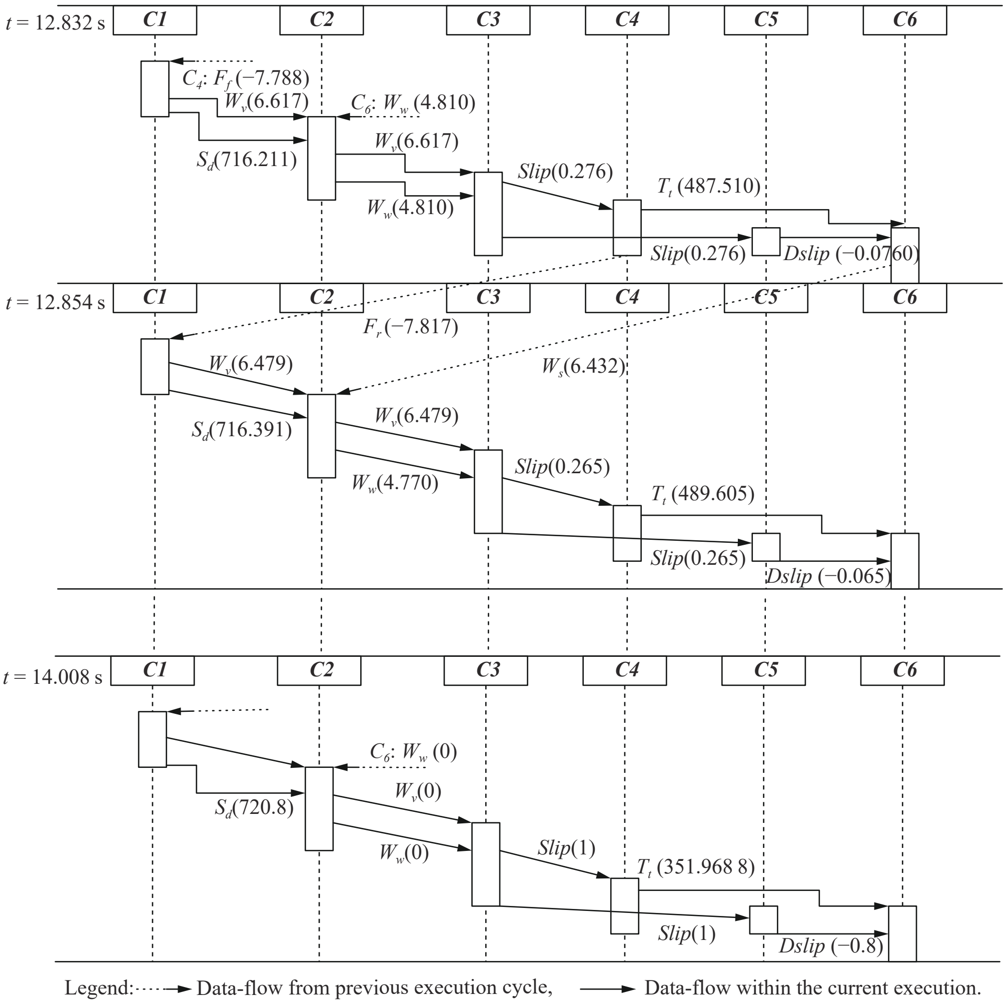


Fig. 5 Sequence diagram for absolute system

of the system. With these observables, we explored the performance of the system and mapped to the software behavior.

6.2 Analysis phase

The critical parameters are measured as a sequence of data, during the ABS operation over a specified period of time when the brake is applied. The sequence of measured data for the critical parameters S_d, T_t, *Slip* and ω_v & ω_w are recorded for its values. The emission symbols O_1, O_2, O_3 and O_4 are generated according to the variations in the measured values of S_d, T_t, Slip and ω_v & ω_w as shown in Table 3.

By overlooking the sequence of measurement data, we could detect that the ABS system has software malfunctioning. The software behavior of system is analyzed to extract the software error patterns. Investigating each execution cycle as shown in Figs. 5 and 6 provides knowledge of how a logic fault gets activated and this error propagates across ABS components. It is found that there are three different software errors occurring at different instances, namely logic error, Mandelbug and timing error.

When the system with a logic fault starts executing, all critical parameters were within the threshold as depicted in Table 3. It is claimed as error free state S_1. At $t = 12.832$ s, slip becomes negative, due to value error and ω_w becomes more than ω_v which is logic error, here it is categorized as S_3. The situation continues till $t = 12.854\,0$ s. The system runs in a normal mode till $t = 12.958\,0$ s. Later at $t = 12.958\,0$ s, tire torque becomes negative, which is due to a Mandelbug and it is marked as state S_4. This continues for eight execution cycles and the system recovers. Subsequently at $t = 13.712\,0$ s, the stopping distance increases beyond its estimated value of 720.8, this is due to the existence of Mandelbug and timing error and the state of the software being marked as S_2.

As per the listed rules in Table 3, the emission symbol O_1 is assigned when every critical parameter is as expected. The symbol O_2 is assigned when the S_d goes beyond the estimated value of 720.8, while the other three parameters remain as expected. When $\omega_v > \omega_w$ becomes false, it is named as O_3. The symbol O_4 is assigned when slip goes negative and $\omega_v > \omega_w$ becomes false. Having identified the emission symbols and hidden software

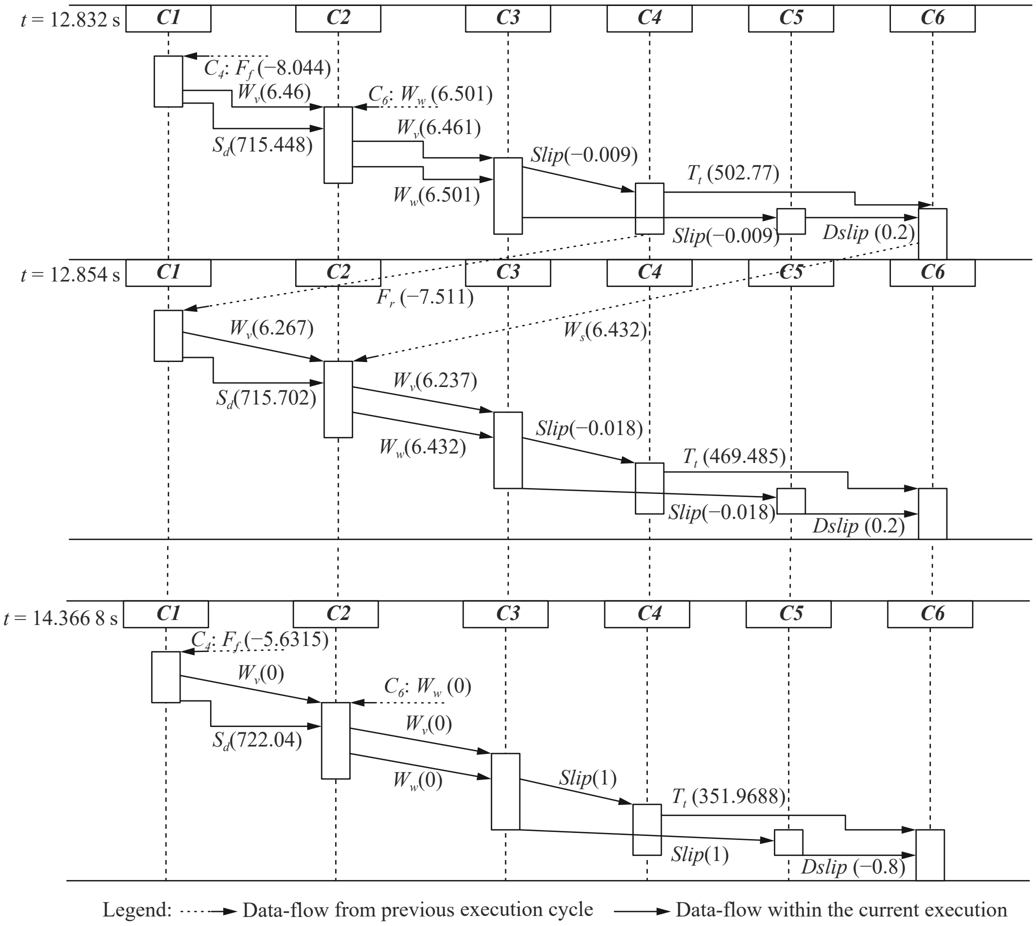


Fig. 6 Sequence diagram for ABS operation with logic error

states, by exploring the operation of ABS with and without fault. The emission symbols of HMM are categorized according to the thresholds as shown in Table 3 and mapped it to the various software error states to build the HMM model for the ABS system with fault.

6.2.1 Building the HMM structure for faulty system

The ABS operations are recorded for timing sequences t_1, t_2, t_3, $\cdots$, t_n. At any instant, the system changes its state among the four mentioned states S_1, S_2, S_3 and S_4 as described in Section 6.2. A state can be a set of events characterizing a system at a given condition or activity. A transition is a passage from one state to another, whose transition probability is the probability of undergoing this transition[49]. The hidden state transitions matrix is the standard way of representing Markov chains. For the dataset as shown in Table A2, the emission symbols are generated as per the rules listed in Table 3 and the corresponding software error states are counted. The hidden transition probability (A) and emission probability (B) matrices are computed[18].

The probability of transition for changing a state from one to another is denoted by a_{ij}. For example, if $i = 1$ and $j = 2$, then a_{ij} represents the probability of changing the state from S_1 to S_2. The HMM parameters A, B, π are adjusted in order to maximize $p\left(\dfrac{O}{\lambda}\right)$ using the Baum-Welch algorithm[19] and the graphical representation is shown in Fig. 11. Hence for our model having 4 states, there are 16 possible transitions and represented in matrix called hidden states probability distribution as depicted in Table 4.

The gross performance of the model is computed for a specific test dataset[50] and the results found to be satisfactory. The performance variables namely sensitivity, precision and F-measure are depicted as shown in Table 5 for four states.

6.3 End phase

In end phase, we will be concentrating on our four different components: content failure, timing failure, content & timing failure and impact analysis on a system as a derivative of hidden Markovian model applied on the real time system ABS. The following discussions on various failures and its parameters depict the complete scenario of the effect of logic error in the real time system.

6.3.1 Content failure

The observable parameters delivered by the real time system ABS deviates from implementing system function. As shown in Table 4, when the system traverses in the states either at S_3 or S_4, the observable parameters slip

Table 2 Tabulation for statistics of error scenarios

	Absolute system					System with logic error (LE)				
	ω_v (m/s^2)	ω_w (m/s^2)	S_d(m)	*Slip*	T_t(N·m)	ω_v(m/s^2)	ω_w(m/s^2)	S_d(m)	*Slip*	T_t(N·m)
	Time $t = 12.832$ s					Time $t = 12.832$ s				
$C1$	6.617		716.21			6.46		715.44		
$C2$	6.617	4.810				6.46	6.501			
$C3$				0.276					–0.009	
$C4$					487.51					502.77
$C5$										
$C6$		6.432								
	Time $t = 12.854$ s					Time $t = 12.854$ s				
$C1$	6.479		716.39			6.267		75.702		
$C2$	6.479	4.770				6.267	6.432			
$C3$				0.265					–0.018	
$C4$					489.605					469.48
$C5$										
$C6$		0								
	Time $t = 14.008$ s					Time $t = 14.3668$ s				
$C1$	0		720.8			0		722.04		
$C2$	0	0				0	0			
$C3$				1					1	
$C4$					351.968					351.968
$C5$										
$C6$		0					0			

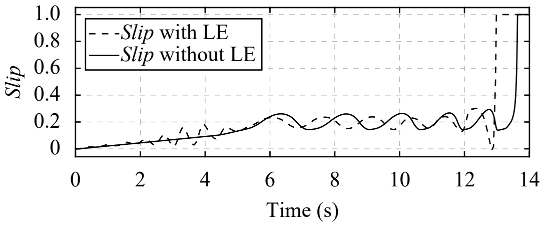


Fig. 7 Normalized *Slip* versus time

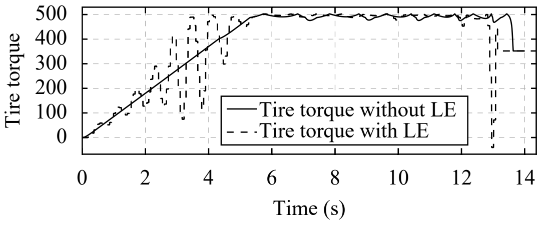


Fig. 9 Tire torque (T_t) versus time

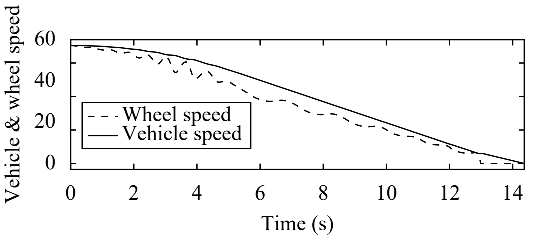


Fig. 8 ω_v & ω_w versus time

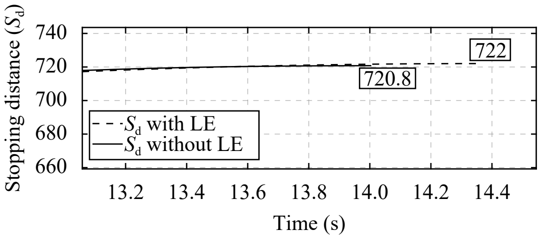


Fig. 10 Stopping distance (S_d) versus time

value becomes negative and the wheel speed is slightly greater than vehicle speed that is an abnormal dynamic. After few milliseconds the system resumes its value as expected. The behavior of the system in this way is due to content failure and is revocable.

6.3.2 Content & timing failure

It is observed that, when the real time system traverses from state S_1 to state S_2, it encounters content & timing failure as shown in Fig. 11, which is a critical failure and it is irrevocable. At the transition state S_2, the

Table 3 Rules for generating emission symbols

Association of software states with emission symbols	S_1	S_2	S_3	S_4
	O_1	O_2	O_3	O_4
S_d	<720.8	>720.8	<720.8	<720.8
T_t	>0	>0	<0	>0
Slip	>0	>0	≤0	≤0
$\omega_v > \omega_w$	True	True	False	False

Table 4 Hidden states transition probability distribution

$A = \{a_{ij}\}$	S_1	S_2	S_3	S_4
S_1	0.9900	0.0013	0.0013	0.0057
S_2	0.0130	0.9600	0.0130	0.0130
S_3	0.2500	0.1300	0.5000	0.1300
S_4	0.1670	0.0830	0.0830	0.6700

critical parameter S_d is observed to be exceeding the expected value. Under this scenario, the system takes much longer time to complete its execution as discussed in Section 6.2. The errors occurring at this stage are identified as Mandelbug and timing error.

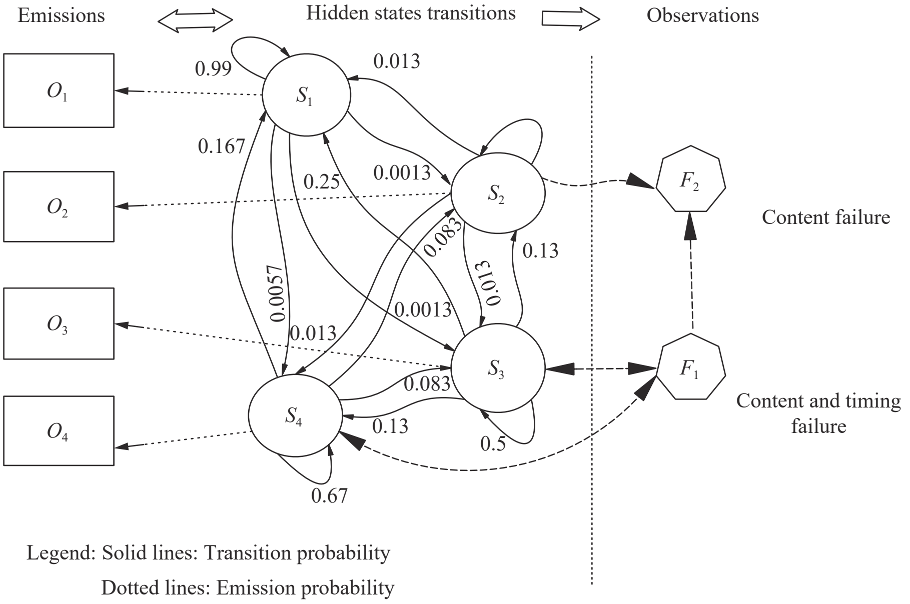
6.3.3 Timing failure

The service delivered may be either too early or late and this failure is not encountered in this scenario.

6.3.4 Impact analysis on the system

All these encountered failures have an impact on the performance of the entire system and this can be discussed in the view of reliability of the system. Reliability is the probability that the system does not confront an error state. The reliability can be estimated by a steady state vector[51]. The steady state or equilibrium vector helps to assess how the software error behavior affects the overall system reliability[52]. A finite Markov chain is a sequence of probability vectors π_0, π_1, π_2, π_3, $\cdots$, π_n, where π_0 is the initial state vector. For our real time system $\pi_0 = (1\ 0\ 0\ 0)$, since the system starts its execution in the error free state, then π_1, π_2, π_3, $\cdots$, π_n are calculated as, $\pi_1 = A\pi_0$, $\pi_2 = A\pi_1$, $\cdots$, $\pi_n = A\pi_{n-1}$ and attains steady state vector[52]. The steady state vector of the transition matrix A is the unique probability vector that satisfies the following equation,

$$\pi_{t_{ss}} = A\pi_{t_{ss}},\ \text{where } t_{ss} = \text{time of steady state} \tag{2}$$



Legend: Solid lines: Transition probability
Dotted lines: Emission probability
Dashed lines: Observations: Failure

Fig. 11 HMM structure for faulty ABS system and its observations

Table 5 Performance measure

Class	Sensitivity/True positive rate	Precision/ Positive predictive value	F-measure
Class1	0.99	0.99	0.99
Class2	1	0.97	0.98
Class3	0.75	0.75	0.75
Class4	1	0.88	0.94

$$Reliability_{worst\ case} = 1 - \sum_{i=2}^{4} \pi(S_i). \qquad (3)$$

In (3), $\pi(S_i)$ is the steady state probability vector. The reliability factor depends on the probability of being in a failure at steady state t_{ss}. The Markov property assumes that the probability of transition to the next state at time t depends on the system at previous state at time t–1 and is independent from its past history. Under this assumption, the reliability is estimated on the probability of being in a failure state and is independent of the exclusive path(s) taken to reach the particular failure state[52].

7 Result analysis

7.1 Unfolding software error states

The HMM model can capture various software error states and allows us to make inferences about the performance of the software at each instance. For example, a logic fault in the design can lead to an erroneous computation for specific values of program variables S_d, $Slip$, ω_v, ω_w and T_t. The software can use this incorrect result internally for further computations, in which case the error propagation leads to additional errors. In the time between the fault activation and the final failure occurrence, the system traverses different error states in its error propagation path. The various error states S_2, S_3 and S_4 are visualized in the trellis diagram as presented in Fig. 12.

7.2 Estimation of failure and reliability

The failure prediction approach is designed in terms of temporal behavior of error occurrence and its transformations. A failure occurs only when the system makes incorrect calculations due to some existing error or the actual execution time is not matching the expected execution time. In our experimental analysis, we found that two types of failure occurred. At time t = 12.832 s, content failure occurred[23] and this exists for 2 ms. This is a transient in nature and it is detected by overlooking the corresponding error state S_3. Again, at t = 12.958 s due to the error state S_4, the system experiences a failure. When the system encounters state S_2 at t = 13.712 s, the ABS system undergoes content and timing failure and it is a permanent failure. Here, the relationship between fault, error and failure is estimated as the worst-case reliability of the system,

$$\text{Steady state vector} \ \ \pi_{ss} = \begin{array}{cccc} S_1 & S_2 & S_3 & S_4 \\ [0.861\,0 & 0.107\,5 & 0.008\,8 & 0.022\,7] \end{array}$$

$$Reliability_{worst\ case} = 0.861.$$

8 Threat to validity

The proposed framework might not be suitable for all other safety critical systems that are not included under the classification of automotive systems. The framework SFELE evaluation is concerned with the specific variables S_d, ω_v, ω_w, $Slip$ and T_t only. Further evaluation may be taken with other parameters also. There are other parameters also to be considered for precision in the evaluation in future. The recommended model λ with the principle of hidden Markov approach is built for the selected injected fault. The same model λ might not be fit for the same system with any other injected fault. The behavior of the real time system with various injected faults might not have maximum likelihood for the model λ.

9 Conclusions

We presented a data driven framework SFELE for the reliability estimation at the early design of the safety critical system. The framework is built extensively on an unsupervised machine learning technique "hidden Markov model". The model is checked for its performance, which

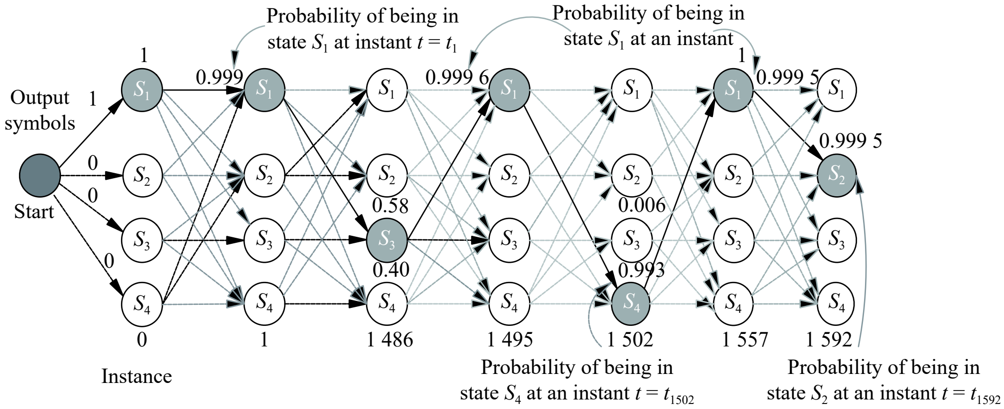


Fig. 12 Trellis: Error propagation path

gives satisfactory results. The framework is built in such a way that the outcomes are presented in a hierarchical way. The first outcome gives the underlying various software error states that the system is traversing within the time period of activation of logic faults to failure occurrence. It is found that the interacting system components propagates software errors namely logic error, Mandelbugs and timing error. The results are presented in a graphical representation called a Trellis diagram. The second outcome is finding out the type and nature of failure occurrence and it is found that the system experiences content, content & timing failure. The final outcome is the reliability estimation under the worst-case scenario, the ABS system with logic fault. Reliability estimation is not worthwhile if the estimation does not contribute to improving the system dependability. We believe that the effort of estimating reliability at the early design stage will help the software practitioners to build reliable safety critical software in a cost-effective manner. This approach helps in proactive fault management and helps the design engineers for effective support for developing any safety critical system.

Appendix

Table A1 Example dataset for ABS with no fault

Time(s)	*Slip*	ω_v(m/s^2)	ω_w(m/s^2)	S_d(m)	T_t(N·m)
0.0000	0.0000	70.4000	70.4000	0.0000	0.0000
0.0979	0.0008	70.3987	70.3426	8.6123	3.2057
0.3979	0.0072	70.3397	69.8360	35.0039	28.8054
0.6979	0.0142	70.1751	69.1811	61.3538	56.9775
1.9879	0.0445	68.2293	65.1962	173.2128	178.8186
1.9979	0.0447	68.2064	65.1582	174.0655	179.7658
3.1979	0.0730	64.5708	59.8559	273.8671	293.7197
5.9867	0.2390	49.6970	37.8202	475.3702	494.9717
7.9763	0.2201	37.1916	29.0046	583.4208	498.7636
8.9009	0.1683	31.3400	26.0647	623.0249	493.2568
9.8034	0.2323	25.6759	19.7103	655.2089	496.3080
10.9186	0.1572	18.6763	15.7409	686.1050	489.8912
11.9876	0.1430	11.9294	10.2231	706.5381	475.8240
12.8390	0.2851	6.5904	4.7111	716.3943	485.6872
12.8490	0.2821	6.5282	4.6866	716.4763	486.3012
12.8590	0.2785	6.4659	4.6654	716.5576	487.0331
12.9544	0.2000	5.8637	4.6910	717.2926	502.8125
12.9679	0.1820	5.7774	4.7262	717.3907	497.3684
12.9932	0.1542	5.6177	4.7513	717.5710	489.0035
13.0032	0.1461	5.5555	4.7441	717.6408	480.9867
13.0132	0.1416	5.4944	4.7163	717.7098	473.3986
13.0232	0.1395	5.4341	4.6759	717.7781	469.8311
13.7039	1.0000	1.3542	0.0000	720.6203	351.9688
13.8939	1.0000	0.4982	0.0000	720.8402	351.9688
14.0039	1.0000	0.0026	0.0000	720.8747	351.9688
14.0045	1.0000	0.0000	0.0000	720.8747	351.9688
14.0045	1.0000	0.0000	0.0000	720.8747	351.9688

Table A2 Example dataset for ABS with logic fault

Time (s)	*Slip*	ω_v (m/s^2)	ω_w (m/s^2)	S_d (m)	T_t (N·m)
0.0000	0.00	70.40	70.40	0.00	0.0000
1.2800	0.02	69.53	67.88	112.23	110.8606
3.9160	0.18	61.89	50.86	330.63	356.3574
5.9820	0.23	49.77	38.26	475.34	498.2446
7.9760	0.21	37.10	29.18	583.60	496.4340
8.9060	0.22	31.22	24.26	623.31	500.0544
9.9820	0.17	24.40	20.31	660.69	480.3883
10.9120	0.15	18.49	15.64	685.63	491.8359
11.9860	0.15	11.69	9.93	705.89	462.3852
12.8320	–0.01	6.46	6.51	715.51	499.8979
12.8340	–0.01	6.45	6.51	715.53	454.8094
12.8440	–0.01	6.39	6.43	715.61	454.8094
12.8540	0.00	6.33	6.35	715.69	454.8094
12.9580	0.51	5.88	2.91	716.48	–40.5181
12.9680	0.69	5.89	1.82	716.55	–40.5181
12.9949	1.00	5.90	0.00	716.75	–40.5181
13.0049	1.00	5.91	0.00	716.82	–40.5181
13.0149	1.00	5.91	0.00	716.90	–40.5181
13.0200	1.00	5.92	0.00	716.93	71.4510
13.7020	1.00	2.99	0.00	720.80	351.9688
13.7120	1.00	2.95	0.00	720.83	351.9688
14.3668	1.00	0.00	0.00	722.04	351.9688
14.3668	1.00	0.00	0.00	722.04	351.9688

References

[1] G. Carrozza, R. Pietrantuono, S. Russo. A software quality framework for large-scale mission-critical systems engineering. *Information and Software Technology*, vol. 102, pp. 100–116, 2018. DOI: 10.1016/j.infsof.2018.05.009.

[2] E. Kovacs. NIST tool finds errors in complex safety-critical software, [Online], Available: https://www.securityweek.com/nist-tool-finds-errors-complex-safety-critical-software, April 26, 2019.

[3] M. Grottke, K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and Rejuvenate. *Computer*, vol. 40, no. 2, pp. 107–109, 2007. DOI: 10.1109/MC.2007.55.

[4] H. Altinger, Y. Dajsuren, S. Siegl, J. J. Vinju, F. Wotawa. On error-class distribution in automotive model-based software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, IEEE, Suita, Japan, pp. 688–692, 2016. DOI: 10.1109/saner.2016.81.

[5] W. Mostowski. AUTO-CAAS: Model-Based Fault Prediction and Diagnosis of Automotive Software, Technical Report, Halmstad University, Halmstad, Sweden, 2016.

[6] R. S. Pressman. *Software Engineering: A Practitioner's Approach*, 7th ed., New York, USA: McGraw Hill, 2010.

[7] J. L. Boulanger, V. Q. Dao. Requirements engineering in a model-based methodology for embedded automotive software. In *Proceedings of IEEE International Conference on Research, Innovation and Vision for the Future in Computing and Communication Technologies*, IEEE, Ho Chi Minh City, Vietnam, pp. 263–268, 2008. DOI: 10.1109//RIVF.2008.4586365.

[8] M. L. Shooman. Bohrbugs, mandelbugs, exhaustive testing and unintended automobile acceleration. In *Proceedings of IEEE 23rd International Symposium on Software Reliability Engineering Workshops*, IEEE, Dallas, USA, pp. 5–6, 2012. DOI: 10.1109/ISSREW.2012.25.

[9] R. L. Glass. Persistent software errors. *IEEE Transactions on Software Engineering*, vol. SE-7, no. 2, pp. 162–168, 1981. DOI: 10.1109/TSE.1981.230831.

[10] R. L. Glass. Two mistakes and error-free software: A confession. *IEEE Software*, vol. 25, no. 4, Article number 96, 2008. DOI: 10.1109/MS.2008.102.

[11] J. B. Bowen. Standard error classification to support software reliability assessment. In *Proceedings of National Computer Conference*, ACM, Anaheim, California, USA, pp. 697–705, 1980. DOI: 10.1145/1500518.1500638.

[12] B. J. Czerny, J. G. D'Ambrosio, B. T. Murray, P. Sundaram. Effective Application of Software Safety Techniques for Automotive Embedded Control Systems, Technical Report 2005-01-0785, SAE International, Detroit, USA, 2005. DOI: 10.4271/2005-01-0785.

[13] P. H. Feiler, J. B. Goodenough, A. Gurfinkel, C. B. Wein-

stock, L. Wrage. Reliability Validation and Improvement Framework, Technical Report CMU/SEI-2012-SR-013, Pittsburgh Pa Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, USA, 2012.

[14] H. Pham. Software reliability and fault-tolerant systems: An overview and perspectives. *Handbook of Performability Engineering*, K. B. Misra, Ed., London, UK: Springer, pp. 1193–1208, 2008. DOI: 10.1007/978-1-84800-131-2_72.

[15] J. J. Hudak, P. H. Feiler. Developing AADL models for control systems: A practitioner′s guide, [Online], Available: http://www.sei.cmu.edu/reports/07tr014.pdf, 2007.

[16] A. Hosseinzadeh-Mokarram, A. Isazadeh, H. Izadkhah. Early reliability assessment of component-based software system using colored petri net. *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 27, pp. 2681–2696, 2019. DOI: 10.3906/elk-1805-82.

[17] F. Salfner. Predicting failures with hidden Markov models. In *Proceedings of the 5th Europe Dependable Computer Conference*, pp. 41–46 2005, [Online], Available: http://www.rok.informatik.hu-berlin.de/Members/Members/salfner/publications/salfner05predicting.pdf, 2005.

[18] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989. DOI: 10.1109/5.18626.

[19] R. Bharathi, R. Selvarani. Software reliability assessment of safety critical system using computational intelligence. *International Journal of Software Science and Computational Intelligence*, vol. 11, no. 3, pp. 1–25, 2019. DOI: 10.4018/ijssci.2019070101.

[20] A. Sundararajan, R. Selvarani. Case study of failure analysis techniques for safety critical systems. In *Proceedings of the Second International Conference on Computer Science, Engineering and Applications*, Springer, New Delhi, India, pp. 367–377, 2012. DOI: 10.1007/978-3-642-30157-5_36.

[21] I. Tumer, C. Smidts. Integrated design-stage failure analysis of software-driven hardware systems. *IEEE Transactions on Computers*, vol. 60, no. 8, pp. 1072–1084, 2011. DOI: 10.1109/TC.2010.245.

[22] A. Avizienis, J. C. Laprie, B. Randell, C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2.

[23] J. K. Horner, J. Symons. Understanding error rates in software engineering: Conceptual, empirical, and experimental approaches. *Philosophy & Technology*, vol. 32, no. 2, pp. 363–378, 2019. DOI: 10.1007/s13347-019-00342-1.

[24] M. Hamill, K. Goseva-Popstojanova. Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system. *Software Quality Journal*, vol. 23, no. 2, pp. 229–265, 2015. DOI: 10.1007/s11219-014-9235-5.

[25] J. A. Duraes, H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006. DOI: 10.1109/TSE.2006.113.

[26] J. Alonso, M. Grottke, A. P. Nikora, K. S. Trivedi. An empirical investigation of fault repairs and mitigations in space mission system software. In *Proceedings of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, IEEE, Budapest, Hungary, pp. 1–8, 2013. DOI: 10.1109/DSN.2013.6575355.

[27] J. Alonso, M. Grottke, A. P. Nikora, K. S. Trivedi. The nature of the times to flight software failure during space missions. In *Proceedings of IEEE 23rd International Symposium on Software Reliability Engineering*, IEEE, Dallas, USA, pp. 331–340, 2012. DOI: 10.1109/ISSRE.2012.32.

[28] F. Zhang, X. S. Zhou, Y. W. Dong, J. W. Chen. Consider of fault propagation in architecture-based software reliability analysis. In *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications*, IEEE, Rabat, Morocco, pp. 783–786, 2009. DOI: 10.1109/AICCSA.2009.5069416.

[29] S. G. Shu, Y. C. Wang, Y. K. Wang. A research of architecture-based reliability with fault propagation for software-intensive systems. In *Proceedings of Annual Reliability and Maintainability Symposium*, IEEE, Tucson, USA, pp. 1–6, 2016. DOI: 10.1109/RAMS.2016.7447984.

[30] V. Cortellessa, V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In *Proceedings of the 10th International Symposium on Component-Based Software Engineering*, Springer, Medford, USA, pp. 140–156, 2007. DOI: 10.1007/978-3-540-73551-9_10.

[31] M. Hiller, A. Jhumka, N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *IEEE Transactions on Computer*, vol. 53, no. 5, pp. 512–530, 2004. DOI: 10.1109/TC.2004.1275294.

[32] A. Jhumka, M. Leeke. The early identification of detector locations in dependable software. In *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*, IEEE, Hiroshima, Japan, pp. 40–49, 2011. DOI: 10.1109/ISSRE.2011.34.

[33] L. Fiondella, S. S. Gokhale. Architecture-based software reliability with error propagation and recovery. In *Proceedings of International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, IEEE, Toronto, Canada, pp. 38–45, 2013.

[34] S. Sinha, N. Kumar Goyal, R. Mall. Early prediction of reliability and availability of combined hardware-software systems based on functional failures. *Journal of Systems Architecture*, vol. 91, pp. 23–38, 2019. DOI: 10.1016/j.sysarc.2018.10.007.

[35] X. W. Wu, C. Li, X. Wang, H. J. Yang. A creative approach to reducing ambiguity in scenario-based software architecture analysis. *International Journal of Automation and Computing*, vol. 16, no. 2, pp. 248–260, 2019. DOI: 10.1007/s11633-017-1102-y.

[36] NASA Software Safety Guidebook, NASA-GB-8719.13, 2004.

[37] R. C. Cheung. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 118–125, 1980.

[38] R. Baldoni, L. Montanari, M. Rizzuto. On-line failure prediction in safety-critical systems. *Future Generation Computer Systems*, vol. 45, pp. 123–132, 2015. DOI: 10.1016/j.future.2014.11.015.

[39] A. A. Markov. An example of statistical investigation of

the text *Eugene onegin* concerning the connection of samples in chains. *Science in Context*, vol. 19, no. 4, pp. 591–600, 2006. DOI: 10.1017/S0269889706001074.

[40] K. Wang, X. X. Long, R. F. Li, L. J. Zhao. A discriminative algorithm for indoor place recognition based on clustering of features and images. *International Journal of Automation and Computing*, vol. 14, no. 4, pp. 407–419, 2017. DOI: 10.1007/s11633-017-1081-z.

[41] S. Honamore, S. K. Rath. A web service reliability prediction using HMM and fuzzy logic models. *Procedia Computer Science*, vol. 93, pp. 886–892, 2016. DOI: 10.1016/j.procs.2016.07.273.

[42] G. I. F. Neyens, D. Zampunieris. Using hidden markov models and rule-based sensor mediation on wearable eHealth devices. In *Procedings of the 11th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, IARIA, Barcelona, Spain, pp. 19–24, 2017.

[43] E. Dorj, C. C. Chen, M. Pecht. A bayesian hidden markov model-based approach for anomaly detection in electronic systems. In *Proceedings of IEEE Aerospace Conference*, IEEE, Big Sky, USA, pp. 1–10, 2013. DOI: 10.1109/AERO.2013.6497204.

[44] S. Ghassempour, F. Girosi, A. Maeder. Clustering multivariate time series using hidden Markov models. *International Journal of Environmental Research and Public Health*, vol. 11, no. 3, pp. 2741–2763, 2014. DOI: 10.3390/ijerph110302741.

[45] A. Simões, J. M. Viegas, J. T. Farinha, I. Fonseca. The state of the art of hidden markov models for predictive maintenance of diesel engines. *Quality and Reliability Engineering International*, vol. 33, no. 8, pp. 2765–2779, 2017. DOI: 10.1002/qre.2130.

[46] J. B. Durand, O. Gaudoin. Software reliability modelling and prediction with hidden Markov chains. *Statistical Modelling*, vol. 5, no. 1, pp. 75–93, 2005.

[47] SimulinkDemo. Modeling an anti-lock braking system - Matlab & Simulink - MathWorks India, [Online], Available: https://in.mathworks.com/help/simulink/slref/modeling-an-anti-lock-braking-system.html?s_tid=srchtitle, January 5, 2019.

[48] R. Bharathi, R. Selvarani. A machine learning approach for quantifying the design error propagation in safety critical software system. *IETE Journal of Research*, to be published. DOI: 10.1080/03772063.2019.1611490.

[49] W. L. Wang, D. Pan, M. H. Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, vol. 79, no. 1, pp. 132–146, 2006. DOI: 10.1016/j.jss.2005.09.004.

[50] S. R. Devi, P. Arulmozhivarman, C. Venkatesh, P. Agarwal. Performance comparison of artificial neural network models for daily rainfall prediction. *International Journal of Automation and Computing*, vol. 13, no. 5, pp. 417–427, 2016. DOI: 10.1007/s11633-016-0986-2.

[51] Y. Z. Jin, H. Zhou, H. J. Yang, S. J. Zhang, J. D. Ge. An approach to locating delayed activities in software processes. *International Journal of Automation and Computing*, vol. 15, no. 1, pp. 115–124, 2018. DOI: 10.1007/s11633-017-1092-9.

[52] R. Roshandel. Calculating architectural reliability via modeling and analysis. In *Proceedings of the 26th International Conference on Software Engineering*, IEEE, Edinburgh, UK, pp. 69–71, 2004. DOI: 10.1109/icse.2004.1317426.

R. Bharathi received the M. E. degree in computer science and engineering from Bharathiar University, India in 2001. She has a progressive teaching experience of 20 years and currently working as a faculty at PES University, Electronic City Campus, India and research scholar at Visveswaraya Technological University, Belagavi, India.

Her research interests include safety critical software systems, machine learning, computational intelligence, and software design quality estimation.

E-mail: sbharathi235@gmail.com, rbharathi@pes.edu (Corresponding author)

ORCID iD: 0000-0001-5412-4381

R. Selvarani received the Ph. D. degree from Jawaharlal Nehru Technological University, India in 2009. She is currently working as a professor having a progressive teaching experience of 28 years and program director for doctoral program at Alliance College of Engineering and Design, Alliance University, India. She has a patent in software architecture and design domain. Her publication in *Information and Software Technology* was selected in terms of having "the best content" in the area of Information Technology for the year 2012 by VERTICAL NEWS, USA. She is carrying out collaborative research with Leeds Metropolitan University, UK and her name is listed in Who's Who for Science and Technology, USA.

Her research interests include machine learning, Internet of things software design quality estimation, service oriented cloud applications, software safety critical systems, and QoS in distributed networks.

E-mail: selvarani.riic@gmail.com, selvarani.r@alliance.edu.in