Software Quality Journal (2026) 34:55
https://doi.org/10.1007/s11219-026-09783-2

RESEARCH

# Optimizing model-based generated tests for safety-critical embedded software

**Muhammad Nouman Zafar[1] · Wasif Afzal[1] · Eduard Paul Enoiu[1] · Zulqarnain Haider[2] · Nedim Zaimovic[2] · Iram Arshad[3]**

Received: 7 February 2026 / Accepted: 6 August 2026
© The Author(s) 2026

**Abstract**
The test suites generated through model-based testing (MBT) often include redundant test cases that neither enhance coverage nor fault detection but instead reduce test execution efficiency. Such test cases should be discarded to minimize the test suite size and its effect on execution cost while preserving fault detection capabilities. In this paper, we present a test suite optimization approach, implemented in a tool, that leverages symbolic classification, a greedy algorithm, and a similarity measure. It reduces MBT-generated test suites for embedded software by identifying and eliminating redundancy while minimizing its impact on the fault detection rate. We comparatively evaluate the optimized test suites against the original MBT-generated and also the manually created test suites, focusing on fault detection effectiveness and test execution efficiency. We also examine the robustness of proposed approach using two substantially different industrial case studies from Alstom Rail AB, Sweden. Results showed a significant reduction of over 80% in test suite size with a minimal effect on fault detection effectiveness. We further find that the test execution time of optimized test suites is equivalent to that of manually created ones, while achieving a fault detection rate ranging from 87% to 100%. Hence, results indicate the effectiveness and robustness of the proposed approach despite different redundancy sources (i.e., behavioral and structural) across the considered systems during the MBT test generation process.

**Keywords** Model-based Testing · System-level · Black-box Testing · Safety-critical Software · Optimization

## 1 Introduction

Model-based testing (MBT) (Utting and Legeard, 2010) is an automated testing technique that can generate test artifacts at early stages of the development life cycle. It uses an explicit model, such as a Finite State Machine (FSM), representing the concise behavior of the Software Under Test (SUT) to generate abstract test cases by traversing the model. These abstract test cases can then be automatically transformed into concrete test scripts that get

Extended author information available on the last page of the article

executed on the SUT to produce test verdicts. MBT uses different coverage criteria and generation algorithms for model exploration to generate an adequate test suite to detect faults. The generated test suite contains a wide range of inputs, expected outputs, and interactions within the system, representing scenarios that cover a substantial part of the implemented code. However, they also include redundant test cases that do not add value in revealing new faults (Yoo and Harman, 2012) and are also considered as one of the major obstacles in employing MBT in industrial practices (Kalaee and Rafe, 2019). The redundant test cases can be categorized as identical and non-identical test cases (Yoo and Harman, 2012). Identical test cases are replicas of each other, consisting of similar test inputs/outputs and timing constraints. In contrast, non-identical test cases contain different test inputs/outputs and onstraints but cover similar system functionality scenarios. Both types of redundancy collectively affect the scalability of a test suite. It poses a substantial challenge, particularly in industrial settings where cost and resource efficiency are key. Hence, identifying and discarding such test cases is essential to enhance test efficiency in dynamic development environments. Previous attempts to address the aforementioned challenge through various techniques have often resulted in a trade-off, a decrease in the fault detection rate (Zhong et al., 2008) (Cruciani et al., 2019; Rothermel et al., 2002). Moreover, this problem becomes particularly challenging at the black box system-level testing stage, where the source code of the SUT is inaccessible, and artifacts such as precise coverage data are unavailable.

The test suite generated from a model, such as an FSM, lacks information about fault detection effectiveness or coverage of implemented code before execution; therefore, a substitute measure is required to quantify the effectiveness and necessity of each test case contributing towards the overall test goal. Several studies in the literature (e.g., Nikolik (2006); Cai and Lyu (2005); Kochhar et al. (2015), etc.) have demonstrated a moderate to strong correlation between diversity, coverage, and fault detection effectiveness. Diversity in a test suite ensures a wide range of test scenarios are explored, while coverage ensures a significant part of a system is tested (Nuh et al., 2023). Together, these can increase the fault detection effectiveness of a test suite, ensuring a robust and reliable software system (Mondal et al., 2015). Given the importance of these factors, a tool is needed to facilitate the design of an effectively reduced MBT-generated test suite with high diversity and coverage while maximizing the fault detection effectiveness. Hence, in this work, we have proposed, implemented, and evaluated a new tool, TIGER+[1](Model-based Test scrIpt GenEration fRamework+). TIGER+ reduces the size of an MBT-generated test suite by identifying and eliminating redundant test cases based on two key metrics: Modified Condition Decision Coverage (MCDC) and a similarity measure inspired by the Jaccard index (Jaccard, 1912) It also integrates data extraction, pre-processing, and symbolic classification of the test data. The key features of our proposed approach include the creation of a requirement traceability matrix using rule-based classification, ensuring adequate coverage of each requirement, and application of MCDC on the guard conditions of an FSM model to remove redundancy alongside a similarity measure to remove duplicate and similar test cases within a test suite while preserving fault detection effectiveness. We also conducted a comparative evaluation of these generated subsets with the original MBT-generated test suite and a test suite created manually by an industrial tester in terms of effectiveness and execution efficiency. A prior evaluation of the proposed approach (Zafar et al., 2024) had a focus on a single

[1] TIGER+ is available at https://github.com/MuhammadNoumanZafar/TIGERPlus

system, leaving an open question about its robustness across different systems, especially when test redundancy in MBT can arise from different sources (i.e., model exploration techniques, behavioral and structural complexity of the MBT model representing the system under test). Thus, in this paper, TIGER+ is evaluated on two different case studies of the Train Control Management System (TCMS) from Alstom Rail AB, Sweden, with respect to existing manual industrial practices as baseline. The SUTs differ substantially in terms of parametrization, model size, and behavioral and structural complexity, enabling investigation under real-world constraints. Through cross-case analysis, we investigate the potential benefits and limitations of the proposed approach in industry, as well as the characteristics of the system that may affect its robustness. We also introduced the algorithms employed in the proposed approach to facilitate its adoption and replication by other researchers and practitioners. Finally, we offered a detailed discussion of the implications of our research findings in a broader context, highlighting future directions. Based on our objectives, we have defined the following research question:

- RQ. To what extent does the proposed approach affect the MBT fault detection effectiveness, test suite size, and execution efficiency relative to manually created test suites across industrial case studies?

The results of the experimental evaluation show that:

- TIGER+ can effectively detect redundant test cases at the abstract level without requiring direct access to source code. It achieves a significant reduction in test suite size within the range of approx. 80% to 85% and 89% to 92% when optimized at the highest and lowest levels of the expression tree representing guard condition, respectively.
- TIGER+ improved the execution efficiency of the original MBT-generated test suites while preserving a higher fault detection rate within the range of 87% to 100% when compared with manually created test suites.
- Despite the differences between the SUTs and redundancy sources, TIGER+ delivered consistent results, reinforcing the effectiveness and robustness of the proposed approach.

The rest of the paper is organized as follows: Section 2 discusses the related concepts and studies, Section 3 presents a detailed description of the proposed approach, Section 4 covers its experimental evaluation, including a description of the SUTs, development and execution of test suites, experimental setup, and result. Section 5 presents a discussion on the results, and threats to the validity of this study are discussed in Section 6, followed by a conclusion and future work in Section 7.

## 2 Background & related work

This section briefly introduces safety-critical software, its development and system-level testing, data pre-processing and symbolic classification, test suite optimization, and mutation analysis. It also summarizes related work concerning the optimization of MBT-generated test suites.

### 2.1 Safety-critical software and its development

Safety-critical software is crucial in controlling and monitoring complex functions in embedded systems whose failure or malfunction can lead to catastrophic events. It also ensures the reliability and integrity of all the operations to mitigate the risks associated with their failure. Such software is usually implemented for specially designed computers known as Programmable Logical Controllers (PLCs), which are connected with sensors and actuators via various networks and are responsible for realizing safety-critical tasks. The development of safety-critical software follows certain industry-specific guidelines and adheres to regulatory standards, ensuring the reliability of the system. It starts with a rigorous and thorough analysis of the requirements for the quantification of safety based on certain qualitative and quantitative analysis techniques such as Markov analysis, fault tree analysis, Safety Integrity Level (SIL) assessment, etc. Rouvroye and Bliek (2002), followed by the design, implementation, and testing phases.

There exists a set of programming languages developed and published by the International Electrotechnical Commission (IEC) (Maslar, 1996) for the implementation of safety-critical applications for PLCs (e.g., Ladder Diagram (LD), Functional Block Diagram (FBD), Structural Text (ST), etc.). These languages have been widely adopted across various domains where PLC programming is prevalent for automation. However, in our context, FBD serves as a fundamental language at Alstom to develop the systems under consideration. FBD provides a graphical visualization of control logic known as a function block. These function blocks are divided among ten distinct functional groups (e.g., logic, arithmetic, timer, etc.) as stated in IEC 1131-3 (Maslar, 1996). Each function block in a group encapsulates a predefined functionality and can be integrated with other function blocks through a set of input and output terminals to create complex logical algorithms based on defined functional and non-functional requirements. An example of a FBD program is presented in Fig. 1, consisting of six blocks representing three logical operators (i.e., AND, OR, NOT) and one timer operator (i.e., TON). It defines the functional behavior of a program that requires four input parameters (i.e., I1, I2, I3, and I4) and provides one output (O1) within a time constraint of 3 seconds.

### 2.2 System level testing of safety-critical software

Testing safety-critical software is iterative, encompassing unit, integration, and system levels. The complexity and effort required to design and conduct testing activities increase

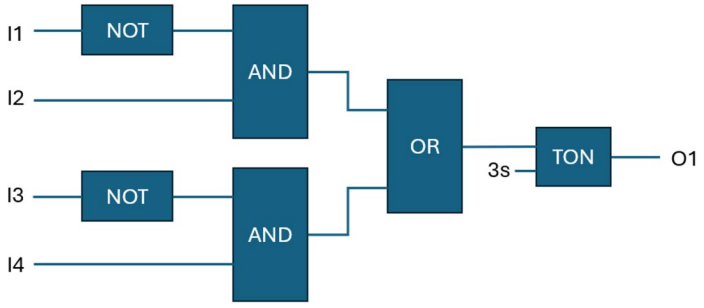


**Fig. 1** An example FBD program

with the increase in each testing level. For instance, testing at the unit and integration level focuses on small segments of the system, thus limiting the test objectives, complexity, and input/output space. Whereas, at the system level, the testing complexity, test objectives, and input/output space expand as the system scope increases with the number of integrated modules (Zafar, 2022).

System-level testing in industry is a systematic process to verify and validate the functionality of safety-critical software at an abstract level without delving into the internal structure or implementation details. Test cases are designed based on functional and non-functional requirements of the system, which cover different operational scenarios and boundary values of the system. After the execution of test cases, test engineers analyze the test verdicts to evaluate the system's behavior and its adherence to safety regulations and standards. Different manual and automated testing techniques exist that can be used to validate software at the system level, e.g., equivalence class partitioning (Homès, 2013), boundary value analysis (Homès, 2013), combinatorial testing (Kuhn et al., 2013), mutation testing (Woodward, 1993), and random testing (Hamlet, 1994).

### 2.3 Test suite optimization

Test suite optimization aims to enhance effectiveness and efficiency while reducing resource utilization (Gupta et al., 2019). It includes various techniques (i.e., selection, minimization, or prioritization of test cases in a test suite) to achieve specific test objectives (e.g., maximizing test coverage, enhancing fault detection capabilities, minimizing execution time, etc.). Test case selection deals with choosing a subset of test cases based on specific coverage criteria or objectives. Minimization involves the reduction of test suite size or complexity while preserving fault detection capabilities. Prioritization focuses on the sorting of test cases based on coverage, fault detection capability, or critical features. All the techniques mentioned above require a thorough understanding of the system, test data, test objectives, and adequacy criteria and can complement each other to improve the testing process and quality of a test suite.

### 2.4 Data pre-processing and symbolic classification

Data pre-processing, symbolic classification, and test suite optimization are three different concepts, however, in our proposed approach, we connected them in a sequential and interdependent flow to make an informed, explainable, and data-driven decision to reduce the size of the model-generated test suite.

The data pre-processing prepares the large volume of generated data for analysis and classification. It includes several steps (i.e., data transformation, cleaning, and imputation) to ensure consistency, completeness, and quality of the data (Malley et al., 2016). Data transformation deals with the conversion of all data into a consistent and suitable format, cleaning reduces or eliminates noise from the data by removing duplicate or identical values, and data imputation completes the data by adding missing values based on a defined criterion.

Symbolic classification (Mali and Mitra, 2005) refers to the process of assigning categories to data instances based on domain knowledge represented as human-interpretable symbols such as logical statements, rules, constraints, etc. Such classification is especially used

where reproducibility and transparency are critical. A common form of symbolic classification is rule-based classification (Li and Liu, 2014), where data is categorized into distinct groups based on a list of predefined rules or conditions while balancing accuracy and simplicity. Each rule represents a logical statement, natural knowledge, or information that can create relationships between attributes to facilitate the accurate distribution of data among different groups. These rules can be derived through a thorough analysis of the available attributes in the data, specification, or domain expertise. These rules are typically written in an *IF-THEN* format where *IF* (known as the antecedent) specifies the condition and *THEN* (known as the consequent) provides the prediction or decision to classify the data.

### 2.5 Mutation analysis

Mutation analysis (Acree et al., 1979) is an evaluation technique used to assess and analyze the fault detection capabilities of a test suite. It involves the creation of different versions of a real system by inducing a small fault in each version, known as a mutant. Each mutant represents an error in the programming language (i.e., syntactical or logical) caused by a developer due to an honest mistake or misinterpretation of requirements during the development of the system. These mutants can be categorized into three groups, i.e., equivalent, non-equivalent, and stubborn mutants (Yao et al., 2014). Equivalent mutants exhibit identical behavior to the original programs and cannot be killed by any test case in a test suite. In contrast, stubborn and non-equivalent mutants exhibit different behavior than the original programs and can be killed by a corresponding test case in a test suite; however, stubborn mutants are a special type of mutants that are hard to kill and require special or additional test cases. A mutant is said to be killed if the test verdict of a test suite executed on the original program and the mutated version shows a contradiction; otherwise, it is considered alive. After executing a test suite on each mutant, a mutation score can be calculated either based on an output-only oracle where a fault in a program must affect the outcome and be detected by a test case (i.e., strong mutation) or an internal oracle in which fault in a program does not affect or propagate to the final outcome (i.e., weak mutation). A mutation score is a ratio between the number of mutants killed by a test suite and the total number of mutants, representing its fault detection rate.

### 2.6 Related work

There are various studies (i.e., Fraser and Wotawa (2007); Deneke et al. (2022); Liu and Yu (2022); Hemmati et al. (2013); Coutinho et al. (2016); Pan et al. (2023); Shu et al. (2024); Pan et al. (2024); Cruciani et al. (2019); Arefin et al. (2018); Koitz-Hristov et al. (2022); Kushik et al. (2021, 2023); Sachtleben and Peleska (2022); Kalaee and Rafe (2019)) that have proposed different methods in various contexts to generate an adequate sized test suite. To systematically discuss the existing methods, we classify them in three categories (i.e., Redundancy-based, Similarity-based, and Coverage-based). Additionally, Table 1 provides a summary comparison of these methods, including TIGER+.

### 2.6.1 Redundancy-based techniques

Fraser and Wotawa (2007) proposed an optimization approach to reduce the size of the test suites generated through model checker-based techniques. The proposed approach transforms the test cases by sequentially identifying and removing the redundant parts within the test cases. The experimental evaluation shows the improved quality of test cases while having minimal effect on fault detection effectiveness. Deneke et al. (2022) addressed regression testing challenges arising from increasing test suite size and executing exhaustive testing within a given time and becoming infeasible. Recognizing that test suite optimization is an NP-completeness problem without a singular solution, the authors proposed practical Swarm Optimization (PSO) for removing the redundant test cases. The authors compared the proposed technique with Greedy Weighted Set Cover (G WSC) algorithm, Greedy (G) algorithm, Harrold-Gupta-Soffa (HGS) algorithm and Greedy Redundant Essential (GRE) algorithm based on the size of reduced test set and execution time. The results showed that PSO resulted in 55.55% improvement in minimization, more than shown by G WSC, G, HGS and GRE. PSO also resulted in less execution cost when compared with these techniques. To improve the automated test case generating tool Randoop, Liu and Yu (2022) proposed Randoop-TSR (Test Suite Reduction) that follows three strategies. 1) similarity-based input sequence selection (i.e., by analyzing input sequences for similarities, redundant sequences are filtered out, ensuring a more concise test suite) 2) redundant and duplicate assert statements elimination based on test smell detection (i.e., utilizing test smell detection techniques, unnecessary assert statements are removed, enhancing test clarity and effectiveness) and 3) redundant test cases elimination without breaking test requirements such as code coverage and mutation score. The evaluation of Randoop-TSR demonstrated improvements in test suite size reduction, execution efficiency and fault detection capability.

### 2.6.2 Similarity-based techniques

Hemmati et al. (2013) introduced a family of similarity-based minimization techniques for test suites generated through state machines and reported a comparison with other similar techniques in the literature. They also investigated different parameters, such as encoding, similarity functions, and minimization algorithms to show their potential effect on fault detection capabilities. The similarity-based test reduction was found to be a more effective technique for producing a scalable test suite. Similarly, the application and thorough investigation of six distance functions for the reduction of MBT-generated test suite reduction are presented in Coutinho et al. (2016). The empirical results show that the choice of distance function has a minimal effect on the test suite size. However, it has shown significant effects on the fault coverage and stability of the test suite. Pan et al. (2023) proposed a test suite minimization approach by converting the source code of the program into an Abstract Syntax Tree (AST) and then utilizing four tree-based similarity measures to apply the genetic algorithm for test suite minimization. The results indicate the effectiveness of the proposed approach by achieving a high fault detection rate of 82% on average with the reduced test suite. Shu et al. (2024) focused on the limitation of relying solely on similarity-based metrics for test case prioritization, as this can hinder the universal effectiveness and performance optimization that may not always result in detecting early faults, thus leading to reduced efficiency. To enhance early fault detection performance, the authors proposed a Diver-

**Table 1** Summary table of representative test suite optimization techniques, including TIGER+

| Study | Approach | Experimental Evaluation | Results |
|---|---|---|---|
| Fraser and Wotawa (2007) | Optimization to reduce test suite size by removing redundant parts. | Evaluated test case quality and fault detection effectiveness. | Improved quality with minimal effect on fault detection under certain conditions, and with a drawback of runtime complexity. |
| Deneke et al. (2022) | PSO-based minimization to reduce redundant test cases. | Compared against benchmark techniques (GWSC, GRE, G, HGS) on test suite size and execution time. | PSO achieved 55.55% minimization and lowest execution cost while maintaining requirement coverage. |
| Liu and Yu (2022) | Randoop-TSR mechanism employing similarity-based redundancy elimination and test smell detection. | Evaluated test suite size reduction, execution time and fault detection capabilities. | Significantly reduced test suite size and execution time while maintaining fault detection effectiveness. |
| Hemmati et al. (2013) | Similarity-based minimization for state machine-generated test suites based on historical defect data. | Compared with other techniques; evaluated fault detection. | Effective in producing scalable test suites with a tradeoff in fault detection effectiveness. |
| Coutinho et al. (2016) | MBT-based test suite reduction. | Evaluated the impact of distinct functions on test suite size, fault coverage, and stability. | Minimal effect on size, significant on fault coverage and stability. |
| Pan et al. (2023) | Test suite minimization by converting source code to AST with a genetic algorithm. | Evaluated fault detection rate and test suite size reduction. | High fault detection rate of 82% on average with a test suite minimization rate of 50%. |
| Shu et al. (2024) | Diversity-driven Learn To Rank TCP using whale Optimization and Random Forest. | Evaluated Average Percentage Fault Detection (APFD) metric on benchmarks. | Achieved average APFD of 0.953, outperforming existing methods by 11.73%. |
| Pan et al. (2024) | Language model-based minimization using semantic embeddings and genetic algorithms. | Compared scalability, execution time, and fault detection against ATM and FAST-R. | Outperformed ATM and FAST-R, achieving higher fault detection and scalability; reduced test suites nearly 5x faster for large systems. |
| Cruciani et al. (2019) | FAST-R enhancement: Similarity-based scalable reduction leveraging big data algorithms and black-box test data. | Evaluated computational efficiency and fault detection on large-scale suites (500k test cases). | Selected optimal subsets under 10 seconds, achieving high scalability and effective fault detection without relying on traditional artifacts. |
| Arefin et al. (2018) | Improved MBT for safety-critical software using MCDC at the spec level and manual dependency analysis. | Evaluated code-level coverage, model-level constraint coverage, and search algorithm efficiency. | Improved constraint coverage to 65.1% and code coverage by 33.5%. Better fault detection at higher computational cost. |
| Koitz-Hristov et al. (2022) | Checked-coverage-based TSR ensuring code correctness via assertions. | Compared traditional code coverage TSR vs. check coverage TSR. | Improved balance between test suite size and fault detection effectiveness. |
| Kushik et al. (2021) | TSR from faulty FSM model with human-defined probabilities. | Evaluated detection of faulty implementations. | Effective in minimizing size while maintaining detection. |
| Kushik et al. (2023) | TSR by shortening input combination lengths. | Evaluated test suite size and fault detection. | Significant reduction in size with maintained/improved detection. |

**Table 1** (continued)

| Study | Approach | Experimental Evaluation | Results |
|---|---|---|---|
| Sachtleben and Peleska (2022) | Test generation and minimization for FSM-based models using grey-box information. | Evaluated test suite size based on reference model size. | Significant reduction depending on reference model size. |
| Kalaee and Rafe (2019) | Metaheuristic integration with model checker (GROOVE) to mitigate redundancy and state-space exploration. | Evaluated test coverage, execution speed, and memory usage on graph-based systems. | Improved coverage, reduced redundancy, and addressed state-space issues. |
| Our approach – TIGER+ | Pre-execution optimization using MCDC and similarity measures (Jaccard-based) without needing structural coverage metrics. | Evaluated fault detection rate, execution time, and test suite size reduction. | High fault detection (87–100%) with test suite reduction over 80% in compliance with safety-standards. |

sity-Driven-Learn-To-Rank model-based Test Case Prioritization (TCP) approach named as DLTCP. The proposed method first utilizes the Whale Optimization (WO) algorithm to search for an optimal set of similarity metrics from 15 existing pools of candidates. This results in determining the most suitable metric and test cases are prioritized accordingly. The selected test case ranking served as training data for DLTCP, and later, a machine learning random forest algorithm was used to train a ranking model for test case prioritization.n. To evaluate the effectiveness of the proposed DLTCP approach, the authors utilized the Average Percentage Fault Detection (APFD) metric. The results on seven classical benchmarks demonstrate that the proposed method achieves an average APFD value of 0.953, which is 11. 73% higher than the existing state-of-the-art algorithms. Addressing scalability concerns in black-box test suite minimization and improving the test efficiency by removing redundant test cases, Pan et al. (2024) introduce the Language model-based Test suite Minimization (LTM) method. The authors utilize five language models, CodeBERT, GraphCodeBERT, UniXcoder, StarEncoder, and COdeLlama, to encode test methods as embeddings, which enabled semantic-based similarity measurement. These semantics were used to compute two similarity measures, Cosine similarity and Euclidean distance. This results in guiding the genetic algorithm to search for an optimal minimized test suite, thus reducing the overall search time. The results demonstrated that LTM with UniXcoder Cosine similarity outperformed ATM (Abstract syntax trees based Test case Minimizer) by achieving slightly greater testing time savings, attaining a higher fault detection rate than FAST-R (family of similarity-based test prioritization - reduction), and minimizing test suites nearly five times faster on average for larger test suites and systems, thus achieving much higher scalability. Similarly, Cruciani et al. (2019) identified that traditional artifact-dependent (that is, code coverage and specification model) reduction methods are often too expensive to scale effectively for modern massive systems. The authors proposed effective scalable approaches for similarity-based test suites reduction that leverage big data algorithms. The approach selects test cases based on test similarity measures by requiring only the test case data itself, such as source code or command-line inputs, which makes the technique generalizable even when traditional artifacts are not available.

### 2.6.3 Coverage-based techniques

Arefin et al. (2018) explored the application of MCDC at the specification level to improve model-based testing of safety-critical systems. Instead of relying on the traditionally recommended "all transition coverage" criterion, they propose applying MCDC on specification-level constraints (e.g., UML state machine transition guards) to generate test suites and perform a manual dependency analysis to reduce the search space. In an empirical evaluation, they also demonstrate that their proposed approach improves source code "all condition/decision coverage" by 33.5% compared to the standard transition coverage test suite. It also shows the detection of three new faults and two obsolete legacy specifications, indicating its potential for enhancing structural testing of safety-critical software; however, it has a high computational cost to evaluate the fitness function for test case generation. Koitz-Hristov et al. (2022) address the problem in traditional Test Suite Reduction (TSR) on reducing the size of test suites while maintaining code coverage. However, code coverage does not guarantee fault detection effectiveness alone since it only ensures the execution without validating whether the tested code behaves correctly. This limitation leads the authors to incorporate checked coverage (i.e., coverage of code that is verified by assertions) as a criterion for TSR while maintaining fault detection capabilities (FDC). They conducted an empirical experiment by comparing traditional code coverage-based TSR with checked coverage-based TSR by applying different reduction techniques and measuring their impacts on fault detection effectiveness. The results reveal that incorporating checked coverage in TSR is worth the effort as it balances test suite size and fault detection efficiency better than traditional approaches. Kushik et al. (2021) proposed an approach to minimize the size of an exhaustive test suite derived from a faulty, non-deterministic FSM model. The proposed approach augments the human-defined probabilities to detect each faulty implementation. Moreover, another approach as an extension of this work is presented in Kushik et al. (2023) for test suite minimization by shortening the length of the combinations of inputs representing the checking sequence. Sachtleben and Peleska (2022) proposed a test generation algorithm for an FSM-based model to generate an adequately sized test suite using grey-box information about the system, i.e., information about enabled/disabled inputs. The results show a significant reduction in test suite generation depending on the size of the reference model. Kalaee and Rafe (2019) addressed redundant test cases and the state space explosion problems in Model-Based Testing (MBT). They proposed an approach that leverages metaheuristic algorithms to adapt the model checker for integration testing of systems formally specified by graph transformation. This method optimizes the exploration of the state space, reducing state space explosion while ensuring high test coverage. The proposed method utilizes the modeling and simulation features of the underlying model checker. A population of test suites is generated, where each test suite represents a set of paths in the state space. Using search-based techniques, the population evolves to satisfy all def-use test objectives, reducing redundant test cases while maximizing fault detection effectiveness. To evaluate the efficiency of their approach, the authors use GROOVE, an open-source toolset for model simulation and model checking of graph transformation systems. GROOVE plays a dual role, serving as both a model checker and a validation tool for generating and analyzing test cases. Empirical results confirm that the proposed approach significantly improves test coverage, execution speed, and memory efficiency compared to state-of-the-art techniques. The findings demonstrate that this method outperforms existing techniques by reducing

redundant test cases, handling state space explosion effectively, and ensuring scalable test generation.

**Research gap** *The inherent trade-off between test suite reduction and fault detection effectiveness is a significant issue identified in our examined papers. This issue becomes increasingly complex, especially for system-level black box testing of embedded software, where internal code details (e.g., code coverage) are inaccessible and information on fault detection is unavailable. Although recent efforts have attempted to address this trade-off, there remains a challenge to balance both aspects for the system-level black box testing in compliance with safety standards. Therefore, in this study, we optimize model-based generated test suites with high diversity while maintaining the fault detection effectiveness at the system level black-box testing of safety-critical embedded software.*

## 3 Proposed approach

The proposed approach is intended to provide a reduced but effective fault-detection subset of a test suite generated from an FSM model. Figure 2 illustrates an overview of each step in the proposed approach, consisting of the following five steps:

1. Model-based test generation
2. Data extraction
3. Pre-processing of abstract test cases
4. Data transformation
5. Removal of identical test cases

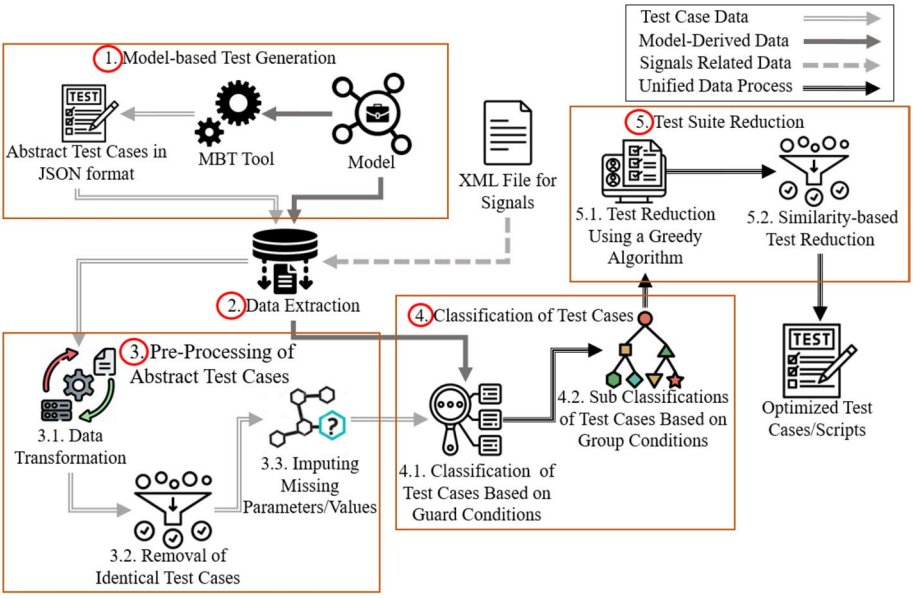


**Fig. 2** An automated symbolic classification-based test reduction approach

6. Imputing missing parameters/values

7. Classification of test cases

8. Classification of test cases based on Guard Conditions
9. Sub-classification of test cases based on Group Conditions

10. Test suite reduction

11. Test reduction using a greedy algorithm
12. Similarity-based test reduction

### 3.1 Model-based test generation

We used the open-source tool GraphWalker (GW)[2] to generate abstract test cases from an FSM model representing the SUT's behavior. Different interconnected sub-diagrams can represent an FSM model in GW. Each sub-diagram represents the behavior of a component in the SUT, consisting of distinct nodes that indicate the system's states and edges that depict transitions between these states when a specific event occurs. Each sub-diagram contains some special nodes known as shared nodes, defining the relation between one another. An FSM model also contains guard conditions to control the transitions between nodes. Guard conditions are Boolean expressions that construct precise functional and non-functional behavior of a model corresponding to a SUT. Moreover, an FSM model can also store information to map requirements to model elements (i.e., nodes and edges) supporting requirement traceability (Kramer and Legeard, 2016). The generation of an effective and complete test suite requires conformance of an FSM model to the requirements of the SUT (Utting and Legeard, 2010).

GW supports different generation algorithms (e.g., random, quick random, A star, etc.) and coverage criteria (edge coverage, vertex coverage, requirement coverage, etc.) to generate abstract test cases in the form of a JSON file, as shown in Listing 1. The abstract test cases generated through an FSM model conventionally represent the traversed paths instantiated with different event parameters (test data) encoded as strings. Similarly, the JSON file generated by GW contains these abstract test cases as a sequence of traversed model elements representing a test step in a test case. Each test case contains several pieces of information for each traversed element, such as the name of the model representing the behavior of a SUT as 'modelName', all parameters and their initial values involved in a model as 'data', parameters with their respective altered values after a transition or at a particular state representing inputs and expected outputs of a test step in the form of 'Actions', and information about a traversed element (i.e., currentElementName, currentElementID, and properties), etc.

[2] https://github.com/GraphWalker/graphwalker-project/wiki

```
{
    "modelName": "ReqInit",
    "data":[
        {"Prev": "true"},
        {"ReqAp": "true"},
        {"IsMan": "true"},
        {"ReqInit": "true"}
    ],
    "currentElementID": "1eacd4d5-3d6c-4624-92f4-65bc5e6d9a5c",
    "currentElementName": "Start",
    "properties":[
        {"x": 221.04553986268414},
        {"y": 326.0896089137364}
    ]
}{
     "modelName": "ReqInit",
    "data":[
        {"Prev": "true"},
        {"ReqAp": "true"},
        {"IsMan": "true"},
        {"ReqInit": "true"}
    ],
    "currentElementID": "19e19385-9a02-45f7-b780-162e3c53f7a5",
    "currentElementName": "Request_Not_Initiated",
    "actions": [{"Action": "ReqInit=false"}],
    "properties":[]
}
```

Listing 1: An example of an abstract test step generated by GW in JSON format

### 3.2 Data extraction

To optimize the abstract test cases, all relevant information from the JSON file, along with guard conditions and requirements linked to model elements from the FSM model, needs to be extracted. Furthermore, in our case, each parameter used in a model represents a logical name[3] of a signal interacting with different components of an embedded system. In order to generate concrete or executable test cases/scripts, these logical names require to be mapped with one or more technical signal names used by a real embedded system to perform its operation. The test scripts at Alstom consist of two main test steps (i.e., forcing input signals (using technical signal names) and verifying output signals) to validate an SUT. To achieve this goal, additional information is required, such as the type of each parameter (i.e., input and output), and their data types (e.g., int, Boolean, etc.). Hence, we have proposed a well-defined format for an XML file as shown in Listing 2 containing such information and extracted the required information from it into the corresponding logical data model (Zafar et al., 2021). The extraction process also ensures the preservation of continuity between the test cases of a generated test suite (i.e., the sequences of traversed elements with their corresponding actions) in the form of an array. We have defined two logical data models to store the extracted data (i.e., a logical data model similar to the format of a JSON file consisting of a 'RootObject' comprising 'modelName', 'currentElementID', 'currentElementName', an array of 'actions', and 'properties' as shown in Listing 3 and a logical data model consist-

[3] a representation of a signal in requirement specifications at Alstom

ing of guard conditions and a list of requirements associated with each guard condition, as shown in Listing 4). It is important to mention that an FSM model also contains information about each model element, such as the IDs, name, and sourceVertexIds, etc., of each edge, and is likely to introduce noise during optimization in our case. Therefore, to avoid noise in the data and minimize effort at the pre-processing step, we have selected and extracted only the required information (i.e., guards and list of requirements) from the model.

```
<Signals xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Signal>
        <Type>Input</Type>
        <DataType>bool</DataType>
        <LogicalName>global.variable1</LogicalName>
        <PrimaryTechnicalName>XYZ</PrimaryTechnicalName>
        <SecondaryTechnicalName>ACB</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
        <Secondary2TechnicalName> </Secondary2TechnicalName>
    </Signal>
    <Signal>
        <Type>Output</Type>
        <DataType>bool</DataType>
        <LogicalName>global.variableoutputx</LogicalName>
        <PrimaryTechnicalName>BCD</PrimaryTechnicalName>
        <SecondaryTechnicalName>BDC</SecondaryTechnicalName>
        <Primary2TechnicalName> </Primary2TechnicalName>
        <Secondary2TechnicalName> </Secondary2TechnicalName>
    </Signal>
</Signals>
```

Listing 2: Example of an XML file containing information about logical and technical names

```
public class Rootobject
{
    public string modelName { get; set; }
    public string currentElementID { get; set; }
    public string currentElementName { get; set; }
    public Actions[] actions { get; set; }
    public object[] properties { get; set; }
}

public class Actions
{
    public string Action { get; set; }
}
```

Listing 3: A logical data model for abstract test cases

```
class DataModelforFSMModel
{
    public string guard { get; set; }
    public List<string> requirements { get; set; }

}
```

Listing 4: A logical data model for guard conditions and associated requirements

### 3.3 Pre-processing of abstract test cases

The JSON file generated from the model contains abstract test data as well as information about a model and each traversed element such as ‘modelName’, ‘currentElementID’, ‘properties’, etc. which, in our case, can be considered as noise that dilutes the test cases with insignificant information. Moreover, to select MCDC-adequate test cases, we require instantiated test data. Hence, pre-processing of generated abstract test cases is required to extract relevant test data, transform it into a scenario-based test case format, and optimize the test. The pre-processing in the proposed approach consists of three main steps: data transformation, removal of identical test cases, and imputing the missing parameters and corresponding values in each test case.

#### 3.3.1 Data transformation

The test steps in each generated test case can contain more than one event parameter and its respective values encoded in a string. Each string from a test case needs to be identified and extracted to create inputs, expected outputs, and timing constraints for a scenario-based test suite. We have defined a generic logical data model consisting of a unique test identifier ‘TestcaseID’, a list of actions containing ‘ActionDescription’, ‘ActionVariable’, and ‘ActionValue’, a list of expected results comprising ‘ResultDescription’, ‘ResultVariable’, and ‘ResultValue’, and a timing constraint ‘Withintime’ to store the dynamic test data in the form of test cases in a test suite as shown in Listing 5. We have also defined a data transformation algorithm to convert model-generated test artifacts into structured test cases. The pseudocode of the employed algorithm is given as Algorithm 1. It splits the strings in the list of ’Actions’ to extract parameters and their values based on delimiters (i.e., ‘;’ and ‘=’) from each root object in a tree of traversed paths of the FSM model representing test cases (lines 6 and 8). It also finds the data type of each parameter from the list of signals extracted from the XML file (line 12) and dynamically populates the defined logical data model by creating test actions or expected results of a test suite (lines 16 to 22), as shown in Listing 6.

```
public class TestCaseModel
{
    public TestCaseModel() {
        testcases = new List<Testcase>();
    }
    public List<Testcase> testcases { get; set; }

}
public class Testcase {
    public Testcase() {

        action = new List<TestActions>();
        expectedresult = new List<ExpectedResult>();
    }

    public int TestcaseID { get; set; }
    public List<TestActions> action { get; set; }
    public List<ExpectedResult> expectedresult { get; set; }
    public string Withintime { get; set; }
}
```

Listing 5: A logical data model of a test suite in C Sharp (C#)

```
"testcases":[
    {
        "TestcaseID": 1,
        "action":[
            {
                "ActionDescription": "Request_Not_Initiated",
                "ActionVariable": "ReqInit",
                "ActionValue": "false"
            },{
                "ActionDescription": "Role_Not_Manager",
                "ActionVariable": "IsMan",
                "ActionValue": "false"
            },{
                "ActionDescription": "Approved_Request",
                "ActionVariable": "ReqAp",
                "ActionValue": "true"
            }
        ],
        "expectedresult":[
            {
                "ResultDescription": "No_Preview",
                "ResultVariable": "Prev",
                "Resultvalue": "false",
            }
        ],
        "Withintime": null
    },
```

Listing 6: An example of a populated logical data model in JSON after data transformation

**Algorithm 1** Pseudo code for data transformation

**Input:** $rootobjects \in RootObjects[...]$, $signal \in Signals[...]$
**Output:** $testcases \in TestSuite(TS)[...]$
**Create** a $testcase$ ▷ New object
**for each** $rootobject \in RootObject$ **do**
  **if** $rootobject$ is not Empty **then**
    **for each** $ActionString \in rootobject.actions$ **do**
      String[...] $listofActions$
      $listofActions \leftarrow actionString$ after **Splitting** $ActionString$ based on delimiter (';')
      **for each** $actionString$ in $listofActions$ **do**
        **Splitting** $actionString$ based on delimiter ('=')
        $parameter \leftarrow variableName$
        $value \leftarrow variableValue$
        **if** $parameter \in Signals[...]$ **then**
          **Find** $datatype$ of $parameter$ from Signals[...]
          **if** $parameter$ is a constraint **then**
            $testcase.WithinTime \leftarrow value$
          **else if** $parameter$ is not a constraint **then**
            **if** $datatype = Input$ **then**
              **Create** a $TestAction$ ▷ New object
              $ActionVariable$ & $ActionValue$ & $ActionDescription \in TestAction \leftarrow parameter$ & $value$ & $CurrentElementName$
            **else if** $datatype = Output$ **then**
              **Create** a $ExpectedResult$ ▷ New object
              $ResultVariable$ & $ResultValue$ & $ResultDescription \in ExpectedResult \leftarrow parameter$ & $value$ & $CurrentElementName$
            **end if**
          **end if**
        **end if**
        $testcase \leftarrow testAction$ or $expectedResult$
      **end for**
      $TS \leftarrow testcase$
      **if** $RootObject$ is not Empty **then**
        **Create** a $testcase$ ▷ New object
      **end if**
    **end for**
  **end if**
**end for**

### 3.3.2 Removal of identical test cases

A test suite generated from a behavioral model using random generation also contains duplicate or identical test cases, which provide no additional value to reveal new faults. Moreover, each test case consists of different test steps that alter the values of a limited number of parameters (based on traversed model elements) to validate the output of a system. Such test cases are usually followed by a 'Reset' test case to restore and validate the initial state of a SUT before the execution of the next test case. A 'Reset' can only be removed from a test suite if each test case contains the values of each parameter of a SUT. For instance, a test suite altering or forcing only one or two signals in each test case without retaining the default or initial values of other signals to verify the outputs of a system requires a 'Reset' before executing the next test case. On the other hand, a test suite executing an altered value of specific signals alongside default values of the remaining signals in each test case to verify the outputs of a system does not require a 'Reset'. Hence, identical test cases need to be removed while preserving the 'Reset' test cases in such test suites.

We defined a naïve comparison algorithm in the proposed approach that uses a similar definition of the Jaccard index as a similarity measure to remove the identical test cases and also preserve the 'Reset' test case generated from a model. The Jaccard index, also known as a similarity coefficient, compares the similarity between two unordered sets. It is a relative measure of the proportion of shared elements to the total number of distinct elements in the set. Hence, the algorithm takes the generated test suite as input and creates a new instance of a logical data model representing a distinct test suite to add distinct test cases. The pseudocode for naïve comparison to identify and remove the identical test cases is given below as Algorithm 2. It starts by adding the first test case to the distinct test suite (line 3) and then by performing a comparison of each test case in the generated test suite with the test cases already added to the distinct test suite based on test actions (i.e., by comparing the 'ActionVariable' and 'ActionValue' in the test actions of each test case) (lines 8 to 10). The algorithm also adds the 'Reset' after each test case to produce the resultant test suite (lines 5 and 6).

**Algorithm 2** Pseudo code for the removal of identical tests

**Input:** $testcases \in GeneratedTestSuite(GTS)[...]$
**Output:** $testcases \in DistinctTestSuite(DTS)[...]$
1: **for each** $testcase.gts \in GTS$ **do**
2: **if** $DTS$ is Empty **then**
3: $DTS \leftarrow testcase.gts$
4: **end if**
5: **if** $testcase.gts$ is a $Reset$ & $Reset$ is not added to the $DTS$ recently **then**
6: $DTS \leftarrow Reset$
7: **else if** $Reset$ is recently added to $DTS$ & $testcase.gts$ is not a $Reset$ **then**
8: **Compare** $testcase.gts$ **with each** $testcase \in DTS$ **based on** $TestActions$
9: **if** $testcase.gts \notin DTS$ **then**
10: $DTS \leftarrow testcase.gts$
11: **end if**
12: **end if**
13: **end for**

### 3.3.3 Imputing missing parameters/values

As described above, each test case generated through an FSM model consists of test steps that force or alter the values of specific parameters without retaining the default or initial values of the remaining parameters in a system. We can refer to such parameters as missing parameters/values in a test case. Hence, we must identify and impute missing parameters and their values in each test case to create the requirement traceability matrix by performing rule-based classification of test cases based on satisfied guard conditions (Section 3.4). To achieve this goal and retain the missing parameter/values from the 'Reset' state, the proposed approach uses an imputation algorithm, similar to the next observation carried backward (NOCB), based on the observed pattern to identify and impute the missing parameters and their values into each test case. It is also important to mention here that we have used Language Integrated Query (LINQ) (Meijer et al., 2006) in the implementation to find the first or default 'Reset' test cases in the test suite.

The pseudocode for the imputation algorithm is given as Algorithm 3. After transforming the data into a scenario-based test case format and the removal of identical test cases,

the defined algorithm checks for 'Reset' test cases in a test suite (line 2). It identifies the missing parameters in a test case based on the parameters or signals provided in the XML file (line 4), creates new test actions based on the missing parameters, extracts the default or initial values from the 'Reset', assigns the extracted value to the respective test action, assign a text "preserved" as a tag in test action description, and adds the test action to a corresponding test case.

**Algorithm 3** Pseudo code for imputing missing values

**Input:** $testcases \in DistinctTestSuite(DTS)[...]$ & $signals \in Signals(Sig)[...]$
1: **for each** $testcase \in DTS$ **do**
2: **Find** first or default $Reset \in DTS$
3: **for each** $signal \in Sig$ **do**
4: **Compare each** $ActionVariable \in TestAction.testcase$ with $signal$
5: **if** $testcase \in DTS$ does not contain $signal$ **then**
6: **Create** a $TestAction$ ▷ New object
7: $TestActions \leftarrow ActionVariable$ & $ActionValue$ from $Reset$ with a tag "preserved" in $TestActionDescription$
8: $testcase \in DTS \leftarrow TestActions$
9: **end if**
10: **end for**
11: **end for**

## 3.4 Classification of test cases

In this step, we created a Requirement Traceability Matrix (RTM) by classifying the test cases into different groups based on the guard conditions in a model. This step has twofold benefit: the traceability provided by RTM can be used as a measure to ensure that each requirement of an SUT has an adequate number of test cases, and it will also assist in optimizing and selecting an MCDC-adequate abstract test suite.

### 3.4.1 Classification of test cases based on guard conditions

We used a rule-based classification (Qin et al., 2009) methodology to create the RTM and groups of test cases based on extracted guard conditions. Rule-based classification that classifies the features of data based on a fixed set of defined rules. These rules represent conditions specifying the behavior of a system or input domain. Such rules can be extracted using data mining techniques (i.e., decision tree induction (Osei-Bryson, 2014)) or can also be defined by domain experts based on prior knowledge of a system or data (Jiawei et al., 2011). Similarly, in a model, guard conditions represent the rules associated with the requirements of a system, along with additional criteria to refine the system's behavior and clarify the applicability of these rules. Hence, we used guard conditions to create an RTM and classify the test cases into different groups. For instance, let us assume a system that displays relevant documents to a user after validation, based on two validation processes (i.e., validation of a request and validation of the user). Figure 5 represents three functional requirements (i.e., Req1, Req2, and Req3) of the system written in a Given-When-Then format, defining its behavior. In this context, a guard condition in a model explicitly repre-

**Table 2** Example of generated groups based on Guard Conditions

| Group ID | Guard Condition |
|---|---|
| 1 | $(ReqInit == true)\&\&(IsMan == true \| \| ReqAp == true)$ |
| 2 | $(ReqInit == false) \| \| (IsMan == false \&\& ReqAp == false)$ |

**Table 3** An example of classified test cases into groups

| Group ID | Test case ID | Inputs | | | Outcome |
|---|---|---|---|---|---|
| | | ReqInit | IsMan | ReqAp | |
| 1 | 1 | true | true | true | true (Preview) |
| | 2 | true | true | false | true (Preview) |
| | 3 | true | false | true | true (Preview) |
| 2 | 4 | true | false | false | false (No Preview) |
| | 5 | false | true | true | false (No Preview) |
| | 6 | false | true | false | false (No Preview) |
| | 7 | false | false | true | false (No Preview) |
| | 8 | false | false | false | false (No Preview) |

senting the behavior, or predicates/decisions in the implementation of the system based on these requirements, can be specified as follows:

$$(ReqInit == true)\&\&(IsMan == true \| \| ReqAp == true) \tag{1}$$

$$(ReqInit == false) \| \| (IsMan == false \&\& ReqAp == false) \tag{2}$$

Where guard condition (1) represents a combined rule of Req1, Req2, and Req3, and guard condition (2) is an illustration of an obvious requirement that represents the behavior of a system when either of the validation processes fails. Hence, all test cases that satisfy a guard condition will also cover the respective system requirement.

The pseudocode of the algorithm we used for rule-based classification is shown as Algorithm 4, whereas Table 2 shows an example of generated groups, and Table 3 illustrates an example of the classification of test cases in different groups. The implemented algorithm takes the distinct test suite generated in the previous step as input, along with the guard conditions and associated requirements extracted during the data extraction step. It evaluates each test case against each guard condition to create various groups of test cases representing an abstract level RTM (lines 4 to 6). The logical data model for RTM is shown in Listing 7. To evaluate test cases against guard conditions, we have used the dynamic expression function (Albahari and Albahari, 2008) of the LINQ library of the .NET framework in our algorithm and replaced the parameters defined in a guard condition with their values in the test actions of test cases.

```
1 class RTMModel
2 {
3     public List<string> requirements { get; set; }
4     public string guard { get; set; }
5     public List<Testcase> testcases { get; set; }
6 }
```

Listing 7: A logical data model for requirement traceability matrix

**Algorithm 4** Pseudo code for classifying and creating RTM.

**Input:** $testcases \in DistinctTestSuite(DTS)[...]$ & $guards, requirements[...] \in DataModel(DM)[...]$
**Output:** $guard, requirements[...]$ & $Testcases(TC)[...] \in RequirementTraceablityMatrix(RTM)[...]$
1: **for each** $data \in DM$ **do**
2: **Create** a new list of $Testcases(TC)[...]$
3: **for each** $testcase \in DTS$ **do**
4: bool result = **Evaluate** $testcase$ against $guard \in data$
5: **if** $result$ is true **then**
6: $TC \leftarrow testcase$
7: **end if**
8: **end for**
9: $RTM \leftarrow data.guard, data.requirements[...]$, **TC**
10: **end for**

### 3.4.2 Sub-classification of test cases based on group conditions

Each group, created in the previous step, can be a representation of more than one requirement. For instance, the guard condition (1) represents the behavior of the system covering all the requirements shown in Fig. 3. However, if we divide it into group conditions, the first group condition (i.e., $ReqInit == true$) represents Req1, the second group condition (i.e., $IsMan == true || ReqAp == true$) illustrates Req2 and a combination of both group conditions with an AND operator depicts Req3 of the system presented in Fig. 3. Hence, in our proposed approach, we further classified the test cases into subgroups based on the

**Req1:**
GIVEN:
The documents have been requested.
WHEN:
The user initiates the request (ReqInit = True).
THEN:
The system received the request.

**Req2:**
GIVEN:
The system validates the user.
WHEN:
The user who requested the document is a manager (IsMan =True) OR the user's special request has already been approved (ReqAp = True).
THEN:
The user is considered valid.

**Req3:**
GIVEN:
The system attempts to provide a preview of the documents to the user.
WHEN:
The user initiated the request AND is considered valid.
THEN:
The documents are previewed.

**Fig. 3** An example of requirements in Given-When-Then format

**Table 4** An example of generated subgroups

| Group ID | Subgroup ID | Group Condition |
|---|---|---|
| 1 | 1 | $(ReqInit == true)$ |
| | 2 | $(IsMan == true \| \| ReqAp == true)$ |
| 2 | 3 | $(ReqInit == false)$ |
| | 4 | $(IsMan == false \&\& ReqAp == false)$ |

**Table 5** An example of classified test cases into subgroups

| Group ID | Subgroup ID | Test case ID | Inputs | | | Outcome |
|---|---|---|---|---|---|---|
| | | | ReqInit | IsMan | ReqAp | |
| 1 | 1 | 1 | true | true | true | true |
| | | 2 | true | true | false | true |
| | | 3 | true | false | true | true |
| | 2 | 1 | true | true | true | true |
| | | 2 | true | true | false | true |
| | | 3 | true | false | true | true |
| 2 | 3 | 5 | false | true | true | false |
| | | 6 | false | true | false | false |
| | | 7 | false | false | true | false |
| | | 8 | false | false | false | false |
| | 4 | 4 | true | false | false | false |
| | | 8 | false | false | false | false |

group conditions in each guard condition. It is important to mention here that a complex logical expression representing a guard condition can be a composition of a nested expression grouped in parentheses at different levels in an expression tree (Desolda et al., 2017). The higher level refers to the level of the expression close to the root of the tree, whereas the lower level refers to the level of the expression close to the leaves of the tree (Crama and Hammer, 2011). Depending on the levels of nested expressions, different numbers of groups can be created at different levels. For instance, in an expression tree, the highest levels of the guard conditions (i.e., (1) and (2)) in the given example are 1 that contain only two nested expressions each at the lowest level (i.e., level 0), hence, the test cases can only be further classified into four subgroups (i.e., two subgroups for each guard condition) at lowest level.

In our proposed approach, we have identified the group conditions within each guard condition by utilizing the regular expression library (Stubblebine et al., 2007) in the .NET framework. We then evaluated each test case in a corresponding group to create subgroups. To manage this data, we developed a logical data model that stores the list of sub-classified test cases. This model includes a 'guard', a list of 'requirements', and an associated list of 'group conditions'. These group conditions are structured as a matrix of 'group ID', 'group condition', and a list of 'test cases' that satisfy each group condition as shown in Listing 8. Table 4 provides an example of subgroups, while Table 5 illustrates the classification of test cases in identified subgroups.

```
class GroupConditionModel
{
    public string guard { get; set; }
    public List<string> requiremenets { get; set; }
    public List<GroupCondition> group { get; set;}
}
class GroupCondition {
    public int groupID { get; set; }
    public string groupcondition { get; set; }
    public List<Testcase> testcases { get; set; }
}
```

Listing 8: A logical data model for sub-classification

### 3.5 Test suite reduction

We defined a greedy and naïve comparison-based approach to optimize the generated test suite. The greedy approach is focusing on selecting MCDC-adequate test cases, while the naive comparison approach identifies diverse test cases and performing similarity-based test reduction. MCDC is a coverage criterion recommended by various standards (i.e., EN 50128, EN 50657 Chen et al., 2019, and ISO 26262) for validating the safety critical systems. It is a stricter criterion compared to decision or branch coverage that requires the coverage at condition level (Jones and Harrold, 2003). MCDC ensures that each condition in a decision has shown an independent effect on the outcome of the decision (Hayhurst, 2001). To achieve this, we implemented a greedy algorithm guided by MCDC to select the MCDC-adequate test suite based on guard conditions. Subsequently, we employed a naïve comparison algorithm, similar to the one described in Section 3.3.2, to identify and remove similar test cases within each subgroup.

#### 3.5.1 test suite reduction using greedy algorithm

In this step, we selected the MCDC-adequate test cases by analyzing the effect of each test action in a test case on the outcome of the corresponding decision. We defined a greedy algorithm that iteratively evaluates and selects test cases with respect to their contribution to satisfy MCDC from each subgroup. The pseudocode of the test suite reduction using a greedy algorithm is presented as Algorithm 5. It takes a test case from a subgroup of the test suite and creates different variants by inverting the value of a test action in each variant based on the conditions in the group condition (excluding test actions containing the retained value from 'Reset') (line 6). The algorithm evaluates all the variants against the group condition and observes the effect on the decision (line 7). If any variants demonstrate an effect on the outcome of the respective group condition (i.e., any variant of a test case does not satisfy the group condition), the test case is added to the resultant test suite, ensuring that the condition independently effect the decision. For example, let's assume a test case with 'Group ID' 1, 'Sub-Group ID' 1, and 'Test Case ID' 1 shown in Table 5. The algorithm inverts the value of 'ReqInit' from true to false, evaluate it against the group condition (i.e., $ReqInit == true$), and add it to the resultant test suite as it will not satisfy the group condition. Conversely, in case of a test case with 'Group' 1, 'Sub-Group' 2, and 'Test Case ID' 1, the algorithm will create two variants of the test cases by inverting the value of

'IsMan' in variant 1 and inverting the value of 'ReqAp' in variant 2. It evaluates both variants against the corresponding group condition (i.e., $IsMan == true || ReqAp == true$) and remove the test case from the resultant test suite since neither variant impacts the outcome of the group condition. Table 6 provides an example of the resultant test suite.

**Algorithm 5** Pseudo code of a greedy algorithm for test reduction.

**Input:** $Testcases(TC)[...]$ & $groupcondition \in GroupCondition(GC)[...] \in GroupConditionModel(GCM)$
**Output:** $ReducedTestcases(RTC)[...]$
**for each** $group \in GC$ **do**
  **bool** $hasEffect$
  **Create** a new list of $ReducedTestcases(RTC)[...]$
  **for each** $testcase \in TC$ **do**
    **for each** $testaction \in testcase$ **do**
      **Create** a new $Varianttestcase$ by inverting the value of $testaction$ corresponding to $groupcondition$ except the "preserved" ones
      $hasEffect$ = **Evaluate** the effect of $Varianttestcase$ against $groupcondition$
      **if** $hasEffect$ **then**
        $RTC \leftarrow testcase$
        break the inner loop
      **end if**
    **end for**
  **end for**
**end for**

### 3.6 Similarity-based test reduction

Similarity-based test reduction deals with the identification and removal of similar test cases within each subgroup. We have used a naïve comparison algorithm as defined in Section 3.3.2, but limited the comparable parameters/test actions between the test cases corresponding to the conditions in a group condition. For instance, in Table 6, all the test cases in the subgroup with 'Sub-Group ID' 1 contain the same values for 'ReqInit' involved in the respective sub-condition (i.e., $ReqInit == true$). Hence, the first test case from the subgroup will be added to the reduced test suite, and the remaining will be removed. Table 7 represents an example of a reduced test suite generated by the proposed approach.

**Table 6** An example of the resultant test suite after the greedy approach

| Group ID | Subgroup ID | Test case ID | Inputs | | | Outcome |
|---|---|---|---|---|---|---|
| | | | ReqInit | IsMan | ReqAp | |
| 1 | 1 | 1 | true | true | true | true |
| | | 2 | true | true | false | true |
| | | 3 | true | false | true | true |
| | 2 | 2 | true | true | false | true |
| | | 3 | true | false | true | true |
| 2 | 3 | 5 | false | true | true | false |
| | | 6 | false | true | false | false |
| | | 7 | false | false | true | false |
| | | 8 | false | false | false | false |
| | 4 | 4 | true | false | false | false |
| | | 8 | false | false | false | false |

**Table 7** An example of the reduced test suite

| Group ID | Subgroup ID | Test case ID | Inputs | | | Outcome |
|---|---|---|---|---|---|---|
| | | | **ReqInit** | **IsMan** | **ReqAp** | |
| 1 | 1 | 1 | true | true | true | true |
| | 2 | 2 | true | true | false | true |
| | | 3 | true | false | true | true |
| 2 | 3 | 5 | false | true | true | false |
| | 4 | 4 | true | false | false | false |

## 4 Experimental evaluation

This section present descriptions of case studies, the process of creating test suites, an overview of experimental setup, and the results of the comparative evaluation.

### 4.1 SUT

In this paper, we have considered two subsystems (Fire Detection System & Traction Control System), controlled and coordinated by TCMS, and belonging to one of the vehicle family of metro trains developed at Alstom. The TCMS is a complex and centralized distributed control system designed for controlling and coordinating various subsystems such as a fire detection system, smoke detection system, traction, doors, and more. It ensures the safe and efficient operations of regular and safety-critical functions of these subsystems and facilitates real-time monitoring of a train. The TCMS employs various networks such as the Ethernet Consist Network (ECN), Multi-function Vehicle BUS (MVB), Centralized Traffic Control (CTC), etc. to establish communication between different subsystems and devices as well as to consolidate routing decisions (Zafar et al., 2021). The Modular Input/Output-Safe (MIO-S) and Central Control Unit (CCU-S) devices are connected through these networks to control and manage the safety-critical functions of a train.

Due to privacy considerations, the graphical FSM models of the SUTs are not included in the paper. However, the FSM models are described textually, with implementation-specific states and transitions. Each FSM representing a SUT is organized into several interconnected sub-diagrams representing distinctive components. In GraphWalker, their interaction is realized through a set of shared nodes that provide transition points for different behavioral flows. Moreover, each edge connecting the nodes in sub-diagrams represents one or a set of signals to achieve a certain state within the system.

Table 8 presents a comparative overview of both subsystems of TCMS as well as FSM models representing them. The number of requirements can provide insight into the software's complexity, scope, and testing objectives (Black, 2014). Meanwhile, the parameters from different components can be used to reflect the testing efforts required to create, execute, and evaluate testing scenarios during the testing process (Spillner and Linz, 2021). The requirements primarily define functional and non-functional aspects and behavior of the system, while the parameters influence the system's behavior. These parameters include inputs (In) that represents the data provided to the system, outputs (Out) illustrate the response of the system, and constraints (Cons) defining the limitations or boundaries of a system, such as response time. Model size characterizes the structural breadth of the FSM model representing the SUT based on the number of states and transitions. The guard complexity

**Table 8** Comparison of case studies in terms of number of requirements, parameters/signals involved, model size, guard complexity, and dominant complexity type

| Subsystem | No. of Req. | Parameters/Signals Involved | | | | Model Size | | Guard Complexity | | | Dominant Complexity Type |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | In | Out | Cons | Total | States | Transitions | AD | MD | AAC | |
| Fire Detection | 5 | 12 | 2 | 1 | 15 | 15 | 53 | 4.1 | 6 | 11.1 | Behavioral (Logical) |
| Traction Control | 10 | 18 | 1 | 1 | 20 | 62 | 84 | 4 | 5 | 6 | Structural (State-Space) |

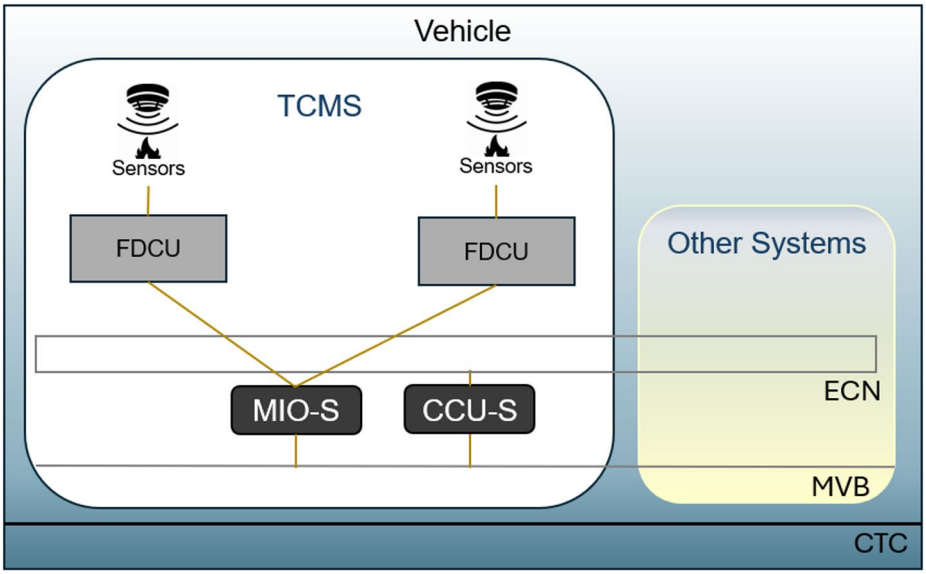


**Fig. 4** An architectural illustration of fire detection system

captures the intricacy of logical expressions in the FSM model, quantified by parsing each guard into a Boolean expression tree and computing average depth (AD), maximum depth (MD), and average number of atomic conditions (AAC). Similarly, the dominant complexity type reflects the possible primary source of redundancy during MBT test generation.

### 4.1.1 Case study 1: Fire detection system

An overview of the fire detection system of TCMS at the architectural level along with its FSM model created using GW studio is given below.

**Architectural description** *The fire detection system in TCMS is a safety-critical subsystem that is used to detect fire in the cabs of the trains based on the inputs provided by Fire Detection Control Units (FDCUs), as shown in Fig. 4. It uses two instances of FDCUs interconnected with fire sensors to detect two types of fire, i.e., internal and external. Each FDCU can possess two types of states, i.e., Master or Slave. The MIO-S device receives the*

*signals from FDCUs along with the signals from sensors and reports them to CCU-S. The CCU-S performs the computational logic in compliance with received signals and functional requirements specified in the documents to indicate the type of fire with a corresponding signal in the driver's cab via MIO-S.*

**FSM model description** *An FSM model of the fire detection system comprises three sub-diagram, where two sub-diagrams represent the behavior of FDCUs, and one sub-diagram incorporates the behavior of TCMS as a black box to indicate the detection of fire. All three sub-diagram form a cyclic state-transition structure enabling the generation of test cases during traversal.*

$FDCU(A) \rightarrow FDCU(B) \rightarrow TCMS \rightarrow FDCU(A)$.

*The two instances of the sub-diagram represent the FDCUs consist of identical internal behavior and are connected through different entry and/or exit nodes (i.e., FDCU1Start, FDCU1Exit, FDCU2Start, and FDCU2Exit). Each FDCU also contains internal nodes (i.e., Master and Slave) depicting Master and Slave states of the component. Moreover, each internal node is connected to respective entry and exit node based on internal state-transition logic. The third sub-diagram represents the behavioral implementation of TCMS comprises of an entry node (i.e., TCMSisActive), and an exit node (i.e., TCMSExitState) preceded by a reset node (i.e., Reset) to restore the initial or default state of the system. Similar to FDCUs the TCMS sub-diagram also consists of internal nodes (i.e., InternalAndExternalFire, NoIntANDExt, InternalFire, and ExternalFire) describing the states of a TCMS when certain fire is detected based on the signals from FDCUs.*

### 4.1.2 Case study 2: Traction control system

A brief description of the architecture of the traction control system along with its FSM model is given below.

**Architectural description** *The traction control system in TCMS is also a safety-critical subsystem used to ensure a train's effective and safe acceleration under certain conditions. We selected 10 requirements that deal with the acceleration of a train based on the inputs provided by the different types of doors (i.e., cab, detrainment, passenger), Vehicle On-Board Controller (VOBC), and Brake Control Unit (BCU) of each car configured in a train as shown in Fig. 5. A typical car in a train usually consists of two cab doors (i.e., one door on each side), while the number of detrainment doors, passenger doors, VOBC, and BCU can vary depending on the design; however, each car should contain at least one VOBC and BCU each. It uses different sensors and actuators to monitor and control the doors, VOBC, and BCUs, respectively. TCMS reads the statuses of the doors (i.e., open, closed, bypassed) in each car connected with MIO-S. In addition, it also reads the information about train acceleration (i.e., standstill, moving) through relays connecting VOBC and BCU to the MIO-S. The MIO-S device receives the signals indicating the corresponding responses from the sensors through DIRs and ZVRs and reports them to CCU-S. The CCU-S performs the logical computation based on the implemented functionality specified in the requirement documents and sends the output signal to control the acceleration of a train.*

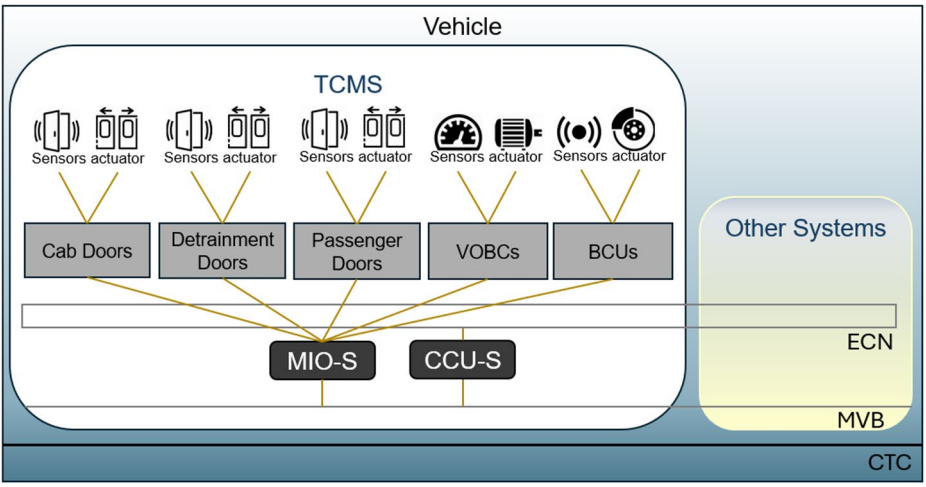


**Fig. 5** An architectural illustration of the traction control system

**FSM model description** *To depict the behavior of selected requirements of the train control subsystem in TCMS, we created seven sub-diagrams, where each sub-diagram represents the behavior of different components of the subsystem i.e., cab doors, detrainment doors, passenger doors, all doors, VOBC, BCU, and TCMS. In the case of traction control system, all the sub-diagrams form an interconnected FSM with cyclic execution flow where some sub-diagrams have high-level cyclic behavior while others have arbitrary internal state-transition structures.*

$$BCU \rightarrow VOBC \rightarrow \left\{ \begin{array}{c} CabDoors \\ DetrainmentDoors \\ PassengerDoors \\ AllDoors \end{array} \right. \rightarrow TCMS \rightarrow BCU$$

*The complete FSM model comprises of a Start node which is an initial and shared node, 24 shared nodes (i.e., GoToBCUStatus, BCUInit, BCUStatus, BCUExit, GoToVOBC, VOBCInit, VOBCStatus, VOBCExit, GoToCabDoor, CabDoorsInit, CabDoorsStatus, GoToDetrainmentDoor, DetrainmentDoorInit, DetrainmentDoorStatus, GoToAllDoors, AllDoorsInit, AllDoorsStatus, GoToPassengerDoor, PassengerDoorInit, PassengerDoorStatus, CabDoorExit, DetrainmentDoorExit, PassengerDoorExit, and AllDoorExit) connecting all the sub-diagrams, and 37 nodes illustrating the internal functional aspect and states of each component in a traction control system, collectively. Sub-diagrams for cab doors and detrainment doors contain the behavior of both sides of doors of a cab (i.e., Side A and Side B) and detrainment doors in terms of the states of each door (i.e., Open, Close, ByPass, and NoByPass), respectively. Whereas the sub-diagrams for passenger and all doors capture the behavioral attributes and statuses (i.e., Open and Close) of the passenger doors and all doors within a car of a train, respectively. Sub diagrams for VOBC and BCU encapsulate the respective characteristics in terms of determining the speed of a train (i.e., ZeroSpeed and NoZeroSpeed). Lastly, the sub diagram for TCMS offers an integrated topology of the system's components and presents the decision-making process of the trac-*

*tion control system (i.e., TractionEnabled and TractionDisabled), including a Reset node to reinstate the default or initial values of all attributes in the system.*

### 4.2 Development and execution of test suites

To comparatively evaluate the effectiveness and efficiency of the test suite generated from the proposed approach, we used the test suites (i.e., test cases and scripts) created by the testers at Alstom for the selected subsystems. For MBT, we analyzed the requirements and test specification to fully understand the subsystems under consideration and utilized GraphWalker [4] (GW), an open-source MBT tool, to create FSM models representing the explicit behavior of the subsystems and to generate abstract test cases. We also implemented and integrated our proposed approach with our previously implemented tool, i.e., Model-based Test script GenEration fRamework (TIGER) (Zafar et al., 2021), and named it TIGER+, to reduce the test suite size and to generate concrete test scripts. The test scripts are then executed on the SUTs to produce test verdicts. A detailed description of the activities for the development of the test suites is given in the following subsections.

#### 4.2.1 Manual test suite creation and execution

The test suite creation at Alstom is a systematic process that starts with the understanding of system requirements. Critical functionalities and potential risks are identified, and test cases are written in natural language in the form of test steps. Each test step consists of inputs, expected outputs, and constraints such as the response time of a system according to the requirements of the system. The test cases created by a tester at Alstom cover different potential scenarios based on equivalence class partitioning and boundary value analysis in compliance with certain safety standards and regulations (i.e., EN 50128 and EN 50657), and are complemented by requirement coverage to ensure the coverage of each requirement of the system. However, in some cases, MCDC is also considered as a coverage criterion for the development of test cases to validate safety-critical software (Zafar et al., 2023). These test cases are then converted into test scripts manually and executed on the SUT using Alstom-specific testing framework and libraries to produce test verdicts.

#### 4.2.2 MBT-generated test suites and their execution

To generate test suites, we created the FSM models of the selected subsystems as described in Section 4.1.1 and 4.1.2 through conducting an in-depth analysis of system requirements and test specification documents to understand the system behavior, determine its states and transitions, and interpret its behavior into guard conditions. We utilized the GW studio version to create these FSM models and the TIGER+, which uses the command-line version of GW, to generate three versions of test suites for each subsystem (i.e., MBT-generated, optimized at the highest level, optimized at the lowest level) using a random generation algorithm and 100% edge coverage. The MBT-generated test suite contains all the test cases, including redundant test cases generated from the selected coverage criteria and generation algorithm. Whereas test suites optimized at the highest and lowest levels contain reduced subsets of MBT-generated test cases through the TIGER+. As mentioned

[4] https://github.com/GraphWalker/graphwalker-project/wiki

in Section 3.4.2, sub-classification of test cases for test reduction can be carried out at different levels depending on the complexity of the guard conditions specified in the FSM to depict the behavior of the system in conformance with requirements. Hence, in the case of our selected subsystem, for evaluation purposes, we only sub-classified the test cases at the highest and the lowest levels of the nested expressions (i.e., guard conditions), yielding two different sets of abstract test cases. Moreover, to execute the generated test cases on the SUT, they need to be transformed into executable test scripts and require the Alstom-specific libraries and framework. The transformation of model-generated abstract test cases into concrete test scripts is an automated process as defined in Zafar et al. (2021). It requires providing the XML file to TIGER+ with information such as the types of each signal (i.e., input, output), the data type of each signal (e.g., Int, Boolean, etc.), and the technical names of each signal. Thus, limiting the manual efforts to the creation of models, XML files, and configuring the test execution environment at the SiL or HiL level. The generated test scripts are then executed on the SUTs to produce test verdicts.

### 4.3 Experimental setup

In our cases, the selected subsystems are developed through Programmable Logical Controller (PLC) programming using Functional Block Diagrams (FBDs), which are then compiled and transformed into source or machine-level code through tools provided by PLC vendors. Moreover, the development of a safety-critical system at Alstom follows certain safety standards (i.e., EN50657) to ensure the reliability and safety of the system. These standards require a well-defined structure of FBD programs for the design and suggest the use of specific FBD operators for the development of safety-critical embedded software. Hence, to assess the fault detection effectiveness of the test suites, we used mutation analysis as described in Section 2.5 and performed similar experimentation as our previous paper (Zafar et al., 2023). We created several mutants of FBD programs by selecting seven mutation operators provided in literature (Enoiu et al., 2016; Oh et al., 2005; Shin et al., 2012) based on safety-critical industrial case studies and FBD specific faults i.e., Logic Block Deletion Operator *(LDO)*, Logic Block Insertion Operator *(LIO)*, Logic Block Replacement Operator *(LRO)*, Logic Block Replacement Operator-Improved *(LRO-I)*, Negation Insertion Operator *(NIO)*, Time Block Replacement Operator *(TRO)*, and Value Replacement Operator *(VRO)* to mimic programmers' mistakes. It is important to mention here that the mutations with *TRO* were found not to be applicable to the traction control system, as no timer block was used in the FBD programs.

After creating FBD mutants, we utilized Alstom-specific compiling tools to generate different builds of virtual trains, which are simulations representing a train. We also utilized the Alstom-specific testing framework and simulation platform to execute the test scripts on the SUTs (i.e., SUTs with no known fault and their mutated versions) and generate test verdicts at the Software-in-the-Loop level.

### 4.4 Results

This section provides our experimental results based on the mutation analysis to comparatively evaluate the MBT-generated test suite, optimized test suites, and manually created test suite in terms of fault detection effectiveness and test execution efficiency.

### 4.4.1 Fault detection effectiveness of test suites

To evaluate the test cases in terms of fault detection effectiveness, we injected faults into the original FBD programs of both subsystems of TCMS. We created 42 and 101 mutants based on the selected operators as mentioned in Section 4.3, respectively. We also calculated the mutation scores of each test suite as shown in Table 9 by executing the test suites on the SUT and analyzing the test verdicts. It is also important to mention that we encountered distinct challenges that influenced the creation of the number of mutants for both subsystems. For instance, the implemented traction control subsystem deals with multiple configurations of cars in a train; however, to ensure focus and a rigorous evaluation within our research context, we only considered one configuration of a car in the train and injected the faults in the respective FBD programs. Similarly, the fire detection system consisted of similar requirements which were implemented by generating different instances of the same FBD programs (Zafar et al., 2023). This limited the creation of mutants for the evaluations. Additionally, we used strong mutation to calculate two types of mutation scores (i.e., mutation score while including stubborn mutants and mutation score while excluding stubborn mutants) at the system level and only considered non-equivalent mutants in our results by eliminating the equivalent ones using a thorough manual analysis of the test results and FBD programs.

The results of the experimentation show that the test suites created using manual testing practices, MBT as well as optimized at the highest level and optimized at the lowest level by the approach were able to kill 38, 39, 40, and 38 mutants, for the fire detection system, and 68, 88, 87, 88 mutants for the traction control system across 42 and 101 mutants, respectively. The comparative analysis of the result shows that the test suite optimized at the highest level provided the highest mutation scores of 95.5% when stubborn mutants were included and 100% when stubborn mutants were excluded in our mutation score calculations. In contrast, MBT-generated delivered 92.85% and 97.5% mutation scores, and the test suites optimized at the lowest level and created manually were able to provide the lowest mutation scores of 90.47% and 95% each by uncovering different interaction faults while including and excluding stubborn mutants, respectively. In the case of the traction control system, the MBT-generated test suite and test suite optimized at the lowest level achieved

**Table 9** Comparative mutation analysis between the MBT-generated test suite, optimized test suites, and manually created test suite

| Case Studies | Test Suite | No. of Test Cases | Mutation Analysis | | | |
|---|---|---|---|---|---|---|
| | | | Alive | Killed | Score incl. Stubborn | Score excl. Stubborn |
| Fire Detection System | Manual | 18 | 4 | 38 | 90.47 | 95 |
| | MBT-generated | 307 | 3 | 39 | 92.85 | 97.5 |
| | Optimized at highest level | 46 | 2 | 40 | 95.23 | 100 |
| | Optimized at lowest level | 22 | 2 | 38 | 90.47 | 95 |
| Traction Control System | Manual | 23 | 33 | 68 | 67.32 | 75.28 |
| | MBT-generated | 209 | 13 | 88 | 87.12 | 98.87 |
| | Optimized at highest level | 40 | 14 | 87 | 86.13 | 97.75 |
| | Optimized at lowest level | 22 | 13 | 88 | 87.12 | 98.87 |

the highest mutation scores of 87.12% and 98.87%, respectively. The test suite optimized at the highest level acquired scores of 86.13% and 97.75%, while the manually created test suite provided the lowest mutation scores of 67.32% and 75.28% when stubborn mutants were included and excluded, respectively.

We further analyzed the test suites and test results to investigate the variation in the mutation score achieved by each test suite, especially, to identify a possible logical explanation for the increase in mutation score of the test suite optimized at the highest level in the fire detection system which is originally a subset of the MBT-generated test suite. The analysis suggests that the test suites i.e., optimized at the lowest level and created manually did not contain the test cases required to detect the injected faults. However, in the case of the MBT-generated test suite, it could not uncover a fault that was detected by both optimized test suites, resulting in a slightly lower mutation score than the test suite optimized at the highest level. During the analysis of this specific undetected fault, the only differences we observed between these test suites (i.e., MBT-generated and optimized test suites) were the number of parameters used in test cases to create different scenarios and the sequence of test cases. For instance, each test case, in the MBT-generated test suite, forces some specific signals, verifies the expected outputs in conformance with the expected behavior of the system, and is followed by a 'Reset' test case to restore and validate the system's initial state. In contrast, the optimized test suites, force the value of each signal involved along with verification of the expected outcome in each test case, and execute the 'Reset' test case only at the end of the test scripts. As a result, both optimized test suites succeeded in creating possible scenarios to uncover that specific fault and significantly enhanced the fault detection capabilities of test suites that arise from the interdependencies of parameters, which may only manifest when multiple parameters are tested together within a single test case. Moreover, a similar structure of test cases was observed in the manually created test suite, which was also able to uncover that specific fault. Hence, we argue that the proposed approach provides a more robust mechanism to optimize MBT-generated test cases irrespective of behavioral and structural redundancy sources, as it ensures that both individual parameter behavior and their interactions are thoroughly validated.

#### 4.4.2 Test execution efficiency of test suites and impact of the optimization approach

For the test execution efficiency of the test suites, we reported the approximate time required for each test suite to execute all the test cases on the SUT with no known faults and on the mutated versions as shown in Table 10. Moreover, we also calculated the test suite reduction percentage achieved by the proposed approach to analyze and report its effectiveness.

The results reported in Table 10 show that the MBT-generated test suite required the highest execution time when executed on the SUT with no known faults as well as mutated SUT for both subsystems. In the case of the fire detection system, an execution time of approximately 21 minutes and within the range of approximately 120 to 140 minutes was witnessed when executed on SUT with no known faults and mutated SUTs, respectively. Similarly, in the case of the traction control system, approximately 7.5 minutes were observed when executed on SUT with no known faults and within the range of approximately 10 to 12.5 minutes were perceived when executed on mutated SUTs. On the other hand, test suites created manually, optimized at the highest level, and optimized at the lowest level provided 7, 8, and 7 minutes and approximately 10 to 12, 13 to 17, and 10 to 13 minutes of execution

**Table 10** Comparative assessment of the impact of the proposed optimization approach and test execution time of each test suite

| Case Studies | Test Suite | No. of Test Cases | Reduction in % | Approx. Execution Time on SUT in minutes | |
|---|---|---|---|---|---|
| | | | | No known faults | Mutated SUTs |
| Fire Detection System | Manual | 18 | N/A | 7 | 10–12 |
| | MBT-generated | 307 | N/A | 21 | 120–140 |
| | Optimized at highest level | 46 | 85.34 | 7 | 13–18 |
| | Optimized at lowest level | 22 | 92.83 | 8.5 | 10–12.5 |
| Traction Control System | Manual | 23 | N/A | 6 | 6–8 |
| | MBT-generated | 209 | N/A | 7.5 | 8–10 |
| | Optimized at highest level | 40 | 80.86 | 6.5 | 6.5–8 |
| | Optimized at lowest level | 22 | 89.47 | 5.5 | 5.5–7.5 |

time when executed on the SUTs i.e., SUT with no known fault and the mutated versions, respectively. Moreover, the analysis of the test verdict shows that the execution time of each test suite depends on the number of test cases in a test suite along with the number of test cases that detect a fault and the waiting time required by a test case in the case of failure to verify a signal in compliance with requirement specification.

As depicted in Table 10, the TIGER+ was able to achieve a significant reduction in the size of the MBT-generated test suite for both subsystems yielding 46 and 22 test cases out of 307 in the case of the fire detection system, and 40 and 22 test cases out of 209 in the case of the traction control system when optimized at the highest and lowest levels, respectively. These findings highlight the effectiveness of the proposed approach in reducing the size of the MBT-generated test suite within the range of approximately 80% to 85% at the highest-level optimization and within the range of approximately 89% to 92% at the lowest-level optimization while preserving the fault detection effectiveness and increasing the execution efficiency of the test suite.

## 5 Discussion

In this study, we employed a similarity measure to identify duplication and utilized a method similar to NOCB to impute missing values in test cases. Both techniques were chosen after careful consideration to increase the accuracy and reliability of the proposed approach. We acknowledge that the choice of an appropriate similarity measure threshold is crucial for ensuring accurate duplication detection. Typically, a higher threshold of 0.7 to 0.9 is recommended to maintain a balance between false positives and negatives across various domains. However, in our case, the strict threshold for the similarity measure is set (i.e., 1.0), as two sets should only be considered duplicates if they are identical with no variation whatsoever. Similarly, to impute the missing values in the test cases, we also acknowledge the potential limitation, particularly concerning the biases of the NOCB method that may arise due to structure or patterns in the data. However, as the imputed values originated from a reset test case that is usually executed to restore the system's intended initial state and to verify the non-existence of unintended system variations over time, it significantly reduces

the risk of introducing distortions due to structure or patterns. Additionally, we validated the consistency of the imputed values by comparing them with the values in reset test cases and baseline test results of manually created test suites, confirming that NOCB was the most appropriate method given the structured reset mechanism.

Our results regarding the effectiveness of the TIGER+ show a comparable reduction of over 80% in the test suite in both case studies when optimized at the highest and lowest levels. However, the test suites in this study are generated using non-deterministic random walks that may yield different test suites of varying sizes and redundancy patterns. For instance, generating different test suites in multiple attempts using a random generation algorithm can produce a different number of test cases in each test suite, including redundant test cases. Hence, this variability may influence the absolute reduction percentage, but does not affect the source of redundancy determined by FSM structure, guard complexity, and parametrization. It is also important to mention here that to avoid bias towards small and large test suites in this study, we generated the test cases from the model using random walks multiple times and selected a median-sized test suite for the evaluation.

We also found the results of mutation analysis in the case of MBT-generated and manually created test suites to be similar to our previous study (Zafar et al., 2023) in terms of detected faults, regardless of the size of the MBT-generated test suite. These results suggest that MBT can complement existing manually created test suites by detecting additional injected faults under our evaluation setup. Moreover, during an in-depth analysis of the mutation scores and undetected faults, we also identified a few hard-to-kill mutants among all the mutants injected into the FBD programs of both systems. We further examined the behavioral characteristics of these mutants to understand their functional impact and relevance in our research context. The subsequent analysis revealed that, in the case of the traction control system, all mutations implemented by replacing logical conjunction (AND) operators with exclusive NOR (XNOR) operators, or those in which XNOR was connected to AND operators, exhibited a persistent resistance to detection demonstrating stubborn behavior. Similar patterns were noted when mutants were injected through the replacement of disjunction (OR) with exclusive disjunction (XOR) or XOR connected to OR operators. We also found two mutants that exhibited stubborn behavior due to the presence of an input signal, which was an output signal of another subsystem and could only be tested upon integration. Hence, we categorized them as “stubborn” since for all these types of mutants, special test cases were required to propagate essential inputs to the fault-injected level of the FBD programming tree. Similarly, in the case of the fire detection system, faults injected based on *TRO* were categorized as "stubborn" as special test cases targeting the basic integrity of the system were required to detect the time-related faults (Zafar et al., 2023). In total, 2 out of 42 and 12 out of 101 stubborn mutants were identified in the FBD programs of the fire detection system and traction control system, respectively.

Excluding the stubborn mutants from the results, our analysis also showed that the MBT-generated test suites struggled to uncover different interaction faults injected based on the *LRO* in both cases, where the MBT tool failed to generate test scenarios. Meanwhile, due to the greedy algorithm in the proposed approach, the test suite optimized at the highest level did not select any test case that could be used to detect a specific fault injected based on the *LRO*, and hence, it provided a relatively lower mutation score in the case of the traction control system. For a similar reason, a lower mutation score was achieved by the test suite optimized at the lowest level in the case of the fire detection system while detecting faults

injected based on *LRO* and *LIO*. On the other hand, the manually created test suite achieved the lowest mutation score in detecting faults injected based on LRO in the case of the fire detection system, while in the case of the traction control system, a significantly varying behavior of the manually created test suite was observed in detecting faults across all the different mutant operators (i.e., it achieved the highest effectiveness in detecting faults based on NIO and lowest effectiveness in detecting faults based on LIO). It is also important to mention that these findings concern the specific test suites and artificially injected faults evaluated under our experimental conditions. They should not be generalized to manual testing as a whole or interpreted as an assessment of the overall verification and validation process or product quality.

## 6 Validity threats

The factor that can affect the internal validity of this study is the conformance of the FSM model depicting the behavior of the selected system to its requirements. The modeling of the system is a manual process that requires sound domain knowledge. The requirements at Alstom are written in natural language using the Given-When-Then format, and there is a chance that misinterpretation of these requirements can impact the conformance of model and generated test suite. However, to mitigate this threat, we spent a substantial amount of time understanding the requirements and the system through careful analysis of the requirements and test specifications. We also iteratively created the models by getting continuous feedback on their correctness from a testing team at Alstom. Furthermore, in order to evaluate the correctness of FSM models, we executed the FSM-based generated test suites on the SUTs with no known faults. The results of executions were found to be consistent with the expected outcomes previously established through manually created tests at Alstom. This consistency indicates that the FSM-generated test suites align with the system's verified behavior, thus confirming the correctness of the FSM models. It is also important to mention here that, to avoid the risk of bias, the manual test suites used in this study were created by another tester without prior knowledge of the model or MBT-generated test cases.

The threats that can influence the reliability and external validity of the study relate to the modeling notation, generation algorithm, human experience of modeling, particularities of the test generation tool, and tools specific to Alstom. TIGER+ is specifically designed for the test cases generated through GraphWalker and contains certain particularities related to the FSM model and the format of files representing the model and test cases. Similarly, the test suite contain particularities related to Alstom-specific testing libraries, development, and testing environment that may not apply to other domains. However, to adapt and generalize the proposed approach to other contexts, we recommend to keep mainly three levels of the given reduction strategy (i.e., behavioral-aware grouping, coverage-based selection, and multi-level similarity-based reduction). On the other hand, the guard conditions can be replaced with domain-specific decision rules, domain-specific coverage criteria can be selected instead of MCDC, and the data extraction process and models representing extracted data to transform them into structured test cases can be adapted to the particularities of the domain and modeling tool. We also suggest replacing the imputation mechanism if necessary, as in our case, the reset states of the given system were already known. In addition to these recommendations, a detailed description of the proposed approach, imple-

mentation procedures for the tool, and experimental methodology for other researchers to understand and apply the proposed approach in their contexts have been comprehensively provided in this work. We also argue that different modeling notations, generation algorithms, and coverage criteria may produce different results in terms of test suite reduction and fault detection effectiveness. Therefore, more industrial case studies from various domains are needed for the generalization of the tool and knowledge in different contexts. We used mutation analysis in this study to evaluate the fault detection effectiveness, as it is often recommended in the literature. The mutant operators were carefully selected after discussions with test engineers at Alstom and based on our prior experience of working with FBD programs.

## 7 Conclusion and future work

In this paper, we presented an optimization approach and implemented a tool, TIGER+, to reduce the size of a test suite generated through an FSM model for system-level testing of safety-critical embedded software. It leverages different automated algorithms to extract, pre-process, and classify the test data into different groups based on guard conditions. It also uses a greedy algorithm guided by MCDC adequacy criteria and a similarity measure inspired by the Jaccard Index to identify and eliminate redundant test cases while complying with safety standards and minimizing the effect on the fault detection rate of the test suite. We also performed a comparative mutation analysis of the test suites reduced by TIGER+, the MBT-generated test suite, and the test suite created manually in terms of fault detection effectiveness and test execution efficiency in industrial settings. Our results show a significant reduction in the MBT-generated test suite, i.e., approximately within the range of 80% to 85% (highest level expression) and approximately within the range of 89% to 92% (lowest level expression) when optimized within the expression tree. The experimental results also indicate that TIGER+ not only contributes towards the test suite optimization to improve the test efficiency but also preserves the fault detection rate despite the different redundancy sources across the two systems, showcasing its potential to enhance resource utilization and testing effectiveness.

In the future, we intend to perform a rigorous evaluation of the proposed approach using more industrial case studies to examine the test suite reduction and its effect on fault detection effectiveness in different contexts. Based on our analysis, we believe that exploring the impact of other similarity measure thresholds as well as algorithms (e.g., for imputation and MCDC adequacy) within the proposed approach for test suite optimization is another possible future work. Moreover, a thorough code-level analysis to measure the coverage achieved by test suites at the structural level is also warranted. Finally, future studies may perform a systematic comparative evaluation of the proposed approach against existing optimization techniques to assess its relative effectiveness.

**Acknowledgements** This work has received funding from: the European Union's Horizon 2020 research and innovation program under grant agreement No. 957212; the ECSEL Joint Undertaking (JU) under grant agreement No 101007350. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, Spain; the Swedish Innovation Agency (Vinnova) through the SmartDelta and NRPCES projects.

**Author Contributions** M.N.Z: conceptualization, methodology, software, investigation, writing–original draft; W.A: conceptualisation, resources, writing: review and editing, project administration, and funding acquisition; E.P.E: conceptualisation, writing: review and editing; Z.H: validation, resources, writing: review; N.Z: validation, resources, writing: review; I.A: writing: review and editing.

**Funding** Open access funding provided by Mälardalen University. This research work has received funding from: the European Union's Horizon 2020 research and innovation program under grant agreement No. 957212; the ECSEL Joint Undertaking (JU) under grant agreement No 101007350. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Sweden, Austria, Czech Republic, Finland, France, Italy, Spain; the Swedish Innovation Agency (Vinnova) through the SmartDelta and NRPCES projects.

**Data Availability** TIGER+ is available at https://github.com/MuhammadNoumanZafar/TIGERPlus based on GraphWalker (GW) https://github.com/GraphWalker/graphwalker-project/wiki. However, the case studies and source code for the test script generation are specific to Alstom Rail AB, Sweden, and are not included in the repository due to NDA obligations.

## Declarations

**Conflicts of Interest/Competing Interests** On behalf of all authors, the corresponding author states that there is no conflict of interest.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

## References

Acree, A. T., DeMillo, R. A., Budd, T., Lipton, R. J., & Sayward, F. G. (1979). *Mutation Analysis*. Georgia Institute of Technology, USA: School of Information and Computer Science.

Albahari, J., & Albahari, B. (2008). *LINQ Pocket Reference: Learn and Implement LINQ For*. NET Applications. O'Reilly Media.

Arefin, S. S., Hemmati, H., & Loewen, H. W. (2018). Evaluating specification-level mc/dc criterion in model-based testing of safety critical systems. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice* (pp. 256–265), ICSE-SEIP '18, Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3183519.3183551

Black, R. (2014). Guide to the Istqb advanced certification as an advanced test manager. In *Advanced Software Testing* (Vol. 2). Rocky Nook Inc.

Cai, X., & Lyu, M. R. (2005). The effect of code coverage on fault detection under different testing profiles. In *Proceedings of the 1st International Workshop on Advances in Model-based Testing* (pp. 1–7).

Chen, Y., Linder, S., & Wigstein, J. (2019). An approach of creating component design specification for safety-related software in railway. In *2019 Annual Reliability and Maintainability Symposium (RAMS)* (pp. 1–4). IEEE.

Coutinho, A. E. V. B., Cartaxo, E. G., & Machado, P. D. D. L. (2016). Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing. *Software Quality Journal,24*, 407–445.

Crama, Y., & Hammer, P. L. (2011). *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press.

Cruciani, E., Miranda, B., Verdecchia, R., & Bertolino, A. (2019). Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 419–429). https://doi.org/10.1109/ICSE.2019.00055

Cruciani, E., Miranda, B., Verdecchia, R., & Bertolino, A. (2019). Scalable approaches for test suite reduction. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 419–429). IEEE.

Deneke, A., Assefa, B. G., & Mohapatra, S. K. (2022). Test suite minimization using particle swarm optimization. *Materials Today: Proceedings,60*, 229–233.

Desolda, G., Ardito, C., & Matera, M. (2017). Specification of complex logical expressions for task automation: an eud approach. In *End-User Development: 6th International Symposium* (pp. 108–116), IS-EUD 2017, Eindhoven, The Netherlands, June 13-15, 2017, Proceedings 6. Springer

Enoiu, E. P., Sundmark, D., Čaušević, A., Feldt, R., & Pettersson, P. (2016). Mutation-based test generation for plc embedded software using model checking. *IFIP International Conference on Testing Software and Systems* (pp. 155–171). Springer.

Fraser, G., & Wotawa, F. (2007). Redundancy based test-suite reduction. *International Conference on Fundamental Approaches to Software Engineering* (pp. 291–305). Springer.

Gupta, N., Sharma, A., & Pachariya, M. K. (2019). An insight into test case optimization: Ideas and trends with future perspectives. *IEEE Access,7*, 22310–22327. https://doi.org/10.1109/ACCESS.2019.2899471

Hamlet, R. (1994). Random testing. *Encyclopedia of Software Engineering,2*, 971–978.

Hayhurst, K. J. (2001). A Practical Tutorial on Modified Condition/decision Coverage. DIANE Publishing.

Hemmati, H., Arcuri, A., & Briand, L. (2013). Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM),22*(1), 1–42.

Homès, B. (2013). *Fundamentals of Software Testing*. John Wiley & Sons.

Jaccard, P. (1912). The distribution of the flora in the alpine zone. *New Phytologist,11*(2), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

Jiawei, H., Micheline, K., & Jian, P. (2011). Data mining concepts and techniques third edition. *The Morgan Kaufmann Series in Data Management Systems*, *5*(4), 83–124

Jones, J. A., & Harrold, M. J. (2003). Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering,29*(3), 195–209.

Kalaee, A., & Rafe, V. (2019). Model-based test suite generation for graph transformation system using model simulation and search-based techniques. In *Information and Software Technology* (Vol. 108, pp. 1–29). https://doi.org/10.1016/j.infsof.2018.12.001

Kochhar, P. S., Thung, F., & Lo, D. (2015). Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 560–564). IEEE.

Koitz-Hristov, R., Stracke, L., & Wotawa, F. (2022). Checked coverage for test suite reduction: is it worth the effort? In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test* (pp. 6–16)

Kramer, A., & Legeard, B. (2016). *Model-based Testing Essentials-guide to the ISTQB Certified Model-based Tester: Foundation Level*. John Wiley & Sons.

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). *Introduction to Combinatorial Testing*. CRC Press.

Kushik, N., Yevtushenko, N., & López, J. (2021). Testing against non-deterministic fsms: A probabilistic approach for test suite minimization. In *IFIP International Conference on Testing Software and Systems* (pp. 55–61). Springer.

Kushik, N., Yevtushenko, N., & L'opez, J. (2023).Probabilistic approach for minimizing checking sequences for non-deterministic fsms. In *IFIP International Conference on Testing Software and Systems* (pp. 237–243). Springer

Li, X., & Liu, B. (2014). Rule-Based Classification. Citeseer.

Liu, X., & Yu, P. (2022). Randoop-tsr: Random-based test generator with test suite reduction. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware* (pp. 221–230)

Mali, K., & Mitra, S. (2005). Symbolic classification, clustering and fuzzy radial basis function network. *Fuzzy Sets and Systems,152*(3), 553–564.

Malley, B., Ramazzotti, D., & Wu, J. T.-y. (2016). Data pre-processing. Secondary analysis of electronic health records, 115–141.

Maslar, M. (1996). Plc standard programming languages: Iec 1131-3. In *Conference Record of 1996 Annual Pulp and Paper Industry Technical Conference* (pp. 26–31). IEEE

Meijer, E., Beckman, B., & Bierman, G. (2006). Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (pp. 706–706).

Mondal, D., Hemmati, H., & Durocher, S. (2015). Exploring test suite diversification and code coverage in multi-objective test case selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)* (pp. 1–10). https://doi.org/10.1109/ICST.2015.7102588

Nikolik, B. (2006). Test diversity. *Information and Software Technology,48*(11), 1083–1094.

Nuh, J. A., Koh, T. W., Baharom, S., Osman, M. H., Babangida, L., Letchmunan, S., & Kew, S. N. (2023). Diversity-based test case prioritization technique to improve faults detection rate. *International Journal of Advanced Computer Science and Applications*, *14*(6),

Oh, Y., Yoo, J., Cha, S., & Son, H. S. (2005). Software safety analysis of function block diagrams using fault trees. *Reliability Engineering & System Safety,88*(3), 215–228.

Osei-Bryson, K.-M. (2014). Overview on decision tree induction. In *Advances in Research Methods for Information Systems Research: Data Mining, Data Envelopment Analysis, Value Focused Thinking* (pp. 15–22)

Pan, R., Ghaleb, T. A., & Briand, L. (2023). Atm: Black-box test case minimization based on test code similarity and evolutionary search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 1700–1711). IEEE.

Pan, R., Ghaleb, T. A., & Briand, L. C. (2024). Ltm: Scalable and black-box similarity-based test suite minimization based on language models. *IEEE Transactions on Software Engineering,50*(11), 3053–3070. https://doi.org/10.1109/TSE.2024.3469582

Qin, B., Xia, Y., Prabhakar, S., & Tu, Y. (2009). A rule-based classification algorithm for uncertain data. In *IEEE 25th International Conference on Data Engineering* (pp. 1633–1640). IEEE.

Rothermel, G., Harrold, M. J., Von Ronne, J., & Hong, C. (2002). Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability,12*(4), 219–249.

Rouvroye, J. L., & Bliek, E. G. (2002). Comparing safety analysis techniques. *Reliability Engineering & System Safety,75*(3), 289–294.

Sachtleben, R., & Peleska, J. (2022). Effective grey-box testing with partial fsm models. *Software Testing, Verification and Reliability,32*(2), 1806.

Shin, D., Jee, E., & Bae, D.-H. (2012). Empirical evaluation on fbd model-based test coverage criteria using mutation analysis. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 465–479). Springer.

Shu, T., He, Z., Yin, X., Ding, Z., & Zhou, M. (2024). Model-based diversity-driven learn-to-rank test case prioritization. *Expert Systems with Applications,255*, 124768.

Spillner, A., & Linz, T. (2021). Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant dpunkt. verlag.

Stubblebine, T. (2007). Regular expression pocket reference: Regular expressions for perl, Ruby, PHP, Python, C, Java And. NET. "O'Reilly Media, Inc".

Utting, M., & Legeard, B. (2010). *Practical Model-based Testing: a Tools Approach*. Elsevier.

Woodward, M. R. (1993). Mutation testing—its origin and evolution. *Information and Software Technology,35*(3), 163–169.

Yao, X., Harman, M., & Jia, Y. (2014). A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 919–930).

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability,22*(2), 67–120.

Zafar, M. N. (2022). Model-based system testing of safety-critical embedded software. http://www.es.mdu.se/publications/6639-

Zafar, M. N., Afzal, W., & Enoiu, E. P. (2023). An empirical evaluation of system-level test effectiveness for safety-critical software. In *18th International Conference on Evaluation of Novel Approaches to Software Engineering*. http://www.es.mdu.se/publications/6638-

Zafar, M. N., Afzal, W., Enoiu, E. P., Haider, Z., Singh, I. (2024). Optimizing model-based generated tests: Leveraging machine learning for test reduction. In *The 20th Workshop on Advances in Model Based Testing*. http://www.es.mdu.se/publications/6959-

Zafar, M. N., Afzal, W., Enoiu, E. P., Stratis, A., & Sellin, O. (2021). A model-based test script generation framework for embedded software. In *The 17th Workshop on Advances in Model Based Testing*. http://www.es.mdu.se/publications/6172-

Zafar, M. N., Afzal, W., Enoiu, E. P., Stratis, A., Arrieta, A., & Sagardui, G. (2021). Model-based testing in practice: An industrial case study using graphwalker. In *Innovations in Software Engineering Conference 2021*. http://www.es.mdu.se/publications/6101-

Zhong, H., Zhang, L., & Mei, H. (2008). An experimental study of four typical test suite reduction techniques. *Information and Software Technology,50*(6), 534–546.

**Publisher"s Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Authors and Affiliations

**Muhammad Nouman Zafar[1] · Wasif Afzal[1] · Eduard Paul Enoiu[1] · Zulqarnain Haider[2] · Nedim Zaimovic[2] · Iram Arshad[3]**

✉ Muhammad Nouman Zafar
muhammad.nouman.zafar@mdu.se

Wasif Afzal
wasif.afzal@mdu.se

Eduard Paul Enoiu
eduard.paul.enoiu@mdu.se

Zulqarnain Haider
zulqarnain.haider@alstomgroup.com

Nedim Zaimovic
nedim.zaimovic@alstomgroup.com

Iram Arshad
i.arshad@research.ait.ie

[1] Mälardalen University, Vasteras, Vastmanland, Sweden

[2] Alstom Rail AB, Vasteras, Vastmanland, Sweden

[3] Technological University of Shannon, Midland, Midwest, Ireland