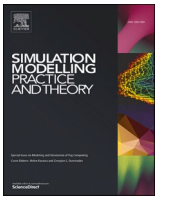
Simulation Modelling Practice and Theory 108 (2021) 102246

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat



Reliability and availability prediction of embedded systems based on environment modeling and simulation

Sourav Sinha [a,*], Neeraj Kumar Goyal [a], Rajib Mall [b]

[a] Subir Chowdhury School of Quality and Reliability, Indian Institute of Technology Kharagpur, India
[b] Department of Computer Science & Engineering, Indian Institute of Technology Kharagpur, India

ARTICLE INFO

Keywords:
Reliability and availability prediction
Hardware-software interactions
Real-time embedded system
Environment model-based testing
Environment modeling

ABSTRACT

The embedded system developers often need to perform software testing even when the target hardware is inaccessible. Due to the inaccessibility of the hardware, the hardware-software interaction testing becomes a challenging task. Researchers have tried to overcome this issue by developing an environment model that simulates the behavior of the actual hardware. In the existing literature, environment modeling has been used for embedded software testing and rarely used for reliability and availability prediction. In this study, we make an attempt to use the environment model for the reliability and availability prediction when actual hardware is inaccessible. At first, we model the hardware environment simulator referring to the System Requirements Specification document. Then, we use the environment simulator for simulating different operational scenarios of the hardware. Based on the operational scenarios, random test cases are generated for testing the embedded software. Finally, we predict the reliability and availability of the system using the test results. Four important aspects covered in this prediction approach are: 1) developing a method for system reliability and availability prediction using environment modeling and simulation technique, 2) considering software-related hardware and hardware-related software interaction failures, 3) applying the proposed reliability and availability model to a case study, and 4) validating the method by comparing the results with an existing approach.

1. Introduction

Embedded systems are often used in day-to-day activities like home appliances, communication, transportation, robotics, defense, space, etc. It is reported that over ninety-eight percent of existing electronic devices are equipped with embedded controllers [1]. Consequently, an embedded system may not only fail due to hardware-specific or software-specific failures; hardware-software interaction failures may also cause system malfunctioning. For example, Iyer and Velardi [2] at Stanford University demonstrated the impact of hardware-related software interaction failures on performance of an Multiple Virtual Storage/ System Extensions (MVS/SP) operating system. They found that 35 percent of the software malfunctioning cases are attributable to hardware failures. On the other hand, bugs in the software may also lead a hardware device to failure [3]. For example, an investigation report suggested that a software glitch in Boeing 737 Max flight led to the infamous accident of October 2018 that took 189 people's life onboard. It is now

* Corresponding author at: Subir Chowdhury School of Quality and Reliability, Indian Institute of Technology Kharagpur, Kharagpur, 721302, India.
E-mail address: sourav.sinha@iitkgp.ac.in (S. Sinha).

https://doi.org/10.1016/j.simpat.2020.102246
Received 14 June 2020; Received in revised form 10 December 2020; Accepted 13 December 2020
Available online 18 December 2020
1569-190X/

getting widely accepted that some hardware-software (HW-SW) faults are inter-dependent [4,5]. Therefore, system reliability and availability evaluation approaches need to consider HW-SW interactions testing along with hardware-specific and software-specific failure. However, in the early development phases of real-time embedded software, team members may not have access to the actual hardware components.

Some previous studies have tried to address the above issue by applying Software-in-the-Loop Simulation (SILS) approach for HW-SW interactions testing [6,7]. In this approach, the software unit is executed in a simulated hardware environment. Later, Iqbal et al. [1] extended this technique for Real-Time Embedded Systems (RTES) testing. They used UML-MARTE profile for the modeling hardware environment. They also demonstrated use of SILS approach using some industrial applications. However, this line of work has been used only for testing of real-time embedded software and has hardly been explored for reliability analysis.

As reported in our previous studies, two types of interaction failures play a significant role in system reliability analysis [8,9]. These are hardware-related software and software-related hardware interaction failures. Some researchers have considered only hardware-related software interaction failures in developing their reliability and availability models [5,10–12]. They have ignored the possibility of software-related hardware interaction failures [5,10–12]. Another group of researchers has considered both, hardware-related software and software-related hardware interaction failures [13–16]. However, they targeted only - qualitative reliability analysis. We are interested in developing quantitative reliability and availability prediction method using the concept of environment modeling and simulation. The scope of our work is limited to the reliability and availability analysis of real-time embedded systems when actual hardware components are inaccessible.

The proposed method performs testing of an executable embedded software in a simulated hardware environment and predicts reliability and availability based on the test results. At first, we develop the domain model of the system using UML Class diagram (with MARTE extensions for modeling real-time scenarios). Each system component (hardware and software) is represented by a Class in the Class diagram. The system components and their dependencies are identified by referring to the System Requirements and Specifications (SysRS) document. We also identify the hardware components' operation mode/ states by referring to a standard reliability

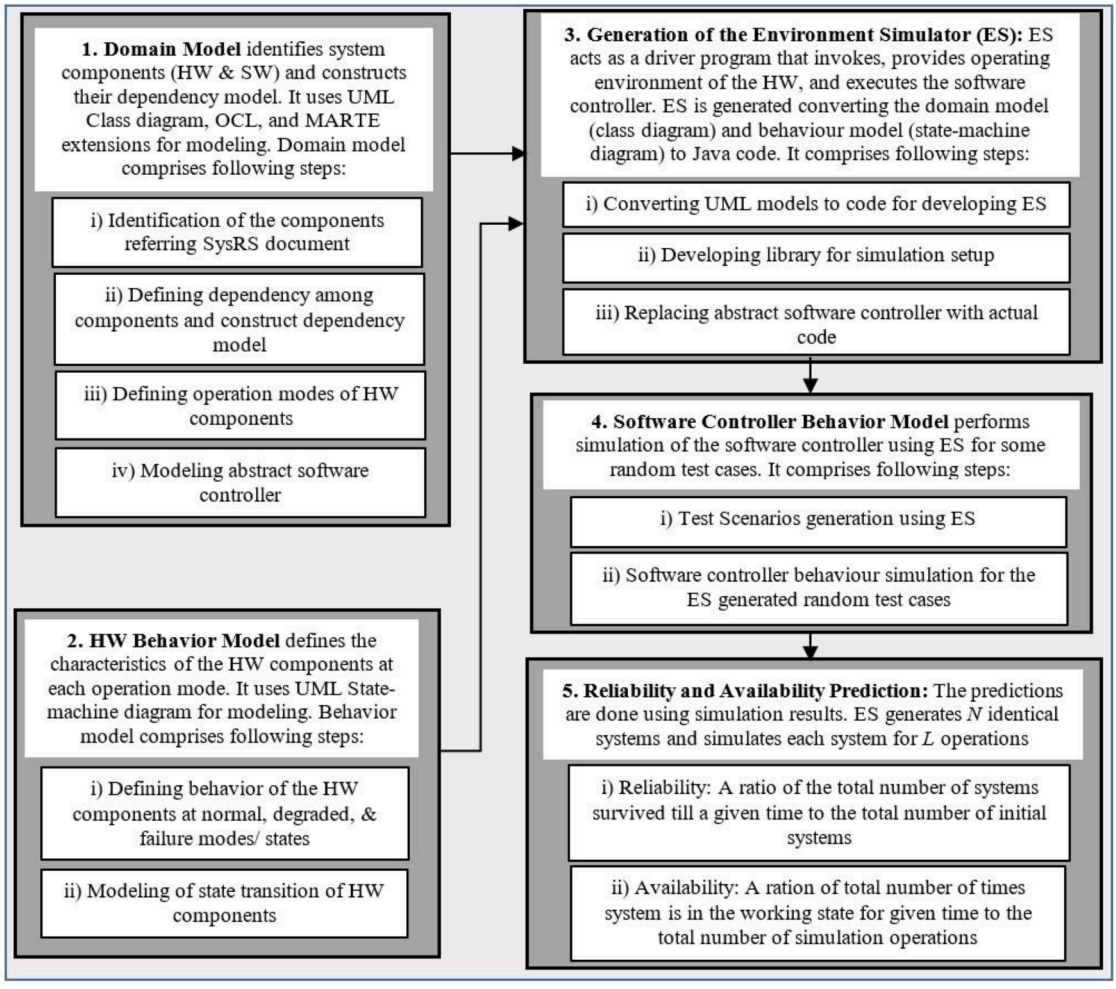


Fig. 1. Diagrammatic representation of the proposed method.

handbook or reliability data-sources. In this phase, the software controller is considered as an abstract component. We replace it with actual code at the time of code generation. Subsequently, we develop the hardware component behavior model using the UML State-machine diagram. For each hardware component, a state-machine diagram is constructed. The behavior model defines a component's characteristics at each state (normal, degraded, and failure). It also models the state transition of the hardware component for different operation conditions. Then, we generate an environment simulator by converting the domain model and hardware behavior models. The environment simulator acts as a driver program that invokes, provides hardware operational scenarios, and executes the software controller during testing. In this regard, we have adopted the work presented by Iqbal et al. [1] to perform software-in-the-loop simulation (SILS). SILS approach performs testing of the software controller for some simulated test cases. Finally, reliability and availability are predicted based on the test results. Reliability is evaluated as the ratio of the total number of systems that have survived till a given time to the total number of initial systems. On the other hand, availability is evaluated as the ratio of the total number of times system is in the working state for a given time to the total number of simulation operations. Fig. 1 provides a schematic representation of our method.

The proposed method predicts the reliability and availability of the RTES, before the embedded software is integrated with the target hardware. It performs testing of an embedded software using hardware environment modeling and predicts reliability and availability based on the test results. The existing literature has reported early testing of embedded system using environment modeling [1,17–22]. However, they have not shown any interest in predicting the reliability and availability of the system. We have extended the environment modeling based testing approach for reliability and availability prediction. Our method gives a quantitative reliability and availability prediction considering all possible types of interaction failures (hardware-related software and software-related hardware), along with hardware-specific, software-specific failures. Most of the existing reliability and availability prediction approaches have not considered all possible types of hardware-software interaction failures [5,10–12] or have not targeted quantitative prediction [13–16]. The proposed method can also reduce the production cost significantly. For example, if the RTES does not meet the target reliability at the time of the post-integration testing due to fault in system components, it incurs considerable costs. In this paper, we have applied the proposed method to an Aircraft Fuel Control System as a case study for demonstrating the transient reliability and steady-state availability evaluation process. The obtained results (reliability and availability) for the case study are compared with an existing method [23] for the validation of the proposed method.

1.1. Practical usage of the proposed method

Previous studies have reported early testing of safety-critical embedded software when the hardware is not accessible [1,17–22]. The hardware often becomes inaccessible because different teams parallelly develop hardware and software parts. Their physical location also may not be the same [1]. Researchers have overcome this issue by developing an environment model that simulates behavior of the actual hardware [1,17–22]. They have introduced the Software-in-the-Loop Simulation (SILS) approach, where environment modeling of the hardware is used to test the embedded software [1,17]. Iqbal et al. [1] has explained their experience of working with two organizations (WesternGeco AS, Norway and Tomra AS, Norway), where they have performed early testing of the embedded software using SILS due to the inaccessibility of actual hardware. They developed embedded software for a seismic acquisition system while working with WesternGeco. Later, another embedded software was developed by them for automated recycle machines while working with Tomra. In both cases, they tested the embedded software using SILS, as the organizations made it mandatory. The organizations instructed them to test the embedded software in a simulated environment before deploying it on the actual hardware. Jeong [18] also performed early testing of safety-critical AUTOSAR (AUTomotive Open System ARchitecture) software used for the Electronic Control Unit (ECU) in the automotive industry. In this approach, they relied on the SILS approach for AUTOSAR software testing using environment modeling. They have mentioned that early testing reduces the risk of severe damage to the system-to-be-controlled by the embedded controller. Early testing may also reduce the cost of iterative development if the software's residual errors are identified earlier. Other published work [17,19–22] also used SILS based embedded software testing for the similar reasons as mentioned above. However, they have used SILS approach for embedded software testing only and rarely used it for reliability and availability prediction. We have extended this line of work to predict the reliability and availability of the embedded system. In the proposed method, we have performed SILS based testing of the embedded software with environment modeling for the hardware. Subsequently, reliability and availability of the system is predicted using the test results.

1.2. Some basic concepts

We have explained some basic concepts that are the foundation of the proposed reliability and availability prediction method. These concepts are frequently used in the entire paper. The explanation we have given are general enough to cover the entire range of embedded systems.

Environment modeling. A newly developed system is often tested in the development platform. Developers use the environment modeling approach to provide a virtual operational environment to the system for execution. The environment modeling approach creates a simulator that invokes the system and provides different aspects of the operational environment during testing. In this paper, we have used environment modeling of the hardware to test the software controller in the simulated hardware environment when actual hardware is inaccessible. The hardware environment model generates an Environment Simulator (ES) program that performs four tasks without any human intervention: i) generates random test cases to execute the software controller, ii) invoke the software controller, iii) provides an operational environment of the hardware to execute the software controller, and iv) record the response of the software controller for the test cases.

Embedded software testing using software-in-the-loop simulation (SILS). During early testing of the embedded software SILS approach is often used. In this approach, the tester creates a driver module and a stub module to perform SILS. The driver module invokes the embedded software and provides an operational environment of the Electronic Controller Unit (ECU). The stub module provides operational environment of the system-to-be-controlled by the embedded software and generates feedback signals for the driver module. The embedded software remains connected with the driver module and stub module in a loop during testing. The SILS is commonly used for the validation of the embedded software.

Reliability and availability prediction using environment modeling. In this approach, ES generates multiple instances (say N) of identical independent systems. ES simulates random input data set during each simulation iteration to execute the software controller and record the response. We consider each simulation iteration as an operation cycle of the system. The operation cycle should be large enough (say L) to cover the entire input domain. During the simulation process, the operational state of the software controller may changes. If the software controller's response complies with the specified normal range of the SysRS document, it is considered as a working state. If it exceeds the normal range, it is considered as a failure state. Based on the outcomes of the simulation process, reliability and availability are predicted. Reliability is defined as the ratio of the total number of systems survived (working state) till a given time to the total number of initial systems (N). During reliability prediction, we consider recovery from a fault-tolerant state is possible. However, recovery from the absorbing state is not possible. During availability prediction, we also consider all instances (N) of the system are executed for L operation cycles. Availability is defined as the ratio of the total number of times the system is in the working state for a given time to the total number of operations (L). Here, we consider recovery is possible from the fault-tolerant states as well as the absorbing states.

This paper is organized as follows: Section 2 reports a review of the existing work. Section 3 proposes environment modeling and based on it, develops simulation-based system reliability and availability prediction method. Section 4 demonstrates the application of the proposed method to a case study. Section 5 validates the proposed method. Finally, Section 6 concludes this paper.

2. Literature review

We discuss the related literature in two sections: i) environment modeling and environment model-based testing of RTES, and ii) reliability and availability prediction of embedded systems duly considering hardware-software interactions.

2.1. Environment modeling and environment model-based testing of RTES

Literature in this area can be further categorized into two groups: a) black-box testing approaches and b) white-box testing approaches.

We discuss some of the major black-box testing approaches in this section [1,17,24–31]. Auguston et al. [24] demonstrated environment modeling and testing of RTES using Attributed Event Grammar. At first, they referred to the SysRS document to identify some potential faulty scenarios of the system. Then, they tested the interaction between the executable software program and its hardware environment for the above-identified scenarios. To perform the simulation operation, they used a digital simulator that generates random test cases. Some other studies [27,29,30] have developed RTES test profiles referring to the SysRS and environment assumptions. They used a timed automata tool UPPAAL for defining the system specifications and auto-generating test cases. Then, they tested the RTES functionality for the auto-generated test cases. Later, the UPAAL based testing approach also adopted by Larsen et al. [30] for industrial applications. Krichen and Tripakis [28] used a non-deterministic timed automata approach for conformance testing of RTES. It helps the user to define the modeling assumptions of the RTES environment and also act as an interface between the tester and the executable software program. Lindlar et al. [31] developed an RTES testing model using a digital simulator for large scale industrial set-up. In this model, the testing team provides annotated finite state machines (FSMs) that define the boundaries for the test cases. They evaluated the outputs of the test cases using a fitness function. Based on the output of the fitness function, they determined the success and failure of the test cases. Iqbal et al. [1,17] proposed environment model-based testing approach for the RTES. They have used UML/ MARTE profile to model the environment of the real-time embedded system. This model address the practical issues faced by the industries during real-time testing like synchronization of the events, time span of event execution, etc. They also used the Object Constraint Language (OCL) expression for defining the system constrains. Recently, Deng and Gao [26] proposed a testing approach for safety-critical software. Their testing approach followed the hardware-in-the-loop simulation model in a hierarchical distributed architecture. Such architecture decomposes the testing task into small units, so that each system functionality can be tested efficiently to uncover the faults. Silano and Iannelli [32] proposed a tool (CrazyS) for SILS testing of Crazyflie 2.0 nano-quadcopter. This tool is an extension of robot operating system RotorS that generates virtual environment for SILS testing. They demonstrated the application of the tool using a case study of the quadcopter's flight controller. Here, they shown the way tool performs SILS and its dynamical model exchange information with the flight control system of the quadcopter.

We review the white-box testing approaches that use environment model-based testing of RTES [33–39]. Du Bousquet et al. [35] developed a framework to automate the RTES testing based on its environment. This framework kept provision for applying some operational conditions during testing like environmental constraints, system behavior, etc. They used time-related (temporal) logic for defining these operating conditions. Hatebur et al. [36] combined the use of a requirements model and environment model for RTES testing. The requirements model was developed using SysRS document and the environment model was defined using UML state machines. They demonstrated two types of testing strategies. The first one was on-the-fly testing strategy where test cases were prepared and executed immediately. The other one was batch testing strategy where test cases were executed at the time of regression testing. Adjir et al. [33] proposed a testing approach for real-time systems using Prioritized Timed Petri Nets (PTPN). The PTPN was

used for simulating the operating environment and auto-generation of test cases. Two concurrent sub-nets were constructed for defining the operating environment and system responses. Later, a similar approach was adopted by Peleska et al. [37,38] for auto-generation of test cases. They developed a tool named RT-Tester to provide the operating environment to the SUT and also to auto-generate the test-cases. A major advantage of their work was that RT-Tester supported various platforms. Yu et al. [39] developed a regression testing tool SIMRT that detected the faults induced during imperfect debugging. They provided a test case selection technique considering the characteristic of the program and its potentiality to generate faults. It also minimizes the number of test cases required to uncover the faults. In this regard, they suggested prioritizing test cases based on their potential to uncover faults. Later, Binh et al. [34] proposed a new regression testing tool called LusRegTes for real-time testing of safety-critical applications. This tool was only applicable to the Lustre programs. For each Lustre program, a set of execution paths and execution factors were identified. The test cases were generated by comparing paths between different versions of the same program. Finally, the programs were executed to uncover the faults.

2.2. Reliability and availability prediction considering hardware-software interactions

The existing reliability/ availability/ failure analysis literature can be divided into two categories based on the hardware-software interactions: a) models considered only hardware-related software interaction failures, and b) models considered both hardware-related software and software-related hardware interactions failures.

The models that considered hardware-related software interaction failure have mostly used Markovian approach [5,10–13]. For example, Teng et al. [5] modeled the HW-SW interaction failure using Markov chain. They assumed that a system fails whenever the software fails, the hardware fails, or HW-SW interaction fails. The reliability of the system is evaluated as a product of independent hardware reliability, independent software reliability, and HW-SW interaction reliability. They considered hardware reliability follows the Weibull model whereas software reliability follows NHPP model. They also considered that HW-SW interaction reliability follows Markov chain model. Later, Roy et al. [11] applied this approach to predict reliability of the Phasor Measurement Units (PMU). Sumita and Masuda [12] modeled the system reliability and availability as a multivariate stochastic process. They used semi-Markovian process and matrix Laguerre transformation for modeling the system failure. This model considered failure of hardware/ software is independent of each other. Hardware component failures were modeled using an alternating renewal process. The downtime of the renewal process follows general statistical distribution, whereas the uptime follows an exponential distribution. They illustrated that if hardware-related failures lead to software failure, two different situations may arise. First, completion of a software repair without any interruption from hardware failures. Second, hardware failure may interrupt software repair operations. They modeled both the situations using stochastic processes and computed system reliability using matrix Laguerre transform. Costes et al. [10] modeled reliability and availability of a repairable system using Markov chain approach. They assumed that the software debugging process might introduce new errors and that the number of remaining errors is unknown. They also considered that software failures follow exponential distribution whereas hardware failures follow Poisson distribution. The software failure process is modeled using Trivedi and Shooman model [40] and hardware failure process is modeled using Poisson distribution. Finally, the reliability and availability models were developed by combining hardware and software failure process models. This model is applicable to the stand-by systems as they have considered multiple unit of the same system to achieve high availability. Zeng et al. [41] propose a non-repairable system reliability prediction model using an Integral-based approach. This model identified the system's state transition paths and developed an integral expression for each path. Then, they solved the integral expressions to obtain the probability of each state. The system reliability was predicted by adding the probabilities of the working states. This model assumed the system fails only due to hardware failure (partial and complete) and hardware-related software failure. The software failures were considered as an impact of partial failure of the hardware. The possibility of software-specific failure or software-related hardware failure was not considered in this model. They verified the correctness of the model using a Markovian approach.

A group of researchers has considered both, software-related hardware and hardware-related software interaction failures. They used the Functional Failure Identification and Propagation (FFIP) framework or its variant for reliability/ availability/ failure analysis of the system. Jensen et al. [42] introduced FFIP framework in the domain of qualitative reliability analysis. They used functional model to configure the dependency diagram of the system. They analyzed signal/ material flow pattern throughout the functional model. The objective of the analysis was to isolate critical nodes of the system. Then they altered the flow path to restrict signal/ material flow through the critical nodes. It helped them to avoid fault propagation in the system. Tumer and Smidts [43] also adopted FFIP for system failure analysis. They developed a functional layout using conceptual components of the system. They identified all the failure modes of components. Subsequently, they identified some key components (called monitoring nodes) having high potential to fail. Sierla et al. [15] proposed another fault propagation model using extended FFIP. Unlike others, they took into consideration signal/ material flow across the boundaries of the mechatronic systems. Furthermore, they incorporated concurrent integration of risk assessment at the system design phase. They also demonstrated the application of the model using Simulink/ Stateflow environment. However, the FFIP based failure analysis is not flawless. Papakonstantinou et al. [44] reported the following a) FFIP failed to give consistent results when a different modeling technique is used for the same system, b) it also does not suggest the best design alternative, if multiple options are available for the same system. Papakonstantinou et al. [44] suggested some modifications in FFIP to overcome such drawbacks. However, this model is quite similar to its predecessor [43]. Later, Papakonstantinou et al. [45] tried to improve the drawbacks of the earlier version. They introduced Hierarchical Functional Fault Detection and Identification (HFFDI) framework that combined traditional FFIP and machine learning techniques. Diao et al. [46] proposed a failure analysis model using Integrated System Failure Analysis (ISFA) [47]. ISFA is an extension of the traditional FFIP framework. It helped to detect faults in cross-domain functionalities. A new feature was added in this framework, which was named as online monitoring (OLM) system and

helped to detect and analyze potential faults. Irshad et al. [48] proposed a qualitative reliability prediction approach, which is a combination of the Human Error and Functional Failure Reasoning (HEFFR) approach and the Depth First Search (DFS) method. HEFFR, an extension of FFIP, evaluates potential functional failures, human errors, and their propagation at early design phases. DFS considered each branch's level as a time-step and each branch as an input scenario for HEFFR. This approach considered all types of hardware-specific and software-specific failures, along with their interactions failure. The authors also claimed their approach uncovered hard-to-detect failure scenarios that arise due to human error.

3. Proposed reliability and availability prediction method

In our proposed method, we perform testing of a software controller in a simulated hardware environment and predict system reliability and availability based on test results. The applicability of the proposed method is limited to real-time embedded systems. At first, we specify the assumptions that have been considered to develop the method. Then, we propose the method for reliability and availability prediction of the embedded system. Finally, we apply the method to an aircraft fuel control system as a case study.

3.1. Assumptions of the proposed method

1) The testing team does not have access to the actual hardware for real-time testing of the HW-SW interactions. The hardware often becomes inaccessible because two different teams develop hardware and software in parallel. Their physical location also may be different.
2) The testing team has access to the SysRS document. The SysRS is used to identify the hardware components and the dependency among the components. It becomes accessible in the early phases of the development.
3) The testing process considers the software controller as a black-box. As the proposed method follows SILS approach, black-box testing is performed considering the entire software a single unit.
4) All operation modes of the hardware are known. In this regard, some standard reliability handbook or failure data-source is referred.
5) The probability distribution of the occurrence of hardware operation modes is known. Here also some standard reliability handbook or failure data-source is referred.
6) Operation modes of a hardware component are divided into three categories, as observed in the literature [5]:
   - Normal: Component exhibits its work as per specification
   - Degraded/ partial failure: Component performs its work in a degraded manner. It is also two types: a) Additive polarity: It enhances performance/ signal strength of the component, b) Subtractive polarity: It diminishes the performance/ signal strength of the component
   - Complete failure: It disrupts the overall functionality of the component. This type of failure may be observed as transient or permanent in nature.
7) The functional behavior of the software controller is described in the SysRS document. The response of the software determines the operation modes of the embedded system. The operation modes of the embedded system are broadly divided into two categories, as observed in the literature [49]:
   - Working: The output of the software controller conforms to the SysRS document. It can work in two states: a) Normal working state: Software exhibits its designated work adequately without any hardware component failure, and b) Fault-tolerant state: Software exhibits its designated work adequately even when some hardware component has failed [50,49].
   - Failure: The output of the software controller does not conform to the SysRS document.
8) Operation modes of the hardware-software interaction are divided into two categories, as observed in the literature [5,8,16]:
   - Normal: The successful exchange of data/ signal/ material among hardware-software components
   - Failure: The unsuccessful exchange of data/ signal among hardware-software components. Such failures are two types: a) Hardware-related software interaction failure: The transient hardware failure (like indeterminate memory and delay) leads this type of failure, and b) Software-related hardware interaction failure: At some operating conditions, the software response may be undefined or incorrect leading to hardware failure.

We propose the reliability and availability prediction method based on the above assumptions. The major steps of the proposed method are discussed in the following.

3.2. Domain model

The domain model identifies the major components of the system, their properties and dependencies. We refer to the SysRS document to collect the information related to domain model. Based on the information available in the SysRS document, we develop the domain model. We use UML Class diagram to represent the domain model. Each Class of the Class diagram represent a system component. We also use some MARTE extension to support real-time application-specific customization and OCL expression for constraints modeling. The major steps involved in developing the domain model are the following:

Identification of the components: We consider that a system consists of n number of hardware components and one software (controller) component. Out of n number of hardware components, $(n-1)$ are the input peripherals and one is the microcontroller. The i^{th} hardware component is represented as C_i where, $i \in \{1, n\}$ and the software component is represented as S. During construction of

the Class diagram, each hardware and software component is represented as a Class.

Defining dependency among components: We define all the physical and logical dependencies among the environment components that are required for the flow of signal/ data/ material through the system. The association between two components is represented as $\alpha \xrightarrow{x \ldots y} \beta$. Here, α and β are the system components. The notation x and y represent the cardinality number of the components (Classes). So, we express the above association as x number of α components are associated with y number of β components.

Operation modes of a hardware component: Each hardware component C_i has m_i number of operation modes (working and failure). A j^{th} operation mode of component C_i is represented as M_{ij} where, $i \in \{1, n\}$ and $j \in \{1, m_i\}$. Each state depicts a distinguished behavior of the component. The current operation mode of a component (Class) is included as an attribute of the Class.

Modeling abstract software controller: In this phase, the software controller is considered as an abstract component. It is represented as an Abstract Class of the Class diagram. During system simulation, we replace the abstract class with the code of the software controller.

3.3. Hardware component behavior model

The behavior model defines the characteristics of a component at each operation mode. Each component has a distinct behavior model. We use UML State-machine diagram to construct the behavior model of the hardware components. Each state of the UML State-machine represents an operation mode of the component. The behavior model of a component is described as follows:

Behavior of the Component State: During execution, a hardware component will be in one of its operation mode at any time, say M_{ij}. For any input x, if the response signal of a component C_i at operation state M_{ij} is denoted as v_i, then v_i is expressed as

$$v_i = M_{ij}(x)$$

We categorize the response of the components, v_i, into three groups:

Normal (F_{Normal}): If the response signal v_i of a component C_i at operation state M_{ij} does not deviate from its normal operation range $C_i(Normal)$, as specified in the SysRS document, then the component is considered to be in the normal state. It is expressed as following:

$$v_i = C_i\_Normal$$

Partial Failure/ Degraded ($F_{Degraded}$): Sometimes, component response deviates from its normal operation range as specified in the SysRS document. The reason behind such deviation may be moderate degradation of the materials, noise, oscillation, reverse polarity, etc. This is considered as partial failure of the component. The partial failure can be additive polarity if the response signal v_i of a component C_i at operation state M_{ij} is higher than its normal operation range $C_i(Normal)$. Another type of partial failure can be subtractive polarity, if the response signal v_i of a component C_i at operation state M_{ij} is lower than its normal operation range $C_i(Normal)$. It is expressed as follows:

$$v_i > C_i\_Normal$$

or

$$v_i < C_i\_Normal$$

Failure ($F_{Failure}$): If a component stops responding, it is considered to be in the failure state. The failure state can be of two types: permanent and transient. Sometimes due to open circuit failure, short circuit failure, degradation of materials, complete breakdown of component, intermittent signal, etc. a component stop responding completely, in this case, it is said to have permanent failure ($F_{Permanent}$). A component may undergo temporary failures due to delay in signal, indeterminate bit values, improper synchronization of the signals, etc. Such failures of a component is considered as a transient failure ($F_{Transient}$). In both cases, component failure can be expressed as the following:

$$v_i \equiv C_i\_No\_response$$

Modeling of state transition: We assume that all the components are initially at the normal working state. During execution, a component state may change based on its probability of failure. A mapping function $\mathbb{f}$ maps current operational state M_{ia} of component C_i to a new operational state M_{ib} with a probability of occurrence $p_{i_{ab}}$ where $a, b \in \{1, m_i\}$, $i \in \{1, n\}$, and $\sum p_{i_{ab}} = 1$.

$$M_{ia} \xrightarrow[\mathbb{f}(M_{ia}, p_{i_{ab}})]{} M_{ib}$$

We assume that initially, all the system components are in normal working state. Therefore, the state M_{ia} represents the normal working state of the component C_i. If a transition occurs, then the component C_i transits from normal state to a partial/ complete failure state. So, the state M_{ib} represents any partial/ complete failure state of the component C_i.

3.4. Generation of the environment simulator

The environment simulator (ES) acts as a driver program that invokes, provides operational environment, and executes the

software controller during testing. The ES is developed by converting the domain model and behavior models to a software package. The model to code conversion process makes use of the state design pattern [51]. It helps to convert the UML artifacts to code following the best practice of Object-oriented software development.

Generation of ES and its helper library: We use Simulator Generator (SG) framework that performs the conversion model-to-code conversion process. The SG takes UML artifacts as input and generates the ES. A library package is also developed that helps the ES to fix the sequence of operations, setting their priorities, etc. The major components of the SG are a) set of drivers convert UML artifacts to code, b) a set of libraries that convert the *MOFscript* to code, c) another set of libraries that translate the *OCL expression* to code, and d) a set of classes defines the environment configuration for test setup. The detailed model-to-code conversion process to be discussed in the case study section (Section 4).

Replacing abstract software controller with code: We replace the abstract software controller with the actual Java code. The software controller is treated as a black-box to perform the software-in-the-loop simulation-based testing.

3.5. Software controller behavior model

In this phase, we model the state transitions of the software controller due to different operational scenarios generated by the environment simulator (ES). The ES considers that an embedded system consists of $(n-1)$ input peripherals and a microcontroller. During each simulation iteration, the ES randomly selects the operation mode of each input peripheral using Monte Carlo simulation. Let us consider that during the simulation process, the j^{th} operation mode of the peripheral C_i occurs where $i \in \{1, n-1\}$ and $j \in \{1, m_i\}$. Then, the operation state of the peripheral C_i is denoted as M_{ij}. At operation state M_{ij}, the response of the input peripheral C_i for any initialization parameter x is denoted as $v_i = M_{ij}(x)$. Let us consider, δ is the random dataset generated by the $(n-1)$ input peripherals. Therefore, the dataset is denoted as $\delta = \cap_{i=1}^{n-1} v_i$. The input dataset is fed to the microcontroller. The microcontroller also has a number of operation modes. Therefore, the ES also sample the operation modes of the microcontroller (C_n). Let us consider, during sampling process k^{th} operation mode of microcontroller C_n has occurred. Then, the operation state of the microcontroller is denoted as M_{nk}. The response of the microcontroller for input dataset δ is denoted as $M_{nk}(\delta)$. Finally, the ES feeds the response of the microcontroller to the software controller as test case t ($t = M_{nk}(\delta)$). During each simulation iteration, the state transition of the software controller is observed for the test case t.

ES assumes that all the system components remain in the normal state at the start of the simulation process. Therefore, the system is in the normal state, initially. The system state transition occurs due to three reasons: a) failure ($F_{Permanent}$) or degradation ($F_{Degradation}$) of the input peripherals, b) failure ($F_{Permanent}$ or $F_{Transient}$) or degradation ($F_{Degradation}$) of the microcontroller, or c) combined failure or degradation of microcontroller and peripherals. We have noticed three types of state transition of the system: a) normal to fault-tolerant ($S_N \rightarrow S_T$), b) normal to failure ($S_N \rightarrow S_F$), and c) fault-tolerant to failure ($S_T \rightarrow S_F$). We can use any digital simulation platform to test the system using ES generated test cases and observe the state transition. The ES generated test cases that cause system state transition, are classified into the following types of test scenarios:

Test Scenarios 1 (TS1): Let us consider that the ES simulates a scenario in which the microcontroller is in the normal working state and l out of $(n-1)$ peripherals fail. In this scenario, if the ES generated test case t causes system state transition from normal working state to a fault-tolerant state, then t is considered as an element of set $TS1$. Here, the set $TS1$ represents collection of all test cases that trigger state transition from normal to fault-tolerant state. So, any test cases t of the $TS1$ is represented as:

$$t \in TS1$$

where $t = M_{nk}(\delta)$ where $\delta' = \cap_{i=1}^{l} v_i$, $\delta' \subseteq \delta$, $v_i \in F_{Permanent}$, $l \in \{1, n-1\}$, $k \in \{1, m\}$ and $l < n$

Test Scenarios 2 (TS2): Let us consider, the ES simulate a scenario where the microcontroller is in the normal working state and p out of $(n-1)$ peripherals fail. In this scenario, if the ES generated test cases t causes system state transition from normal working state to complete failure state, then t is considered as an element of set $TC2$. Here, the set $TC2$ represents a collection of all sets of test cases that trigger state transition from normal to fault-tolerant state. So, any test cases t of the $TC2$ is represented as:

$$t \in TS2$$

where $t = M_{nk}(\delta)$ where $\delta' = \cap_{i=1}^{p} v_i$, $\delta' \subseteq \delta$, $v_i \in F_{Permanent}$, $p \in \{1, n-1\}$, $k \in \{1, m\}$ and $p \leq n$

Test Scenarios 3 (TS3): Let us consider the ES simulates a scenario in which the microcontroller is in the complete failure state and all/ some peripherals are in normal working state. In this scenario, if the ES generated a test case t and it causes system state to transit from normal working state to complete failure state, then t is considered as an element of set $TS3$. Here, the set $TS3$ represents a collection of all sets of test cases that trigger state transition from normal to fault-tolerant state. So, any test cases t of the $TS3$ is represented as:

$$t \in TS3$$

where $t = M_{nk}(\delta) = \varnothing$ where $k \in \{1, m\}$

Test Scenarios 4 (TS4): The software controller may fail to restrict the system response within its operational limits due to exceptional input conditions. Such exceptional input conditions may occur due to logical errors, out of range inputs, error in input data filtering, etc. Due to such exceptional conditions, the control signal generated by the software controller causes malfunctioning of the associated hardware components. This is considered as software-related hardware interaction failure. We refer SysRS to identify

potential exceptional input signals invoked by the set of degraded components. Let us consider that the ES simulates a scenario where the microcontroller is in normal working state and r input peripherals out of $(n-1)$ generate exceptional input condition to the software controller due to degraded mode. In this scenario, if the ES generated test case t that causes system state to transit from normal/ fault-tolerant state to complete failure state, then t is considered as an element of set $TS4$. Here, the set $TS4$ represents collection of all sets of test cases that trigger state transition from normal/ fault-tolerant state to failure state. So, any test cases t of the $TS4$ is represented as:

$$t \in TS4$$

where $t = M_{nk}(\delta)$ where $\delta^{'} = \cap_{i=1}^{k} v_i,\ v_i \in F_{Degraded}$, $\delta^{'} \subseteq \delta$ and $k \in \{1, m\}$

Test Scenarios 5 (TS5): The transient behavior of the components like indeterminate bit value, delay in signal/ abnormal pulse, improper synchronization, abnormal sequence of signal, etc. also lead to software failure. It is considered as hardware-related software interaction failure. Let us consider that ES simulates a scenario in which the microcontroller is in the transient failure state and all/ some peripherals are in normal working state. In this scenario, if the ES generated test case t causes system state transition from normal working/ fault-tolerant state to complete failure state, then t is considered as an element of set $TC5$. Here, the set $TC5$ represents collection of all sets of test cases that trigger state transition from normal/ fault-tolerant state to failure state. So, any test cases t of the $TC5$ is represented as follows:

$$t \in TS5$$

where $t = M_{nk}(\delta) = \varnothing$ where $M_{nk} \in F_{Transient}$, and $k \in \{1, m\}$

Software controller behavior simulation: The test scenarios represent either a hardware-specific failure, a software-specific failure, or a hardware-software interaction failure. During the software controller behavior simulation, we group the above-identified test scenarios based on their impact on software controller state transition.

a) The software controller state transition occurs from *normal working state* to the *fault-tolerant state* for any test case t ($t \in TS1$) if comprises some component failure. The software controller state transition is expressed using the function G_{NT} as below:

$$G_{NT}: \ S_N \rightarrow_{t \in TS1} S_T$$

b) The software controller state transition occurs from *normal working state* to *complete failure state* for any test case t if a) t comprises any key component failure ($t \in \{TS2$), or b) t creates any exceptional input condition ($t \in TS3$), or c) t comprises transient failure of microcontroller ($t \in TS5$). The software controller state transition is expressed using the function G_{NF} as shown below:

$$G_{NF}: S_N \rightarrow_{t \in TS2 \ || \ t \in TS3 \ || t \in TS4 \ || t \in TS5} S_F$$

c) The software controller state transition occurs from *normal working state* to *complete failure state* for test case t if a) t comprises any key component failure ($t \in \{TS2$), or b) t creates any exceptional input condition ($t \in TS3$), or c) t comprises transient failure of microcontroller ($t \in TS5$). The software controller state transition is expressed using the function G_{TF} as below:

$$G_{TF}: S_T \rightarrow_{t \in TS2 \ || \ t \in TS3 \ || t \in TS4 \ || t \in TS5} S_F$$

3.6. Reliability and availability prediction

The reliability is defined as a ratio of the total number of systems survived ($\aleph_s$) till time interval ζ to the total number of initial systems ($\aleph_0$). During the simulation process, we assume that $\aleph_0$ identical systems are generated and each system is executed for L operations. We consider each simulation iteration as an operation cycle of the system. If each operation cycle takes a unit time, then L operations takes L unit time. We divide the total operation time L into i equal intervals and record the frequency of systems' failure (due to system state transition from normal to failure G_{NF} and state transition from fault-tolerant to failure G_{TF}) in each interval. Let us consider, at ζ^{th} interval ($\zeta \in \{1, i\}$) total number of systems' failure is $\aleph_f$. Therefore, the number of systems' survival at ζ^{th} interval is $\aleph_s$ ($\aleph_s = \aleph_0 - \aleph_f$). We plot the ratio $\aleph_s/\aleph_0$ against the time intervals i and get the transient reliability graph. So. the reliability of the system for time interval ζ is expressed as

$$R_{sys}(\zeta) = \aleph_s / \aleph_0$$

where $\aleph_s$ is total number of systems survived till time interval ζ and $\aleph_0$ is total number of systems.

The availability of a system is defined as a ratio of the total number of times system is in working state till time ζ to the total number of operations (L_ζ). If we consider more than one identical system is working at the same time then, availability is defined as a ratio of the average number of times the systems are in the working state till time ζ to the total number of operations (L_ζ). During availability evaluation, we consider recovery from the failure sate is possible. Therefore, the system state transition from failure to fault-tolerant

($S_F \rightarrow S_T$) and fault-tolerant to normal ($S_T \rightarrow S_N$) are taken into consideration. The time taken to repair a failure state is considered as unit time. Let us consider, for the i^{th} system the estimated number of times transitions G_{NT} (normal to fault-tolerant), G_{NF} (normal to failure), and G_{TF} (fault-tolerant to failure) occurs are x_i, y_i, and z_i, respectively. So, the total number of times i^{th} system response undergoes functional failure is estimated as $(y_i + z_i)$ and total number of times system is in working state is W_i ($W_i = L - y_i + z_i$). If the total number of simulation operations performed till time ζ are L_ζ, then the availability of the i^{th} system is expressed as $A_i(\zeta) = \frac{W_i}{L_\zeta}$. We have $\aleph_0$ identical systems running in parallel, and each system performs L_ζ operation cycles. Therefore, system availability, on average, is expressed as

$$A_{sys}(\zeta) = \frac{\sum_i^{i=N}(W_i/\aleph_0)}{L_\zeta}$$

where $\sum_i^{i=N}(W_i/\aleph_0)$ is average number of times systems are in working state till time ζ and L_ζ is total number of operations till time ζ.

4. Case study

We have applied our proposed method to an aircraft fuel control system as a case study. The working principle of the aircraft fuel control system is available at the Mathworks website [49]. As depicted in Mathworks website, an aircraft fuel control system comprises an engine, fuel tank, pump, fuel rate controller (microcontroller), actuator, and four sensors. The sensors are throttle sensor, engine fan speed sensor, exhaust gas oxygen (EGO) sensor, and manifold absolute pressure (MAP) sensor. The functional details of each component will be discussed in the subsequent sections. During the application of the proposed reliability and availability method, we must have the SysRS document of the system. In this regard, the information at the Mathworks website [49] about the aircraft fuel control system available is treated as the SysRS document. Subsequently, the failure modes of the system components and probability of occurring each failure mode are identified, referring to some standard reliability handbook/ data-sources [52–54].

We have applied the proposed method to the Aircraft Fuel Rate Control System (AFRCS), which is a part of the Aircraft Fuel Control System (AFCS). AFRCS consists of the fuel rate controller (FRC) whose inputs are provided by four sensors (see Fig. 2). We have used IBM Rational Software Architect (RSA) for constructing the Domain Model and Component Behavior Model of the proposed method. However, other tools, like Papyrus, Enterprise Architect, or any UML modeling tool that supports the MARTE extension for creating real-time scenarios, can be used. Some papers [1,17] recommend the use of IBM RSA or Papyrus as these tools support exporting files (.uml) in EMF format. This format is widely used for interchanging UML models among different tools. An EMF format file stores a list of information that comprises the drawing commands, property definitions, and graphics objects. The system modeling tools that support file exporting in EMF format enhance the model's portability. A group of researchers prefers IBM RSA as it has a detailed user guide available on its official website [1,17,55,56]. The developer community of the IBM RSA is quite big and active in responding to queries. In contrast, resources available for learning other tools are scanty, which makes it difficult to use by beginners. Some researchers prefer to use Papyrus over IBM RSA and Enterprise Architect as the former is open-source software. A commercial tool is essential if anyone plans to work with a team collaboratively. Though we have used IBM RSA for its detailed learning resources, an active developer community, and collaborative development framework, users of the proposed method may choose other tools (Papyrus, Enterprise Architect, etc.) based on their requirements. During the simulation process, we use NetBeans IDE for Java coding and execution of the software controller.

4.1. Environment modeling profile

During the environment modeling of AFRCS, we use extensions of generic UML for customizing the primitive UML data types. Though the UML profile supports customization of the primitive data types, it is not sufficient for modeling real-time applications. For example, the UML profile does not support time expression. In this regard, we use the MARTE extension that supports real-time

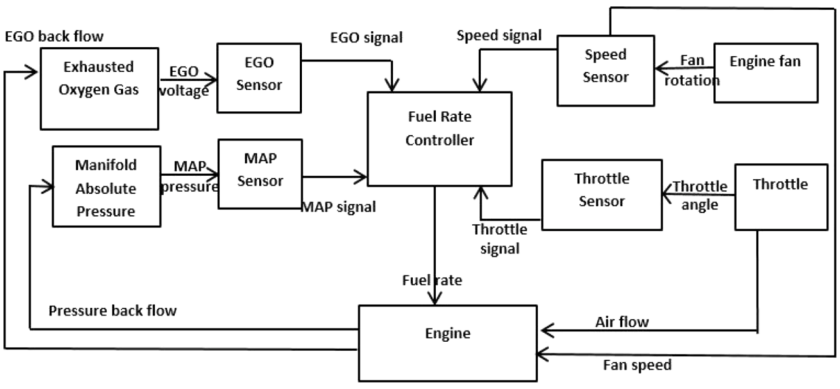


Fig. 2. Block diagram of Aircraft Fuel Control System.

application-specific customization. MARTE extension allows imposing constraints or conditions on the RTES environment scenarios. For example, we can define the delay in feedback signal, span of an event, etc. UML is a general-purpose standard modeling language that is meant to cover a wide range of application domains. It comprises a large volume of semantics. MARTE is a comprehensive UML profile that covers all the features required to model RTES. MARTE also comprises a large volume of semantics. It becomes important to identify the subsets of UML and MARTE that are required for a particular RTES application. A user would be reluctant to adopt a full set of UML and MARTE semantics and define a custom profile using a UML and MARTE subset. The major benefit of a custom-defined profile is that it uses lightweight extensions of the parent UML and MARTE. A custom-defined profile also helps to avoid conflict with other original semantics of UML or MARTE. A custom-defined profile has certain disadvantages. Adopting a full set of UML and MARTE semantics is almost effortless, and it does not put any limit on the usability. If system requirements change at the later phases, there is no need to add or remove any new UML or MARTE subset.

We have shown our environment modeling profile in Fig. 3. The environment modeling profile prevents newly added UML extensions from contradicting standard semantics. We use standard UML and its existing extensions to model the environment of the RTES. As depicted in Fig. 3, we have defined a set of stereotypes to model the environment of the RTES. These stereotypes will be used during the domain modeling and behavior modeling, in the latter part of this paper. Some subset of MARTE that has also been incorporated in the profile is shown in Fig. 3. For example, we have used the Time package, the Non-functional Properties (NFP) package, and the Generic Quantitative Analysis Modeling (GQAM) package. As the name suggests, NFP is used to define the non-functional properties of system (like memory use, delay, throughput, etc.). On the other hand, GQAM is used to define time interval, schedulability, etc. of the system functionalities.

4.2. Application of the proposed reliability and availability method

We apply the proposed method to AFRCS for reliability and availability prediction. Each step of the method demonstrated below.

4.2.1. Domain modeling

The domain model provides structural information of the AFRCS. It defines the composition of the components, their properties, relationships with each other, and their cardinalities. We use the UML class diagram for modeling the domain model. The domain model of the AFRCS is depicted in Fig. 3.

Identification of the components: The AFRSC comprises five hardware components and a software controller (see Fig. 4). The hardware components are microcontroller, exhaust gas oxygen (EGO) sensor, manifold pressure (MAP) sensor, throttle sensor, and engine fan speed sensor. The AFRSC has two forward sensors (throttle and fan speed) and two feedback sensors (EGO and MAP) that feed the signals to the microcontroller. Once the microcontroller receives all the input signals from the sensors, it passes them to the software controller for further processing. The OCL constraints are used to provide the range of the sensors' signals. These ranges are

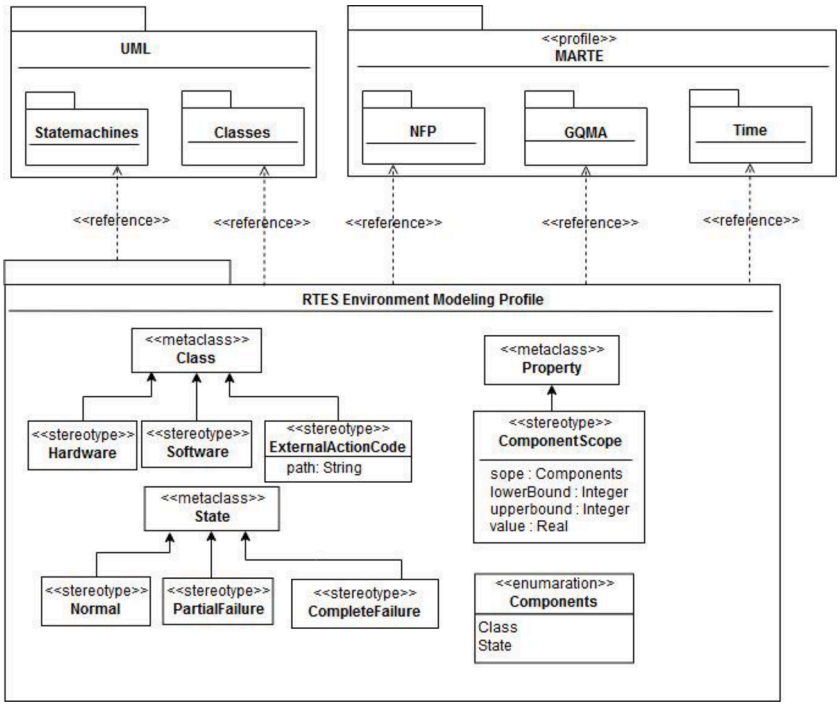


Fig. 3. Environment modeling profile for AFRCS.

taken into consideration during initializing the simulation process.

Defining the dependency among the components: We include all the physical/ logical relationships among the AFRCS components that are required to establish communication. The dependency among all components of the AFRCS is depicted in Fig. 4.

Operation Mode of the Hardware Components: The operational modes of a hardware component are divided into three types: a) normal, b) partial failure/ degraded, and c) complete failure. The partial failure/ degradation modes of the generic sensors have been identified from the reliability data-source [57]. We found partial failure/ degradation may cause deviation of the signal with additive/ subtractive polarity. As observed in the reliability data-source [57] these degraded modes are: a) incorrect signal/ calibration error, b) error in the transmission line, c) error computation device, and d) improper response to the recipient [57]. Again, complete failure modes of sensors that cause loss of signal are: a) loss of signal from sensor, b) loss of signal from the transmission line, c) short circuit, and d) open circuit [57]. In first and fourth column of Table 1 we have listed the above-mentioned partial and complete failure modes of the generic sensors.

We have identified the generic failure modes of the microcontroller. As identified in the reliability data-sources [52–54], the partial failure/ degraded modes of the microcontroller are: a) erroneous data for untimely transmission, b) intermittent, and c) impossible transmission. As identified in the literature [52–54], the complete failure modes of the microcontroller are: a) short circuit, b) open circuit, c) loss of signal, d) delay/ pulse, e) memory failure/ indeterminate, and f) abnormal signal sequence. All operation modes of the microcontroller are shown in the first and fourth columns of Table 2.

Modeling abstract software controller: The software controller class is stereotyped as «System». It is treated as an entirely abstract class. During the code generation phase, we implement it. We have depicted the relationship of the software controller class with other environment classes (see Fig. 4).

4.2.2. Hardware component behavior model

We define behavior model for each component of the domain model. As already mentioned, a component may have multiple operation modes. During the execution of the system, one of the operation mode occurs at a time. It is called the current operation state of the component. The behavior model presents the response of the component at each state and the state transition conditions of the component. We use UML state machine diagram to construct the behavior models. The main steps of constructing behavior model are below:

Behavior of the component states: The behavior of the component at each state is distinct. In Table 1, we have listed the failure modes/ states of sensors. The probability of occurring each failure mode is also given in Table 1, as identified in the literature [53]. We consider, on-demand maximum one failure in ten thousand observation, as we assume system requirement specification instructed to use sensor components that quality Safety integrity level (SIL) 4, which is defined in IEC EN 61508. As represented in Table 1, during the complete failure of the sensor, the signal strength becomes zero except the short circuit case, in which it rises abruptly to high values. On the other hand, during partial failure, the signal strength erroneously deviates from the actual value. This error follows the standard normal distribution $N(\mu=0, \sigma)$. We assume standard deviation (σ) of the sensor signal due to incorrect signal, error in transmission line, error in computation device, and improper response to recipient as 0.3%, 0.4%, 0.5%, and 0.6% of the actual signals, respectively. All partial/ complete failure modes of the generic sensors, probability of occurring each mode, and sensors' response

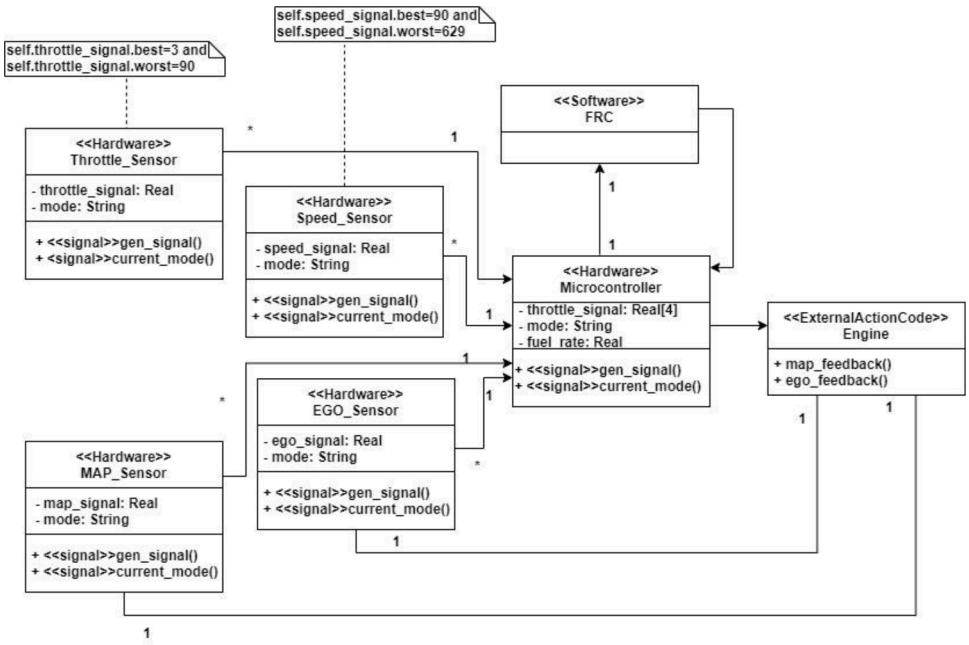


Fig. 4. Domain model of the AFRCS.

Table 1
Failure modes, probability of failure modes, expected component behavior at each failure mode of an EGO sensor.

Complete Failure			Partial Failure			
Failure Modes (j)	Probability of occurring failure mode j	EGO sensor response at failure mode j	Failure Modes (j)	Probability of occurring failure mode j	EGO sensor response at failure mode j $M_{ij}^{n}(x) = C_i(\boldsymbol{Normal}) + \varepsilon$ Where standard error $\varepsilon = N(0,\sigma)$	
					Additive polarity	Subtractive polarity
Loss of signal from sensor	0.0001	*No_signal*	Incorrect signal	0.00058	$C_i(Normal)+N(0,0.003 \times C_i(Normal))$	$C_i(Normal)-N(0,0.003 \times C_i(Normal))$
Loss of signal from transmission line		*No_signal*	Error in transmission line		$C_i(Normal)+N(0,0.004 \times C_i(Normal))$	$C_i(Normal)-N(0,0.004 \times C_i(Normal))$
Short circuit	0.0002	∞	Error in computation device		$C_i(Normal)+N(0,0.005 \times C_i(Normal))$	$C_i(Normal)-N(0,0.005 \times C_i(Normal))$
Open circuit	0.00012	*No_signal*	Improper response to recipient		$C_i(Normal)+N(0,0.006 \times C_i(Normal))$	$C_i(Normal)-N(0,0.006 \times C_i(Normal))$

during each failure modes are shown in Table 1.

All failure modes of the microcontroller are shown in Table 2 [52,54]. The probability of occurrence each failure modes is also given in the table, as identified in the literature [52,54]. Here also we consider, on-demand maximum one failure in ten thousand observations, as we assume system requirement specification instructed to use a microcontroller that quality Safety integrity level (SIL) 4, which is defined in IEC EN 61508. As represented in Table 2, during the complete failure of the microcontroller, the signal strength becomes zero except the short circuit case, in which it rises abruptly to high values. The response of the microcontroller erroneously deviates from the actual value during the partial failure. We assume error follows the standard normal distribution $N(\mu=0, \sigma)$ where standard deviation (σ) is 0.1% of the actual signal. All partial/ complete failure modes of the generic microcontroller, probability of occurring each mode, and microcontroller's response during each failure modes shown in Table 2.

Modeling state transition: During the simulation process, state transition of a component occurs due to operational conditions. The behavior model identifies the operational condition for each state of a component. The change of an operational condition occurs due to two reasons: triggering off some time event and executing some Class method. An example of a component's state transition due to time event is the EGO sensor's state transition from *warm-up* state to *normal* state. This state transition is triggered by a time event *after ()*, as shown in Fig. 5. On the other hand, an example of a component's state transition due to Class method is the EGO sensor's state transition from *normal* to any *failure* state. This state transition is determined by the outcome of a method *current_mode()* of EGO Class (see Fig. 5).

4.2.3. Generation of the environment simulator

The environment simulator is generated by converting the environment models (domain model and behavior model) to equivalent software code. The model-to-code transformation process follows the method given by Iqbal et al. [1] with some modifications. The modified model-to-code transformation process is briefly discussed here. The architecture of the environment simulator is shown in Fig. 6. It comprises six Java classes/ packages:

1 *Environment Models:* It consists of domain model and behavior model. We mark them with a stereotype «*artifact*». The environment models are the input to Simulator Generator package.
2 Simulator Generator (SG): It includes some library package/ driver class that converts the environment models to Java code:
- *TransformationDrivers* contains major library classes that convert UML artifacts to code.
- *MofscriptTransformations* contains library classes that convert the *MOFscript* [58] of the models to Java code.
- *OCLToJavaTranslator* is used by the *MofscriptTransformations* to translate the OCL expression to code.
- *EnvironmentConfigurationGenerator* defines the environment configuration for test setup.

The outputs of the SG are the *EnvironmentSimulator* package and the *EnvironmentConfiguration* package.

1 *EnvironentSimulator* package: It includes all the Java classes that simulate different operational scenarios for the testing of the software controller.
2 *EnvironmentConfiguration* class: This class provides the setting of the operating environment.
3 *OCLConstrainsSolver* package: It provides the library classes for converting the OCL constraints to Java code.

Table 2
Failure modes, probability of failure modes, expected component behavior at each failure mode of Microcontroller.

Complete Failure			Partial Failure			
Failure Modes (j)	Probability of occurring failure mode j	Microcontroller response at failure mode j	Failure Modes (j)	Probability of occurring failure mode j	Microcontroller response at failure mode j $M_{ij}^{*}(x) = C_i(Normal) + \varepsilon$ Where standard error $\varepsilon = N(0,\sigma)$	
					Additive polarity	Subtractive polarity
Short circuit	0.00170	∞	Erroneous data	0.00026	$C_i(Normal)+N(0,0.003\times C_i(Normal))$	$C_i(Normal)-N(0,0.003\times C_i(Normal))$
Open circuit	0.00051	*No_signal*				
Loss of signal		*No_signal*				
Delay/ pulse		*Insert_one_unit_delay*				
Memory failure/ indeterminate	0.0006	*No_signal*				
Abnormal sequence		*No_signal*				
Intermittent		*No_signal*				
Impossible transition		*No_signal*				

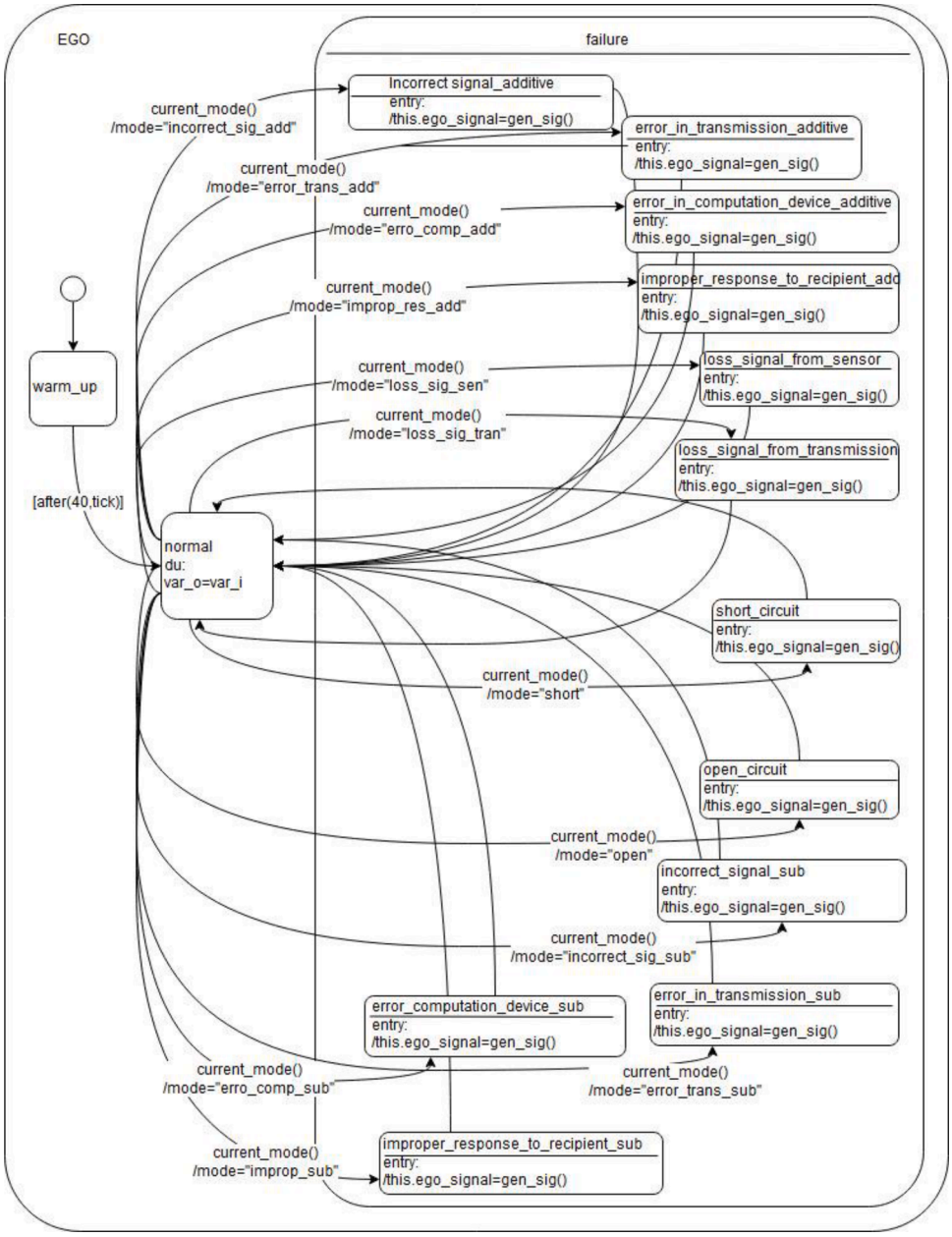


Fig. 5. Behavior model of the EGO Sensor.

4 *SimulatorSupportUtil* package: It helps the SG to fix the sequence of operations, setting their priorities, etc. We treat *SimulatorSupportUtil* as a library because its classes remain same for all applicants. A brief discussion of *SimulatorSupportUtil* library is given below.

The core packages of the *SimulatorSupportUtil* library are depicted in Fig. 7. It includes the following classes:

1 *ActiveObject: ActiveObject* class represents the concept of UML active object. It provides an event queue for the active objects in the form of *java.util.PriorityBlockingQueue*. Its purpose is prioritizing the execution of class instances in the queue. An object of the *java.util.PriorityBlockingQueue* stores instances of *EventInvocation* class.

2 *EventInvocation*: An instance of *EventInvocation* class represents an event that is ready to be executed and is placed in the event queue of an ActiveObject. *EventInvocation* class has three attributes.

- *methodToInvoke* is of type *java.lang.reflect.method* along with its parameter types.
- *parameterValues* contains the parameters (as Java List) of the methods to be invoked.

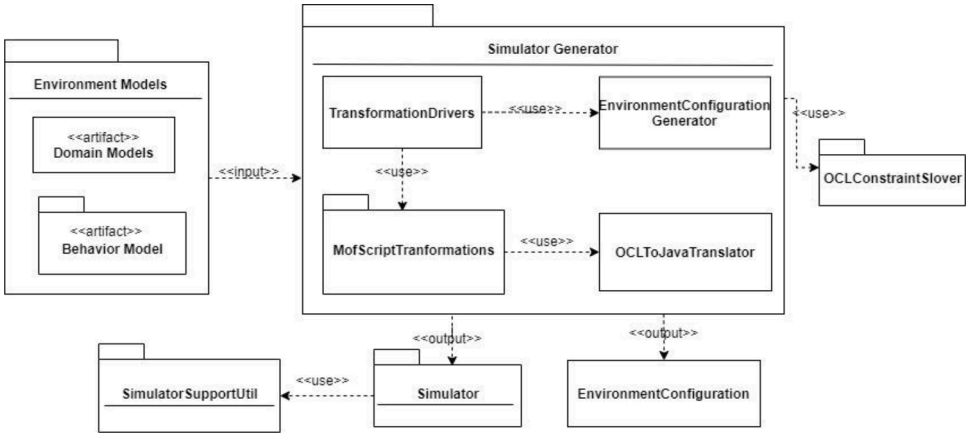


Fig. 6. Architecture of the simulation framework.

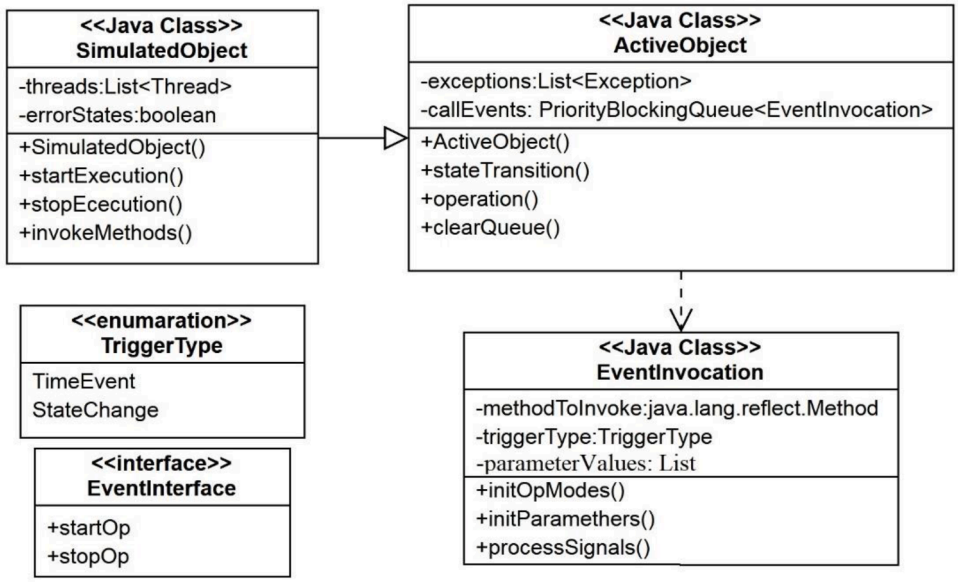


Fig. 7. Important classes in the *SimulatorSupportUtil*.

- *triggerType* represents an instance of class TriggerType (change event or time).

SimulatedObject: It holds a list of threads that have been created for the instance of an environment component.

EventInterface: The *EventInterface* is implemented by all the classes representing UML states.

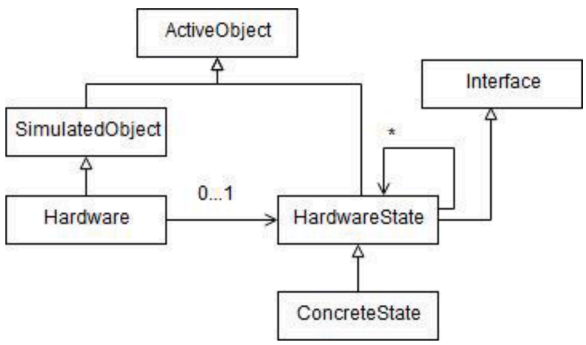


Fig. 8. The meta-model of the extended state pattern.

5 *TriggerType*: An instance of class TriggerType fire a new event based on time function or other constraints.

Fig. 8 shows the proposed meta-model of the extended *state pattern* [51]. The extended state pattern has added some new meta-classes (*ActiveObject, EventInterface*, and *SimulatedObject*) to support environment simulation. Other meta-classes directly inherited from the original state pattern. They are renamed as *Hardware* (called *Context* in the original state pattern), *HardwareState* (called *State* in the original state pattern), and *ConcreteState*. The role of these meta-classes is kept same as described in the original state pattern. The *ConcreteState* represents all the states that exclude the exceptional operating condition.

Replacing abstract software controller with code: We replace the abstract software controller with Java code. The code is generated following the software controller's working principle available at the MathWorks website [49]. We perform software-in-the-loop simulation of the software controller using ES. The software controller of the AFRCS broadly has two operation modes: fuel running (working) and disabled (failure). If all system components are working properly, fuel flows at a constant rate. This is considered as the fuel running mode (working). This running mode can be analyzed further into two types: low emission mode (normal working) and rich mode (fault-tolerant) based on the proportion of the air and oil in the fuel. The running mode turns to disable mode, if more than one sensor or microcontroller fails.

4.2.4. Software controller behavior model

The environment simulator (ES) simulates random test cases for the testing of the software controller. We observe the state transition of the software controller taking place due to various test cases and determine its behavior model. The main steps of the software controller behavior model are described below:

4.2.4.1. Test scenario generation. The test scenario generation process consists of the following steps: 1) ES sets a random value as initial input for each sensor of the AFRCS. These input signals must be within the valid range of the sensors, as given in Table 3, 2) ES samples the operation modes of the sensors. As a result, any of the operation modes described in Table 1, randomly occurs. Based on the current operation modes, the sensors feed input to the microcontroller. 3) ES samples the operation modes of the microcontroller and any of the operation mode described in Table 2, randomly occurs. Based on the current operation mode of the microcontroller, a test case is generated. The ES feeds the test case to the software controller and observes the system state transition. In the below section, we try to group all test cases that have a similar impact on the software controller.

Test Scenarios 1 (TS1): System state transition from normal state to fault-tolerant state due to input peripheral(s) failure. The failure of a set of components causes the system state to transit from normal state to fault-tolerant state. If the microcontroller is in normal working state and one of the four sensors (throttle/ speed/ EGO/ MAP) fails, the AFRCS state transits from normal state to fault-tolerant state [49]. In this case study, any test case that comprises at most one sensor failure, belongs to TS1.

Test Scenarios 2 (TS2): System state transition from normal/fault-tolerant state to failure state due to input peripheral(s) failure. The failure of some components causes the system state to transit from normal/ fault-tolerant state to complete failure state. If the microcontroller is in the normal working state and a combination of any two or more sensors fails, the AFRCS state transits from normal/ fault-tolerant state to failure state [49]. In this case study, any test case that comprises a combination of any two or more sensors failure, belongs to TS2.

Test Scenarios 3 (TS3): System state transition from normal/fault-tolerant to failure due to permanent failure of the microcontroller. The permanent failure of a critical component causes the system state to transit from normal/ fault-tolerant state to failure state. If the microcontroller fails, the AFRCS state transits from normal/ fault-tolerant state to failure state [49]. In this case study, any test case that comprises microcontroller failure, though all/ some sensor is in working state, belongs to TS3.

Test Scenarios 4 (TS4): System state transition from normal/fault-tolerant to failure due to transient failure of the microcontroller. The transient failure of a critical component causes system state to transit from normal/ fault-tolerant state to a failure state. Such transient failure typically occurs due to hardware-related software interaction failure. We have demonstrated two types of hardware-related software interaction failures: memory inaccessibility and delay [59], as demonstrated in Table 4. In this case study, any test case that comprises microcontroller is in the transient failure state, though, all/ some sensors are normal in the working state, belongs to TS4.

Test Scenarios 5 (TS5): System state transition from normal/fault-tolerant to failure due to degradation of the input peripheral. The degradation of one set of components causes the system state to transit from normal state to fault-tolerant state. If the microcontroller is in normal working state and a set of sensors is operating in a degraded state, the AFRCS state may transits from normal/ fault-tolerant state to failure state. The degraded state of the sensors may create exceptional input conditions, out of range inputs, logical errors, etc. Due to such exceptional scenarios, the control signal generated by the software controller may cause malfunctioning of the associated peripheral components. This is considered as software-related hardware interaction failure. We refer SysRS to identify potential exceptional input conditions (see Table 4) of the AFRCS, created by the set of degraded sensors.

Table 3
Acceptance range of the input signals.

	Throttle Angle (TA) (in degree)	Fan Speed (FS) (in rad)	EGO (in volt)	MAP (in bar)
Acceptance Range	3< TA< 90	50<FS<628	EGO<1.2	0.05<MAP<0.95

Table 4
Exceptional input conditions for AFRCS strength.

Sl/No.	Exceptional input conditions	Sl/No.	Exceptional input conditions
1	Throttle <3 degree & Speed <50 rpm	10	Throttle >90 degree & MAP > 0.95 bar
2	Throttle <3 degree & Speed >628 rpm	11	Speed <50 rpm & EGO > 1.2 volt
3	Throttle <3 degree & EGO> 1.2 volt	12	Speed <50 rpm & MAP < 0.05 bar
4	Throttle <3 degree & & MAP< 0.05 volt	13	Speed <50 rpm & MAP > 0.95 bar
5	Throttle <3 degree & MAP > 0.95 bar	14	Speed >628 rpm & EGO > 1.2 volt
6	Throttle >90 degree & Speed < 50 rpm	15	Speed >628 rpm & MAP < 0.05 bar
7	Throttle >90 degree & Speed >628 rpm	16	Speed >628 rpm & MAP > 0.95 bar
8	Throttle <3 degree & EGO> 1.2 volt	17	EGO> 1.2 volt & MAP < 0.05 bar
9	Throttle >90 degree & MAP < 0.05 bar	18	EGO >1.2 volt & MAP > 0.95 bar

4.2.4.2. Software controller behavior simulation. At the beginning of the simulation process, we assume that the software controller is in the normal state. The ES simulates random test cases that construct failure scenarios (hardware-specific failure/ software-specific failure/ hardware-software interaction failure). Such test cases cause state transition of the AFRCS. Here, we try to group the test scenarios that cause similar type of state transition of the software controller.

The state of the software controller transits from normal state to fault-tolerant state for TS1. If the microcontroller is in normal working state and any of the four sensors (throttle/ speed/ EGO/ MAP) fails, the AFRCS state transits from normal state to fault-tolerant state. All the test cases that trigger the AFRCS state transition from normal state to fault-tolerant state belong to the group Test Scenarios 1 (TS1).

The software controller state transits from fault-tolerant state to failure state for TS2, TS3, TS4 or *TS5*. Let us assume that the AFRCS is in the fault-tolerant state due to any of the four sensors (throttle/ speed/ EGO/ MAP) failure. Now, the state of AFRCS may transit from fault-tolerant state to failure state due to any of the following events: a) one or more sensors fail (test case belong to TS2), b) microcontroller fails (test case belong to TS3), c) exceptional input condition due to degradation of hardware components (test case belong to TS4), or d) transient failure of the hardware components (test case belongs to TS5). All the test cases that trigger the AFRCS state transition from normal state to fault-tolerant state belong to the group Test Scenarios 2 (TS2), Test Scenarios 3 (TS3), Test Scenarios 4 (TS4), or Test Scenarios 5 (TS5).

The software controller state transits from normal state to failure state for TS2, TS3, TS4 or *TS5*. Let us assume that the AFRCS is in normal state as all the sensors and microcontroller are in normal working state. Now, the state of AFRCS may transit from fault-tolerant state to failure state due to any of the following events: a) one or more sensors fail (test case belong to TS2), b) microcontroller fails (test case belong to TS3), c) exceptional input condition due to degradation of hardware components (test case belong to TS4), or d) transient failure of the hardware components (test case belongs to TS5). All the test cases that trigger the AFRCS state transition from normal state to fault-tolerant state belong to the group Test Scenarios 2 (TS2), Test Scenarios 3 (TS3), Test Scenarios 4 (TS4), or Test Scenarios 5 (TS5).

4.2.5. Reliability and availability prediction

Our method predicts the reliability and availability based on functional failure. We refer to the SysRS document to a valid range of system response for the admissible set of input(s). In any instance, if the software controller's response violates the given acceptance range, it is considered as functional failure. During reliability prediction, we do not consider the probability of hardware getting repaired.

We have calibrated the failure behavior of the hardware environment model before predicting the reliability and availability of the system. To calibrate the failure behavior, we have simulated random test cases as mentioned in the above section. Based on the simulated test cases, we estimate the deviation of the failure probabilities of the hardware component from the assumed failure probabilities. The third row of the Table 5 represents the deviations in percentage.

We generate ten thousand identical systems ($\aleph_0 = 10000$) using ES. Each of these system executed in parallel for 100000 simulation operations ($L = 100000$). If each operation cycle takes unit time, then 100000 simulation iterations takes 100000 unit time. We divide the operation time into 100 equal intervals and record the number of systems failed in each interval. Subsequently, we also evaluate the number of systems survived in each interval. Let us consider, at ζ^{th} interval ($\zeta \in \{1,i\}$) total number of systems' failure is $\aleph_f$. Therefore, the number of systems' survival at ζ^{th} interval is $\aleph_s$ ($\aleph_s = \aleph_0 - \aleph_f$). Then, for each interval we evaluate the ratio of number of system survived at that interval to the total number of system under testing ($\aleph_s/\aleph_0$). We plot the ratio $\aleph_s/\aleph_0$ against the number of intervals $i\,(i = 100)$ and get the transient reliability graph (see Fig. 9).

During availability prediction, we consider 10 thousand identical systems are operating in parallel. We evaluate average steady-

Table 5
Number of failures, failure probability, and calibration of the hardware failure probabilities.

	Throttle Sensor	Engine Fan Speed Sensor	EGO Sensor	MAP Sensor	Microcontroller
Number of Failures out of 100000 Iterations/ Operations	11	14	7	9	9
Failure Probability (FP)	0.00011	0.00014	0.00007	0.00009	0.00001
Deviations of FP from Assumption (in percentage)	0.01	0.013	0.006	0.008	0

state availability of the systems using simulation process. We consider each system as repairable system. If any component failure occurs, we repair/ replace it as good as new. During simulation process, the time taken to recover the failure is negligible as compare to the full operation cycle. Initially, we set the number of simulation operation (L) to 500 for each system. Let us consider, for the i^{th} system the estimated number of times transitions G_{NT} (normal to fault-tolerant), G_{NF} (normal to failure), and G_{TF} (fault-tolerant to failure) occurs are x_i, y_i, and z_i, respectively. So, the total number of times i^{th} system response undergoes functional failure is estimated as $(y_i + z_i)$ and total number of times system is in working state is W_i ($W_i = L - y_i + z_i$). If the total number of simulation operations performed till time ζ are L_ζ, then the availability of the i^{th} system is expressed as $A_i(\zeta) = \frac{W_i}{L_\zeta}$. We have $\aleph_0$ identical systems running in parallel, and each system performs L_ζ operation cycles. Now, we step by step increase the number of operations until the value of A_i reaches to a steady-state. We use similar method to evaluate steady-state availability of all the 10 thousand systems. Once all the systems reached to its steady-state, we evaluate the average steady-state availability of the systems using the expression $A_{ss}(\zeta) = \frac{\sum_i^{i=N}(W_i/\aleph_0)}{L_\zeta}$ where $\sum_i^{i=N}(W_i/\aleph_0)$ is average number of times a system is in working state till time ζ and L_ζ is total number of operations till ζ. In this case study, we observed after 100000 simulation operation all the A_i values reach to its steady-state and the value is 0.94871 (see Fig. 10).

In this section, we have predicted the reliability and availability of the AFRCS using the proposed method. Many researchers have reported early testing of safety-critical embedded systems using environment modeling [1,17–22]. They have not addressed prediction of the reliability and availability of the system. We have extended the environment modeling based testing approach for reliability and availability prediction. As demonstrated in the case study, our method has considered all possible types of failures –hardware-specific failures, software-specific failures, hardware-related software interaction failures, and software-related hardware interaction failures. At present, there is a scarcity of a unique reliability and availability method that has considered all possible failures. For example, some Markovian model-based approaches have only considered independent component failures (hardware-specific and software-specific) [23,60]. Another group of Markovian models has considered hardware-related software interaction failures [5,10–13]. They have ignored the possibility of software-related hardware interaction failures. FFIP based models have considered both, hardware-related software and software-related hardware interaction failures [15,42,43,47,61,62]. However, they have restricted their study to qualitative reliability analysis. Our proposed method gives quantification of the reliability and availability of the system. Moreover, it enables the developers to uncover hardware-software interaction failures even before integrating the hardware and software. Our method can significantly reduce the production costs for high-reliability systems. In this case study, if the AFRCS does not meet the target reliability during the post-HW-SW integration testing, then it would entail considerable redevelopment costs. Therefore, the proposed method enables the developers to reasonably predict the reliability and availability of the embedded systems when the target hardware is not accessible

5. Validation of the proposed method

We have selected a set Markovian model [5,11,23], a set of FFIP based models [13–16], and set of SILS based models [1,24,28] that perform quantitative reliability prediction or qualitative reliability analysis or availability prediction or testing of the embedded system. We compare the above models with the proposed method based on their reliability concern (hardware-specific failure, software-specific failure, hardware-related software interaction failure, and software-related hardware interaction failure) and goal of prediction (reliability and availability). We have not included Costes et al. [10] as it is applicable only to the stand-by systems, which is not in our scope. As depicted in Table 6, the CTMC model [23] and the proposed method have provided the reliability and availability prediction of a system. Other models have not provided availability prediction. We are more interested in availability value as steady-state availability is independent of time, and it can be used as a more suitable criterion for validation.

To develop CTMC model we use similar approach as demonstrated by Zeng et al. [23]. They used Generalized Stochastic Petri Nets (GSPNs) tool PIPE 4.3 for this purpose. The GSPN gives a reachability graph that is equivalent to CTMC. The CTMC model is used to predict the transient reliability and steady-state availability of the AFRCS. Finally, to validate the proposed method, we compare the obtained reliability and availability obtained using our proposed method with the values obtained using the CTMC model. Fig. 11 shows a GSPN of the AFRCS with one place enabled containing four tokens.

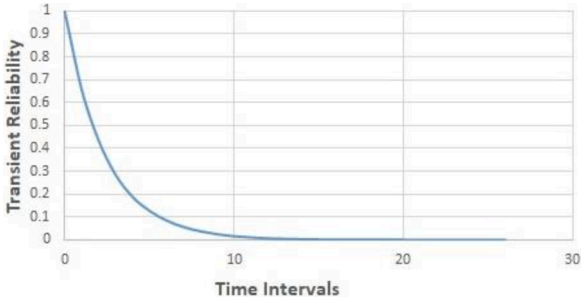


Fig. 9. Reliability curve.

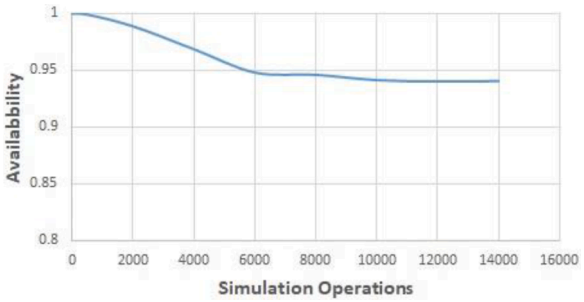


Fig. 10. Availability curve.

Table 6
Comparison of reliability and availability prediction methods.

Type of Models/ Methods	Models/ Methods	Hardware-specific Failure	Software-specific Failure	Hardware-related software Interaction Failure	Software-related Hardware Interaction Failure	Reliability Prediction	Availability Prediction
Markovian Models	Teng et al. (2006) [5]	YES	YES	YES	NO	YES	NO
	Zeng et al. (2012) [23]	YES	YES	YES	NO	YES	YES
	Roy et al. (2014) [11]	YES	YES	NO	NO	YES	NO
FFIP based method	Jensen et al. (2008) [14]	YES	YES	YES	YES	YES (Qualitative analysis only)	NO
	Tumer and. Smidts (2011) [16]	YES	YES	YES	YES	YES (Qualitative analysis only)	NO
	Sierla et al. (2012) [15]	YES	YES	YES	YES	YES (Qualitative analysis only)	NO
	Diao et al. (2018) [13]	YES	YES	YES	YES	YES (Qualitative analysis only)	NO
	Irshad et al. (2020) [48]	YES	YES	YES	YES	YES (Qualitative analysis only)	NO
SILS based Testing Models	Auguston et al. (2006) [24]	YES	YES	YES	YES	NO	NO
	Krichen and Tripakis (2009) [28]	YES	YES	YES	YES	NO	NO
	Iqbal et al. (2015) [1]	YES	YES	YES	YES	NO	NO
	Silano and Iannelli (2020) [32]	YES	YES	YES	YES	NO	NO
Proposed SILS based Method	Proposed Method	YES	YES	YES	YES	YES	YES

Fig. 12 represents the reachability graph produced by GSPN. The 16 tangible states of the reachability graph constitute the CTMC model. If Q denotes the infinitesimal generator and q_{ij} $(i \neq j)$ denotes the rate of transition from state M_i to state M_j , then the infinitesimal generator is defined as $Q = [q_{ij}]$. The value of q_{ij} will be zero ($q_{ij} = 0$) if there is no connection between the states M_i and M_j . In this model, the infinitesimal generator is expressed as $Q = -\sum_{i=1}^{j=16, j\neq i} q_{ij}$. If the steady-state vector denoted as P, then $P = \{P_1, P_2, \ldots, P_{16}\}$. We express this as:

$$P \times Q = 0$$

$$\sum_{i=0}^{16} P_i = 1$$

The transient probability of each state is denoted as $P(t)$ where $P(t) = \{P_1(t), P_2(t), \ldots, P_{16}(t)\}$. As the reachability graph has 16 tangible states so $P(t)$ constitutes 16 first-order linear differential equations. In these equations failure probability of the MAP sensor(λ_m), EGO sensor(λ_e), speed sensor(λ_s), throttle sensor (λ_t), microcontroller failure (λ_u), and their repair probability(μ) are

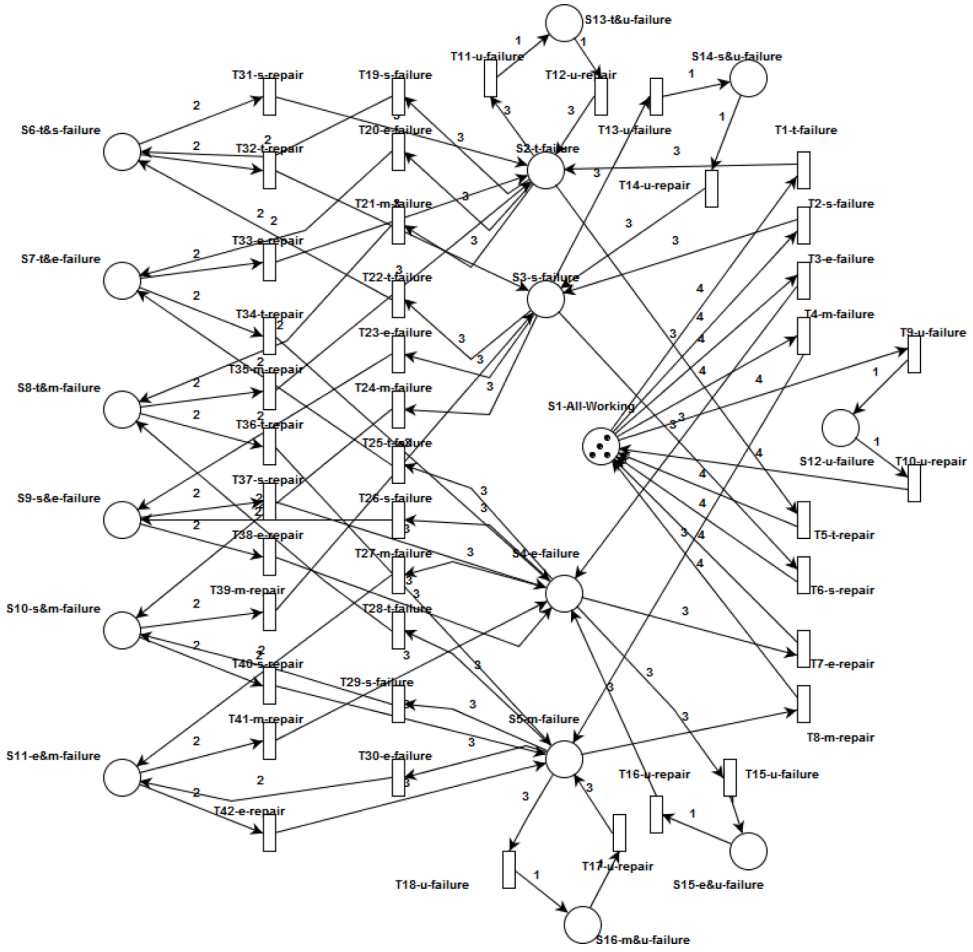


Fig. 11. Petri Nets model of the Fuel Rate Controller System (AFRCS).

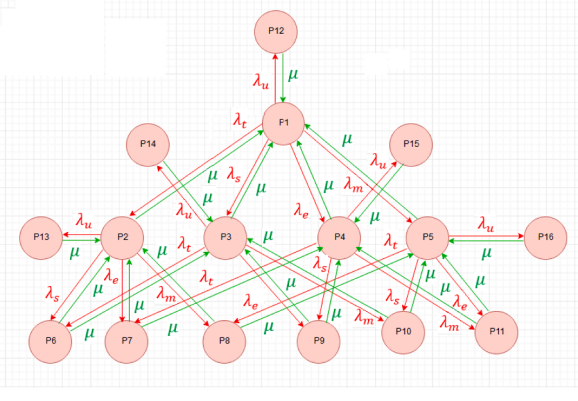


Fig. 12. Reachability graph of the Petri Nets mode.l

known. The equations are as following:

$$1\ \frac{d}{dt}P_1(t) = -(\lambda_t + \lambda_s + \lambda_e + \lambda_m + \lambda_u)P_1(t) + \mu P_2(t) + \mu P_3(t) + \mu P_4(t) + \mu P_5(t) + \mu P_{12}(t)$$

$$2\ \frac{d}{dt}P_2(t) = -(\mu + \lambda_s + \lambda_e + \lambda_m + \lambda_u)P_2(t) + \lambda_t P_1(t) + \mu P_6(t) + \mu P_7(t) + \mu P_8(t) + \mu P_{13}(t)$$

$$3\ \frac{d}{dt}P_3(t) = -(\mu + \lambda_t + \lambda_e + \lambda_m + \lambda_u)P_3(t) + \lambda_s P_1(t) + \mu P_6(t) + \mu P_9(t) + \mu P_{10}(t) + \mu P_{14}(t)$$

4 $\frac{d}{dt}P_4(t) = -(\mu + \lambda_t + \lambda_s + \lambda_m + \lambda_u)P_4(t) + \lambda_e P_1(t) + \mu P_7(t) + \mu P_9(t) + \mu P_{11}(t) + \mu P_{15}(t)$

5 $\frac{d}{dt}P_5(t) = -(\mu + \lambda_t + \lambda_s + \lambda_e + \lambda_u)P_5(t) + \lambda_m P_1(t) + \mu P_8(t) + \mu P_{10}(t) + \mu P_{11}(t) + \mu P_{16}(t)$

6 $\frac{d}{dt}P_6(t) = -2\mu P_6(t) + \lambda_s P_2(t) + \lambda_t P_3(t)$

7 $\frac{d}{dt}P_7(t) = -2\mu P_7(t) + \lambda_e P_2(t) + \lambda_t P_4(t)$

8 $\frac{d}{dt}P_8(t) = -2\mu P_8(t) + \lambda_m P_2(t) + \lambda_t P_5(t)$

9 $\frac{d}{dt}P_9(t) = -2\mu P_9(t) + \lambda_s P_4(t) + \lambda_e P_3(t)$

10 $\frac{d}{dt}P_{10}(t) = -2\mu P_{10}(t) + \lambda_s P_5(t) + \lambda_e P_3(t)$

11 $\frac{d}{dt}P_{11}(t) = -2\mu P_{11}(t) + \lambda_e P_5(t) + \lambda_m P_4(t)$

12 $\frac{d}{dt}P_{12}(t) = -\mu P_{12}(t) + \lambda_u P_1(t)$

13 $\frac{d}{dt}P_{13}(t) = -\mu P_{13}(t) + \lambda_u P_2(t)$

14 $\frac{d}{dt}P_{14}(t) = -\mu P_{14}(t) + \lambda_u P_3(t)$

15 $\frac{d}{dt}P_{15}(t) = -\mu P_{15}(t) + \lambda_u P_4(t)$

16 $\frac{d}{dt}P_{16}(t) = -\mu P_{16}(t) + \lambda_u P_5(t)$

We express the transient probabilities obtained from the above differential equations as the following:

$$\sum_{i=1}^{16} P_i = 1,$$
$$P^T[D - I] = 0,$$

where P is the state probability vector, D is the transition probability matrix and I is the identity matrix.

$$Q = \begin{bmatrix} 1-\lambda_t-\lambda_s-\lambda_e-\lambda_m-\lambda_u & \lambda_t & \lambda_s & \lambda_e & \lambda_m & 0 & 0 & 0 & 0 & 0 & 0 & \lambda_u \\ \mu & 1-\lambda_s \end{bmatrix}$$

$$-\lambda_e-\lambda_m-\lambda_u-\mu 000\lambda_s\lambda_e\lambda_m 0000\mu 01-\lambda_t-\lambda_e-\lambda_m-\lambda_u-\mu 00\lambda_t 00\lambda_e\lambda_m 00\mu 001-\lambda_t-\lambda_s-\lambda_m-\lambda_u-\mu 00\lambda_t 0\lambda_s 0\lambda_m 0\mu 0001-\lambda_t-\lambda_s$$
$$-\lambda_e-\lambda_u-\mu 00\lambda_t 0\lambda_s\lambda_e 00\mu\mu 001-2\mu 0000000\mu 0\mu 001-2\mu 000000\mu 00\mu 001-2\mu 000000\mu\mu 00001-2\mu 00000\mu 0\mu 00001$$

$$-2\mu 00000\mu\mu 000001-2\mu 0 \begin{matrix} \mu\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1-\mu \\ 0\ \mu\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ \mu\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ \mu\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\ 0\ 0\ 0\ 0\ \mu\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{matrix} \begin{matrix} 0 & 0 & 0 & 0 \\ \lambda_u & 0 & 0 & 0 \\ 0 & \lambda_u & 0 & 0 \\ 0 & 0 & \lambda_u & 0 \\ 0 & 0 & 0 & \lambda_u \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1-\mu & 0 & 0 & 0 \\ 0 & 1-\mu & 0 & 0 \\ 0 & 0 & 1-\mu & 0 \\ 0 & 0 & 0 & 1-\mu \end{matrix} \Bigg]$$

To estimate the failure probability of the components, we use the same data that was used during the system simulation. This ensures that the test scenarios used in the proposed method are the same as the test scenarios used for obtaining state transition probabilities of

the CTMC model. The first row of Table 6 represents the total number of failures of throttle sensor, speed sensor, EGO sensor, MAP sensor, and microcontroller during entire 100000 simulation iterations. We estimate the failure probability of each component by the ratio of total number of failures to the total number of simulation iterations. The estimated failure probability of the components is represented in the second row of Table 5.

The reachability graph is constituted by two distinct sets of system states: a) set of states (m_A) represents the working condition and b) set of states ($m_A^{'}$) represents the failure. If the AFRCS is considered as a non-repairable system ($\mu = 0$), then it's transient reliability can be defined as $R(t) = e^{-\sum_{i=1}^{t}\lambda_i t}$ where λ_i is the firing rate of transition whose firing causes the target networks to leave the reliable states. In this reliability expression if t tends to infinity $R(\infty) = 0$. The transient reliability plot of the fuel rate controller is given in Fig. 13.

If we consider AFRCS as a repairable system then the transient availability is $A(t) = \sum_i b_i P_i M_i \in m_A$ and b_i is the coefficient. The repair probability of throttle sensor, speed sensor, EGO sensor, MAP sensor, and microcontroller is assumed as same ($\mu = 0.002$). Then, solving the above first-order linear differential equations, we get steady-state probabilities of the working states (m_A) as given in Table 7. The sum of steady-state probabilities of the working states (see Table 7) gives steady-state availability of the system $Ava = \lim_{t\to\infty} A(t)$. Fig. 14 represent the steady-state availability of the system with respect to time. Finally, we observe the obtained steady-state availability of the fuel rate controller using the CTMC model (0.95871), which is quite similar to the corresponding value (0.94871) achieved using the proposed simulation-based method.

Table 8 presents a comparison of the results (transient reliability and steady-state availability) of the case study obtained by applying the proposed method and the CTMC model. The deviation between CTMC model-based and proposed model-based transient reliability prediction is 0.058168. On the other hand, the deviation between CTMC model-based and proposed model-based steady-state availability prediction is 0.01. So, the predicted values are quite similar for CTMC model and proposed method. The slight differences are appearing as CTMC model has not considered hardware-software interaction failures, whereas we have considered them.

6. Conclusions

An RTES development team may not have access to the actual hardware components during early testing of the system. It becomes a challenging task for the developers to test the hardware-software interaction failures and turns into a bottleneck in system reliability and availability prediction. The proposed method enables the RTES development team to evaluate the system's reliability and availability before the integration of the hardware-software parts. It replaces the actual hardware components with an environment model. The proposed method refers to the SysRS to identify the hardware components of the system. Subsequently, it also identifies the operational modes of the hardware component, referring to some failure data-source or standard reliability handbook. Based on the collected information, an environment model of the system is developed. Then, a digital simulator is used to sample the operation modes of the hardware components. Based on the operational modes, random test cases are generated for the testing of the embedded software. Finally, the reliability and availability of the system are evaluated based on the test results. Presently, there is a lack of quantitative relaibility and availability prediction method that have considered all types of hardware-software interaction failure. In our method, we have considered all possible types of interaction failures, along with hardware-specific and software-specific failures. The proposed method also can significantly reduce the production cost. For example, if the system does not meet the target reliability at the time of the post-integration testing due to software fault, it incurs considerable costs. We have demonstrated the application of the proposed reliability and availability method using a case study of an aircraft fuel rate controller. We have also validated our method comparing with a standard approach. However, the predicted reliability and availability value may not accurately match with

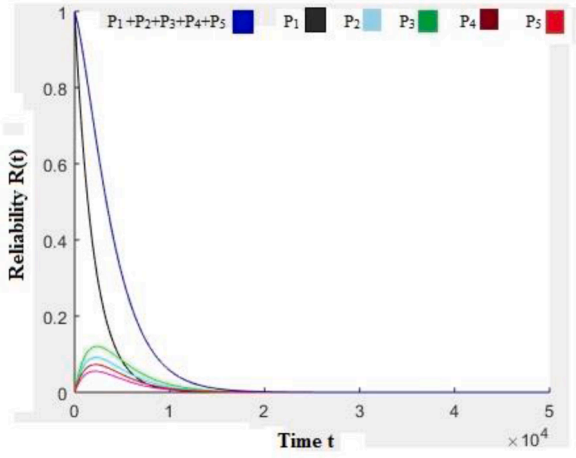


Fig. 13. Graphical representation of the steady-state reliability using CTMC model.

Table 7
Steady-state availability.

States	Steady-state probability
P1	0.93641
P2	0.00046
P3	0.00038
P4	0.00029
P5	0.02117
Steady-state availability	0.95871

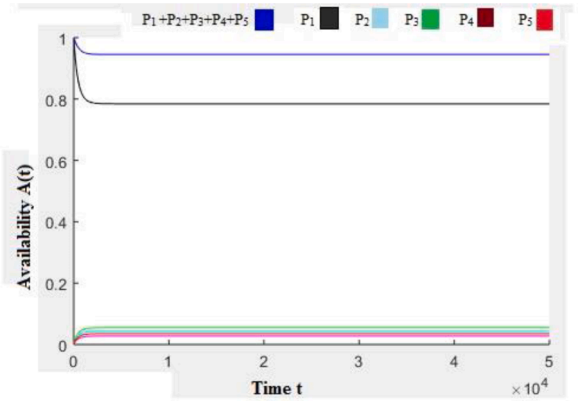


Fig. 14. Graphical representation of the availability using CTMC model.

Table 8
Comparison of transient reliability and steady-state availability of CTMC model and proposed method.

Models/ Methods	Transient Reliability R(t=100)	Steady-state Availability
CTMC Model [23]	0.505099 (system failure rate $\lambda_{sys} = 0.00683$)	0.95871
Proposed Method	0.563268 (system failure rate $\lambda_{sys} = 0.00574$)	0.94871
Deviation	0.058168	0.01

estimated reliability and availability values obtained through post-HW-SW integration system testing. The reason is that the proposed method uses the environment model of the hardware instead of the actual hardware. Nevertheless, it can be reasonably used to predict the reliability and availability of the RTES system when the development team does not have access to the hardware components.

Acknowledgments

We thank the Indian Institute of Technology *(IIT) Kharagpur*, India, for funding this research. We gratefully acknowledge the support of all the research scholars, faculty members, and staffs of the *Subir Chowdhury School of Quality and Reliability, IIT Kharagpur*, India. We also thank the anonymous reviewers for their constructive suggestions that helped us to enrich the manuscript significantly.

References

[1] MZ Iqbal, A Arcuri, L. Briand, Environment modeling and simulation for automated testing of soft real-time embedded software, Softw. Syst. Model. 14 (2015) 483–524.
[2] RK Iyer, P. Velardi, Hardware-related software errors: measurement and analysis, IEEE Trans. Softw. Eng. (1985) 223–231.
[3] S. Mittal, A survey on modeling and improving reliability of DNN algorithms and accelerators, J. Syst. Archit. (2019), 101689.
[4] A Avizienis, J-C Laprie, B Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secure Comput. 1 (2004) 11–33.
[5] X Teng, H Pham, DR. Jeske, Reliability modeling of hardware and software interactions, and its applications, IEEE Trans. Reliab. 55 (2006) 571–577.
[6] B Broekman, E. Notenboom, Testing Embedded Software, Pearson Education, 2003.
[7] PM Kruse, J Wegener, S. Wappler, A highly configurable test system for evolutionary black-box testing of embedded systems, in: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, ACM, 2009, pp. 1545–1552.
[8] S Sinha, NK Goyal, R. Mall, Early prediction of reliability and availability of combined hardware-software systems based on functional failures, J. Syst. Archit. 92 (2019) 23–38.

[9] S Sinha, NK Goyal, R. Mall, Survey of combined hardware–software reliability prediction approaches from architectural and system failure viewpoint, Int. J. Syst. Assurance Eng. Manage. (2019) 1–22.
[10] A Costes, C Landrault, J-C. Laprie, Reliability and availability models for maintained systems featuring hardware failures and design faults, IEEE Trans. Comput. (1978) 548–560.
[11] DS Roy, C Murthy, DK. Mohanta, Reliability analysis of phasor measurement unit incorporating hardware and software interaction failures, IET Gen. Transmission Distrib. 9 (2014) 164–171.
[12] U Sumita, Y. Masuda, Analysis of software availability/reliability under the influence of hardware failures, IEEE Trans. Softw. Eng. (1986) 32–41.
[13] X Diao, Y Zhao, M Pietrykowski, Z Wang, S Bragg-Sitton, C. Smidts, Fault propagation and effects analysis for designing an online monitoring system for the secondary loop of the nuclear power plant portion of a hybrid energy system, Nucl. Technol. 202 (2018) 106–123.
[14] DC Jensen, IY Tumer, T. Kurtoglu, Modeling the propagation of failures in software driven hardware systems to enable risk-informed design, ASME Int. Mech. Eng. Congress Exposition (2008) 283–293.
[15] S Sierla, I Tumer, N Papakonstantinou, K Koskinen, D. Jensen, Early integration of safety to the mechatronic system design process by the functional failure identification and propagation framework, Mechatronics 22 (2012) 137–151.
[16] I Tumer, C. Smidts, Integrated design-stage failure analysis of software-driven hardware systems, IEEE Trans. Comput. 60 (2010) 1072–1084.
[17] MZ Iqbal, A Arcuri, L. Briand, Environment modeling with UML/MARTE to support black-box system testing for real-time embedded systems: methodology and industrial case studies, in: International Conference on Model Driven Engineering Languages and Systems, Springer, 2010, pp. 286–300.
[18] S Jeong, Y Kwak, WJ. Lee, Software-in-the-loop simulation for early-stage testing of autosar software component, in: 2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2016, pp. 59–63.
[19] S Jeong, WJ. Lee, An automated testing method for AUTOSAR software components based on SiL simulation, in: 2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN), IEEE, 2017, pp. 278–283.
[20] S Montenegro, S Jähnichen, O. Maibaum, Simulation-based testing of embedded software in space applications. Embedded Systems–Modeling, Technology, and Applications, Springer, 2006, pp. 73–82.
[21] H Shokry, M. Hinchey, Model-based Verification of Embedded Software, IEEE Computer, 2009.
[22] S Werner, L Masing, F Lesniak, J. Becker, Software-in-the-loop simulation of embedded control applications based on virtual platforms, in: 2015 25th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2015, pp. 1–8.
[23] R Zeng, Y Jiang, C Lin, X. Shen, Dependability analysis of control center networks in smart grid using stochastic petri nets, IEEE Trans. Parallel Distrib. Syst. 23 (2012) 1721–1730.
[24] M Auguston, JB Michael, M-T. Shing, Environment behavior models for automation of testing and assessment of system safety, Inf. Softw. Technol. 48 (2006) 971–980.
[25] A David, KG Larsen, S Li, B. Nielsen, Timed testing under partial observability, in: 2009 International Conference on Software Testing Verification and Validation, IEEE, 2009, pp. 61–70.
[26] F Deng, F. Gao, Design of high confidence embedded software hardware-in-loop simulation test platform based on hierarchical model, in: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2018, pp. 163–168.
[27] A Hessel, KG Larsen, M Mikucionis, B Nielsen, P Pettersson, A. Skou, Testing Real-Time Systems Using UPPAAL. Formal Methods and Testing, Springer, 2008, pp. 77–117.
[28] M Krichen, S. Tripakis, Conformance testing for real-time systems, Formal Methods Syst. Des. 34 (2009) 238–304.
[29] KG Larsen, M Mikucionis, B. Nielsen, Online testing of real-time systems using uppaal. International Workshop on Formal Approaches to Software Testing, Springer, 2004, pp. 79–94.
[30] KG Larsen, M Mikucionis, B Nielsen, A. Skou, Testing real-time embedded software using UPPAAL-TRON: an industrial case study, in: Proceedings of the 5th ACM international conference on Embedded software, ACM, 2005, pp. 299–306.
[31] F Lindlar, A Windisch, J. Wegener, Integrating model-based testing with evolutionary functional testing, in: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, IEEE, 2010, pp. 163–172.
[32] G Silano, L. Iannelli, CrazyS: a software-in-the-loop simulation platform for the Crazyflie 2.0 nano-quadcopter. Robot Operating System (ROS), Springer, 2020, pp. 81–115.
[33] N Adjir, P De Saqui-Sannes, KM. Rahmouni, Testing Real-Time Systems Using TINA. Testing of Software and Communication Systems, Springer, 2009, pp. 1–15.
[34] NT Binh, TC Duy, I. Parissis, LusRegTes: a regression testing tool for lustre programs, Int. J. Electr. Comput. Eng. (2088–8708) (2017) 7.
[35] L Du Bousquet, F Ouabdesselam, J-L Richier, N Zuanon, Lutess: a specification-driven testing environment for synchronous software, in: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat No 99CB37002), IEEE, 1999, pp. 267–276.
[36] D Hatebur, M Heisel, T Santen, D. Seifert, Testing against requirements using UML environment models. Fachgruppentreffen requirements engineering und test, Analyse Verifikation (2008) 28–31.
[37] J Peleska, A Honisch, F Lapschies, H Löding, H Schmid, P Smuda, et al., A real-world benchmark model for testing concurrent real-time systems in the automotive domain, in: IFIP International Conference on Testing Software and Systems, Springer, 2011, pp. 146–161.
[38] J Peleska, E Vorobev, F Lapschies, C. Zahlten, Automated Model-based Testing with RT-Tester, University of Bremen., 2011.
[39] T Yu, W Srisa-an, G. Rothermel, SimRT: an automated framework to support regression testing for data races, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 48–59.
[40] AK Trivedi, ML. Shooman, A Markov Model for the Evaluation of Computer Software Performance, Polytechnic Institute of New York, Department of Electrical Engineering and Electrophysics, 1974.
[41] Y Zeng, L Xing, Q Zhang, X. Jia, An analytical method for reliability analysis of hardware-software co-design system, Qual. Reliab. Eng. Int. 35 (2019) 165–178.
[42] DC Jensen, IY Tumer, T. Kurtoglu, Modeling the propagation of failures in software driven hardware systems to enable risk-informed design, in: ASME 2008 International Mechanical Engineering Congress and Exposition, American Society of Mechanical Engineers, 2008, pp. 283–293.
[43] I Tumer, C. Smidts, Integrated design-stage failure analysis of software-driven hardware systems, IEEE Trans. Comput. 60 (2011) 1072–1084.
[44] N Papakonstantinou, S Sierla, IY Tumer, DC. Jensen, Using fault propagation analyses for early elimination of unreliable design alternatives of complex cyber-physical systems, in: ASME 2012 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, American Society of Mechanical Engineers, 2012, pp. 1183–1191.
[45] N Papakonstantinou, S Proper, B O'Halloran, IY. Tumer, A plant-wide and function-specific hierarchical functional fault detection and identification (HFFDI) system for multiple fault scenarios on complex systems, in: IDETCCIE Conference American Society of Mechanical Engineers, 2015, p. V01BT2A039.
[46] X Diao, Y Zhao, M Pietrykowski, Z Wang, S Bragg-Sitton, C. Smidts, Fault propagation and effects analysis for designing an online monitoring system for the secondary loop of the nuclear power plant portion of a hybrid energy system, Nucl. Technol. (2018) 1–18.
[47] C Mutha, D Jensen, I Tumer, C. Smidts, An integrated multidomain functional failure and propagation analysis approach for safe system design, AI EDAM 27 (2013) 317–347.
[48] L Irshad, HO Demirel, IY. Tumer, Automated generation of fault scenarios to assess potential human errors and functional failures in early design stages, J. Comput. Inf. Sci. Eng. (2020) 20.
[49] MathWorks. MATLAB/ simulink examples. 2011.
[50] NG Leveson, SS Cha, JC Knight, TJ. Shimeall, The use of self checks and voting in software error detection: an empirical study, IEEE Trans. Softw. Eng. 16 (1990) 432–443.
[51] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995.
[52] Bieth P, Brindejonc V. COTS-AEH – Use of complex COTS (Commercial-Off-The-Shelf) in airborne electronic hardware – failure mode and mitigation. Research Project EASA.2012/04 ed2012.
[53] Borgovini R, Pemberton S, Rossi M. Failure mode, effects, and criticality analysis (FMECA). Reliability Analysis Center Griffiss AFB NY; 1993.

[54] Mutuel LH. Commercial off-the-shelf airborne electronic hardware assurance methods—phase 3—embedded controllers. 2017.
[55] AW Brown, M Delbaere, P Eeles, S Johnston, R. Weaver, Realizing service-oriented solutions with the IBM rational software development platform, IBM Syst. J. 44 (2005) 727–752.
[56] D Leroux, M Nally, K. Hussey, Rational Software Architect: a tool for domain-specific modeling, IBM Syst. J. 45 (2006) 555–568.
[57] NSW. Center, Handbook of Reliability Prediction Procedures for Mechanical Equipment: Carderock Division, Naval Surface Warfare Center, 1998.
[58] J. Oldevik, MOFScript user guide version 0.9 (MOFScript v 1.4. 0)., in: The Eclipse Foundation Editorial, Ottawa, Ontario, 2011.
[59] D Gil, J Gracia, JC Baraza, PJ. Gil, Impact of faults in combinational logic of commercial microcontrollers, in: European Dependable Computing Conference, Springer, 2005, pp. 379–390.
[60] SR Welke, BW Johnson, JH. Aylor, Reliability modeling of hardware/software systems, IEEE Trans. Reliab. 44 (1995) 413–418.
[61] D Jensen, IY Tumer, T. Kurtoglu, Flow State Logic (FSL) for analysis of failure propagation in early design, in: ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, American Society of Mechanical Engineers, 2009, pp. 1033–1043.
[62] B Huang, M Rodriguez, M Li, JB Bernstein, CS. Smidts, Hardware error likelihood induced by the operation of software, IEEE Trans. Reliab. 60 (2011) 622–639.