Optimizing Model-based Generated Tests: Leveraging Machine Learning for Test Reduction

Muhammad Nouman Zafar*, Wasif Afzal*, Eduard Paul Enoiu*, Zulqarnain Haider†, Inderjeet Singh†
*Mälardalen University, Sweden
{muhammad.nouman.zafar, wasif.afzal, eduard.enoiu}@mdu.se
†Alstom Rail AB, Sweden
{zulqarnain.haider, singh.inderjeet}@alstomgroup.com

***Abstract*—Several studies have shown Model-based Testing (MBT) as an efficient technique for generating fault-effective test cases. However, the automatic generation of test cases is compromised with redundant test cases providing no additional value to the coverage or fault detection effectiveness while impacting test execution efficiency, especially, in a dynamic development environment where providing timely feedback is crucial. These redundant test cases need to be discarded to minimize the test suite size and their effect on the execution cost and efficiency of a test suite. Reducing a test suite becomes challenging for black box testing at the system level when no information regarding the coverage and fault detection effectiveness of the test suite exists. Hence, in this paper, we have presented a test suite optimization approach leveraging different machine learning algorithms, a greedy algorithm, and a similarity measure. The proposed approach generates a reduced test suite by identifying and eliminating redundant test cases from an MBT-generated test suite while having minimal impact on the fault detection rate. We have also performed a comparative evaluation of the optimized test suites with the MBT-generated and manually created test suites in terms of fault detection effectiveness and test execution efficiency using an industrial case study from Alstom Rail AB, Sweden. The results show a significant reduction of 85% to 92% in the size of the test suite. Moreover, we also found the test execution time of the optimized test suite equivalent to the manually created tests and a fault detection rate within the range of 95% to 100% for all test suites under observation.**

***Index Terms*—Model-based Testing, Test Suite Reduction, Machine Learning, System Level Test**

I. Introduction

Model-based testing (MBT) [1] is one of the automated testing techniques that can generate test artifacts for the verification and validation of different software. It uses an explicit model, created in a modeling language such as Finite State Machine (FSM), Finite Automata, etc., representing the concise behavior of the Software Under Test (SUT) to generate abstract test cases by traversing through the model. These abstract test cases can be automatically transformed into concrete test scripts that can be executed on a 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 and bugs in software. The generated test suite contains a wide range of inputs/outputs and their interactions within the system representing the potential possible scenarios covering a substantial part of the implemented code including redundant ones without adding additional value to reveal new faults [2]. The redundant test cases can be categorized as identical and non-identical test cases [2]. 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 constraints but cover similar scenarios of the system's functionality. The presence of redundant test cases not only affects the scalability of a test suite but also poses a substantial challenge, particularly in industrial settings where cost and resource efficiency is the key. Hence, such types of test cases need to be identified and discarded to increase test efficiency and resource utilization in a dynamic development environment. Previous attempts to mitigate this challenge through various techniques, including the use of machine learning for test suite reduction, have often resulted in a trade-off, a decrease in the fault detection rate [3] [4] [5]. Moreover, this problem becomes challenging at system-level testing when the system is considered a black box, the source code of the SUT is inaccessible and no other information exists related to the test suite e.g., coverage, fault detection effectiveness, etc. The problem in our context can be defined as:

Problem Statement: "Given an FSM-based generated test suite *(TS)* of a fixed size *n* covering a set of requirements *(R)* and having fault detection effectiveness *(F)*, our goal is to reduce the size of a test suite by identifying and removing redundant test cases while having a minimum impact on fault detection effectiveness of the test suite."

Since the test suite generated from an FSM model does not contain any information about fault detection effectiveness prior to execution on a SUT or coverage of implemented code, a surrogate measure is required. In the literature, different studies (e.g., [6], [7], [8], etc.) have shown a moderate to strong correlation between diversity, coverage, and fault detection effectiveness. Diversity in a test suite ensures a wide range of test scenarios, coverage ensures the significant part of a system is tested, and, together, these can contribute toward the effectiveness of a test suite in terms of fault detection [9] [10]. Hence, in this paper, we proposed, implemented, and integrated a tool with our previously developed tool i.e., Model-based Test script GenEration fRamework (TIGER) [11], and

DOI 10.1109/ICSTW60967.2024.00020

named it TIGER+ [1]. TIGER+ reduces the size of a MBT-generated test suite by identifying and eliminating redundant test cases based on Modified Condition Decision Coverage (MCDC) and a similarity measure inspired by the Jaccard index [12]. It also uses different machine-learning techniques for the extraction, pre-processing, and classification of the test data. We have also validated the proposed approach by generating two subsets of MBT-generated test suite and performing a comparative evaluation of generated subsets with a MBT-generated test suite and a test suite created manually. In our evaluation, we employed mutation analysis to measure the fault detection effectiveness and execution efficiency of the test suites using an industrial case study of the Train Control Management System (TCMS) from Alstom Rail AB, Sweden.

The results of the experimental evaluation show that

- TIGER+ improved the execution efficiency of the MBT-generated test suite by reducing the test suite size of approx. 85% and 92% when optimized at the highest and lowest levels, respectively.
- Test suite optimized by the TIGER+ can be used to achieve a higher fault detection rate within the range of 95% to 100% equivalent to MBT-generated and manually created test suites.

The rest of the paper is organized as follows: Section II briefly discusses the related studies, Section III presents a detailed description of the proposed approach, Section IV shows its experimental evaluation including a description of the SUT, development and execution of test suites, mutation analysis, experimental setup, and results, Section V presents a discussion on the results and threats to the validity of this study is discussed in Section VI followed by conclusion and future work in Section VII.

II. Related Work

There exist different studies (i.e, [13],[14], [15], [16], [17], [18], [19]) that have proposed different techniques and approaches in various contexts to generate an adequate size test suite or to reduce the size of a generated test suite. A summary of these studies is provided below.

Fraser et al. [13] 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 between the test cases. The experimental evaluation shows the improved quality of test cases while having minimal effect on fault detection effectiveness. Hemmati et al. [14] 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 have 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-based generated test suite reduction is presented in [15]. The empirical results show that the choice of distance function has minimal effect on the test suite size, however, has shown significant effects on the fault coverage and stability of the test suite. Kushik et al. [16] 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 [17] for test suite minimization by shortening the length of the combinations of inputs representing the checking sequence. Sachtleben et al. [18] proposed a test generation algorithm for an FSM-based model to generate an adequate size 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. Pan et al. [19] 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.

III. Proposed Approach

The proposed approach is intended to provide a reduced but fault-effective subset of a test suite generated from an FSM model. Figure 1 represents 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
   - 3.1 Data transformation
   - 3.2 Removal of identical test cases
   - 3.3 Imputing missing parameters/values
4) Classification of test cases
   - 4.1 Classification of test cases based on Guard Conditions
   - 4.2 Sub-classification of test cases based on Group Conditions
5) Test suite reduction
   - 5.1 Test reduction using a greedy algorithm
   - 5.2 Similarity-based test reduction

A. Model-based test generation

We used an open-source model-based testing tool, GraphWalker (GW), to generate abstract test cases from an explicit FSM model delineating the behavior of a SUT. An FSM model consists of nodes depicting the states of a system and edges representing the transitions from one state to another based on guard conditions. Guard conditions are the logical expressions

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

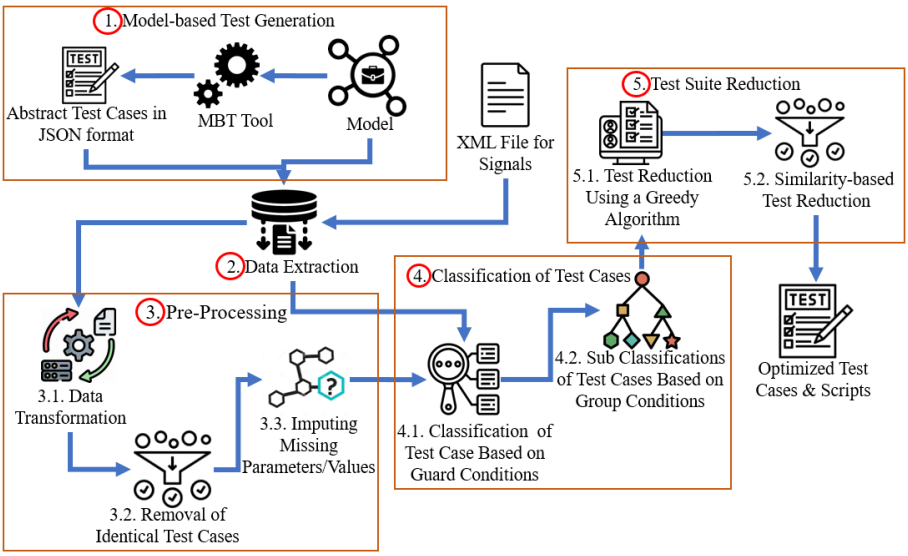

Fig. 1. A machine learning-based test reduction approach

embedded in an FSM model constructing functional and non-functional behavior of a model corresponding to a SUT. Moreover, an FSM model can also store information to map requirements with model elements (i.e., nodes and edges) supporting requirement traceability [20]. The generation of an effective and complete test suite requires conformance of an FSM model to the requirements of a SUT [1].

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 Figure 2. The abstract test cases generated through an FSM model are conventionally the traversed paths instantiated with different event parameters (test data) encoded as strings. Similarly, the JSON file generated by GW contains 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 a model constituting 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.

B. Data extraction

To optimize the abstract test cases, all the information from the JSON file (e.g., actions represented by different parameters and their respective values, sequence of model elements, etc.) 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 [2] of a signal interacting

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


```
{
  "modelName": "VarA",
  "data": [
    { "varD": "true" },
    { "varC": "true" },
    { "varB": "true" },
    { "varA": "true" }
  ],
  "currentElementID": "1eacd4d5-3d6c-4624-92f4-65bc5e6d9a5c",
  "currentElementName": "A",
  "properties": [
    { "x": 221.04553986268414 },
    { "y": 326.0896089137364 }
  ]
} {
  "modelName": "VarA",
  "data": [
    { "varD": "true" },
    { "varC": "true" },
    { "varB": "true" },
    { "varA": "true" }
  ],
  "currentElementID": "19e19385-9a02-45f7-b780-162e3c53f7a5",
  "currentElementName": "No",
  "actions": [ { "Action": "varA=false;" } ],
  "properties": []
} {
```


Fig. 2. An example of an abstract test step generated by GW in JSON format

with different components of an embedded system, and to generate concrete or executable test cases/scripts, it requires 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 a 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 of an XML file containing such information and extracted the required information from it [11]. The extraction process also ensures the preservation of the 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

```
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; }
}
```

Fig. 3. A logical data model for abstract test cases

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

Fig. 4. A logical data model for guard conditions and associated requirements

data (i.e., a logical data model similar to the format of a JSON file (Figure 2) consisting of a 'RootObject' comprising 'modelName', 'currentElementID', 'currentElementName', an array of 'actions', and 'properties' as shown in Figure 3 and a logical data model consists of guard condition and a list of requirements associated with each guard condition as shown in Figure 4).

C. Pre-processing of abstract test cases

The JSON file generated from a model contains abstract test data as well as information about a model and each traversed element such as 'modelName', 'CurrentElementID', 'properties', etc. that can be considered as noise diluting the test cases with insignificant information. Moreover, to select MCDC-adequate test cases, we require instantiated test data. Hence, to extract relevant test data, transform it into a scenario-based test case format, and for test optimization, a pre-processing of generated abstract test cases is required. The pre-processing in the proposed approach consists of three main steps: (3.1) data transformation, (3.2) removal of identical test cases, and (3.3) imputing the missing parameters and corresponding values in each test case.

1) Data transformation: The test steps in each generated test case can contain more than one event parameter and their respective values encoded in a string, which needs to be identified and extracted to create inputs, expected outputs, and timing constraints of 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 Figure 5. We have also defined a data transformation algorithm that splits the string in the list of 'Actions' to extract parameters and their values based on delimiters (i.e., ';' and '='), finds the data type of each parameter from the list of signals extracted from the XML

```
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; }
}
public class TestActions
{

    public string ActionDescription { get; set; }
    public string ActionVariable { get; set; }
    public string ActionValue { get; set; }
}
public class ExpectedResult
{
    public string ResultDescription { get; set; }
    public string ResultVariable { get; set; }
    public string ResultValue { get; set; }
}
```

Fig. 5. A logical data model of a test suite

file, and dynamically populates the defined logical data model by creating test actions or expected results.

2) Removal of identical test cases: A test suite generated from a behavioral model using random generation also contains duplicate or identical test cases, providing no additional value to reveal new faults. Moreover, each test case consists of different test steps altering the values of a limited number of parameters (based on traversed model elements) to validate the output of a system. These kinds of 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 a 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 as well as default values of 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 definition of the Jaccard index as a similarity measure to remove the identical test cases but also preserves the 'Reset' test case generated from a model. The Jaccard index is also known as a similarity coefficient, which is used to compare the similarity between two unordered sets. It is a relative measure between 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 test suite to add distinct test cases. It starts by adding the first test case to the distinct test suite 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). The algorithm also adds the 'Reset' after each test case to produce the resultant test suite.

3) Imputing missing parameters/values: As described above, each test case generated through an FSM model consists of test steps forcing or altering the values of specific parameters without retaining the default or initial values of the remaining parameters of a system. We can refer to such parameters as missing parameters/values in a test case. Hence, to create the requirement traceability matrix by performing the rule-based classification of test cases based on satisfied guard conditions (Section III-D), we are required to identify and impute such missing parameters and their values in each test case. To achieve this goal and retain the missing parameter/values from the 'Reset' state, the proposed approach uses an imputation machine-learning 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. We have used Language Integrated Query (LINQ) [21] in the implementation to find the first or default 'Reset' test case in the test suite.

After the transformation of 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. It identifies the missing parameters in a test case based on the parameters or signals provided in the XML file, 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, and adds the test action to a corresponding test case.

D. 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. The benefit of this step is twofold; the traceability provided by RTM can be used as a measure to ensure that each requirement of a SUT has an adequate number of test cases, and it will also assist in optimizing and selecting an MCDC-adequate abstract test suite.

1) Classification of test cases based on guard conditions: We used a rule-based classification [22] methodology to create the RTM and groups of test cases based on extracted guard conditions. Rule-based classification is a machine-learning technique that classifies the features of data based on 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 [23]) or can also be defined by domain experts based on prior knowledge of a system or data [24]. Similarly, in a model, guard conditions represent the rules associated with the requirements of a system and so additional conditions to refine the system's behavior and applicability of these rules. Hence, we used guard conditions as the rules to create an RTM and

RQ1:

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

RQ2:

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

RQ3:

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. 6. An example of requirements in Given-When-Then format

classify the test cases into different groups based on guard conditions. For instance, let's assume a system that shows relevant documents to a user after validation based on two validation processes (i.e., validation of a request and validation of a user). Figure 6 represents three functional requirements (i.e., RQ1, RQ2, and RQ3) of the system written in a Given-When-Then format defining its behavior. In this context, guard conditions in a model explicitly representing the behavior or predicates/decisions in the implementation of the system based on the above-mentioned requirements, can be specified as follows:

$$(varA == true)\&\&(varB == true||varC == true) \quad (1)$$

$$(varA == false)||(varB == false\&\&varC == false) \quad (2)$$

Where guard condition 1 represents a combined rule of RQ1, RQ2, and RQ3, and guard condition 2 is an illustration of an obvious requirement representing the behavior of a system if either of the validation processes fails. Hence, all the test cases satisfying a guard condition will also cover the respective requirements of the system.

The implemented algorithm takes the distinct test suite generated in the previous step as input, along with the guard conditions and associated requirements extracted in the data extraction step. Then it evaluates each test case against each guard condition to create an RTM, as well as groups of test cases based on guard conditions. The logical data model for RTM is shown in Figure 7. To evaluate test cases against guard conditions, we have used the dynamic expression function [25] 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. Table I shows an example of generated groups and Table II illustrates an example of classification of test cases in different groups.

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,

```
class RTMModel{
    public List<string> requirements { get; set; }
    public string guard { get; set; }
    public List<Testcase> testcases { get; set; }
}
```

Fig. 7. Logical data model for requirement traceability matrix

TABLE I
EXAMPLE OF GENERATED GROUPS BASED ON GUARD CONDITIONS

Group ID	Guard Condition
1	$(varA == true)\&\&(varB == true \| \| varC == true)$
2	$(varA == false) \| \| (varB == false \&\& varC == false)$

the guard condition 1 represents the behavior of the system covering all the requirements shown in Figure 6. However, if we divide it into group conditions, the first group condition (i.e., $varA == true$) represents RQ1, the second group condition (i.e., $varB == true2 \| \| varC == true$) illustrates RQ2 and a combination of both group conditions with an AND operator depicts RQ3 of the system presented in Figure 6. Hence, in our proposed approach, we further classified the test cases into subgroups based on the 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 [26]. 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 [27]. 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 in each guard condition using the regular expression library [28] in the .Net framework and evaluated each test case in a corresponding group to create subgroups. We created a logical data model to store the list of sub-classified test cases. It contains a 'guard', a list of 'requirements', and an associated list of 'group conditions' comprising a matrix of 'group ID', 'group condition', and a list of 'test cases' satisfying each group condition as shown in Figure 8. Table III shows an example of subgroups, whereas Table IV illustrates the classification of test cases in identified subgroups.

E. Test suite reduction

We defined a greedy and naïve comparison-based approach to optimize the generated test suite. The greedy approach is responsible for the selection of MCDC-adequate test cases for each corresponding group and subgroup created in previous steps, whereas the naive comparison approach is responsible for the identification of diverse test cases and performing

TABLE II
AN EXAMPLE OF CLASSIFIED TEST CASES INTO GROUPS

Group ID	Test case ID	Inputs			Outcome
		varA	varB	varC	
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)

```
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; }
}
```

Fig. 8. A logical data model for sub-classification

similarity-based test reduction. MCDC is a coverage criterion recommended by various standards (i.e., EN 50128, EN 50657 [29]) to validate the safety critical systems. It is a stricter criterion than a decision or branch coverage that requires the coverage at condition level [30]. It ensures that each condition in a decision has shown an independent effect on the outcome of the decision [31]. Hence, we implemented a greedy algorithm guided by MCDC to select the MCDC-adequate test suite and then used the naïve comparison algorithm similar to the algorithm described in Section III-C2 to identify and remove similar test cases within each subgroup.

TABLE III
AN EXAMPLE OF GENERATED SUBGROUPS

Subgroup ID	Group Condition
1	$(varA == true)$
2	$(varB == true \| \| varC == true)$
3	$(varA == false)$
4	$(varB == false \&\& varC == false)$

TABLE IV
AN EXAMPLE OF CLASSIFIED TEST CASES INTO SUBGROUPS

Group ID	Subgroup ID	Test case ID	Inputs			Outcome
			varA	varB	varC	
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

TABLE V
AN EXAMPLE OF THE RESULTANT TEST SUITE AFTER GREEDY APPROACH

Group ID	Subgroup ID	Test case ID	Inputs			Outcome
			varA	varB	varC	
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

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 a respective decision. We defined a greedy algorithm that takes a test case from a subgroup of the test suite and creates different variants by inverting the value of a test action (excluding test action containing the retained value from 'Reset') in each variant corresponding to conditions in the group condition. It also evaluates all the variants against the group condition and observes the effect on the decision. If any of the variants show an effect on the outcome of the respective group condition (i.e., any variant of a test case does not satisfy the group condition), it adds the test case to the resultant test suite, ensuring that the condition has its independent effect on the decision. For example, let's assume a test case with 'Group ID' 1, 'Subgroup ID' 1, and 'Test Case ID' 1 given in Table IV. The algorithm will invert the value of 'varA' from true to false, evaluate it against the group condition (i.e., $varA == true$), and add it to the resultant test suite as it will not satisfy the group condition. On the other hand, in the case of a test case with 'Group' 1, 'Subgroup' 2, and 'Test Case ID' 1, it will create two variants of the test cases by inverting the value of 'varB' in variant 1 and inverting the value of 'varC' in variant 2, evaluate both variants against the corresponding group condition (i.e., $varB == true \| varC == true$) and will remove the test case from the resultant test suite as both variants will show no effect on the outcome of the group condition. Table V illustrates an example of the resultant test suite.

F. 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 III-C2 but limited the comparable parameters/test actions to the test cases corresponding to the conditions in a group condition. For instance, in Table V all the test cases in the subgroup with 'Subgroup ID' 1 contain the same values for 'varA' involved in the respective sub-condition (i.e., $varA == true$). Hence, it will add the first test cases from the subgroup to the reduced test suite and will remove the remaining. Table VI represents an example of a reduced test suite generated by the proposed approach.

TABLE VI
AN EXAMPLE OF THE REDUCED TEST SUITE

Group ID	Subgroup ID	Test case ID	Inputs			Outcome
			varA	varB	varC	
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

IV. EXPERIMENTAL EVALUATION

This section presents a brief description of the case study of a fire detection system which is a sub-system of TCMS developed at Alstom, the creation of test suites, an overview of the experimental setup, and the results of the comparative evaluation between the test suites generated by MBT and created manually.

A. SUT

In this paper, we have considered a fire detection subsystem of a TCMS for MOVIA [3], which is a vehicle family of various metro trains developed at Alstom, as a case study. TCMS is a complex and centralized distributed control system designed for controlling and coordinating various subsystems such as fire detection systems, traction, doors, etc. It is also responsible for ensuring the safe and efficient operations of regular and safety-critical functions of these subsystems and facilitates real-time monitoring of a train. It utilizes various networks e.g., Ethernet Consist Network (ECN), Multi-function Vehicle BUS (MVB), etc. to establish communication between different subsystems and devices [32]. The Modular Input/Output-Safe (MIO-S) and Central Control Unit (CCU-S) devices are connected through these network to control and manage the safety-critical functions of a train. 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) [32]. It uses two instances of FDCUs interconnected with and 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 report 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.

B. Development and Execution of Test Suites

To comparatively evaluate the effectiveness and efficiency of the test suite generated from the proposed approach, we used a test suite (i.e., test cases and scripts) created by a tester at Alstom for the selected subsystem. For MBT, we analyzed the requirement and test specification to completely understand the system and utilized GraphWalker [4] (GW), an open source

[3]https://www.sgtrains.com/train-r151.html
[4]https://github.com/GraphWalker/graphwalker-project/wiki

MBT tool, to create an FSM model representing the explicit behavior of the system 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) [11], and named it TIGER+, to reduce the test suite size and to generate concrete test scripts. The test scripts are then executed on the SUT to produce test verdicts. A detailed description of the activities for the development of the test suites is given in the following subsections.

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 a 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 [33]. 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.

2) MBT-generated Test Suites and their Execution: We utilized the GW studio version to create an FSM model of the selected subsystem and the TIGER+, which uses the command line version of GW, to generate three versions of test suites (i.e., MBT-generated, optimized at the highest level, optimized at the lowest level) using a random generation algorithm and 100% edge coverage criterion. 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 in Section III-D2, 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. We transformed the abstract test cases into concrete test scripts by providing an XML file to TIGER+ containing 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 and executed on the SUTs to produce test verdicts.

C. Mutation Analysis

Mutation analysis [34] is an evaluation technique used to assess and analyze the fault detection capabilities of a test suite, especially when a system under observation has no known faults. 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 [35]. 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 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 affects the outcome (i.e., strong mutation) or an internal oracle in which fault in a program does not affect the outcome (i.e., weak mutation). A mutation score is a ratio between the total number of mutants and the number of mutants killed by a test suite representing its fault detection rate.

D. Experimental Setup

In our cases, the selected subsystem is 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-standard 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 IV-C and performed similar experimentation as our previous paper [33]. We created several mutants of FBD programs by selecting six mutation operators provided in literature [36] [37] [38] based on safety-critical industrial cases study and FBD specific faults i.e., Logic Block Deletion Operator, Logic Block Insertion Operator, Logic Block Replacement Operator, Logic Block Replacement Operator-Improved, Negation Insertion Operator, and Value Replacement Operator to mimic programmers' mistakes. We have discarded the Time Block Replacement Operator as mutants created by these operators seem to be stubborn mutants [33]. After the creation

of FBD mutants, we utilized Alstom-specific compiling tools to generate different builds of virtual trains which are the 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., SUT with no known fault and its mutated versions) and generate test verdicts at the Software-in-the-Loop level.

E. 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.

1) Fault detection effectiveness of test suites: To evaluate the test cases in terms of fault detection effectiveness, we injected faults in the original FBD program and created 40 mutants based on the selected operator as mentioned in Section IV-D. We also calculated the mutation score of each test suite as shown in Table VII by executing the test suites on the SUT and analyzing the test verdicts. It is also important to mention here that the selected subsystem consisted of similar requirements which were implemented by generating different instances of same FBD programs [33], hence, limited the creation of mutants for this study. Moreover, we have used strong mutation to calculate the mutation score at the system level and only considered the non-equivalent mutants in our results by excluded the equivalent and stubborn mutants through manual analysis of the test results.

The results of the experimentation show that the TIGER+ was able to reduce the size of the MBT-generated test suite by approx. 85% at the highest-level optimization and approx. 92% at the lowest-level optimization yielding 46 and 22 test cases out of 307, respectively. We also observed that the test suite optimized at the highest level provided the highest mutation score of 100%, MBT-generated delivered a 97.5% mutation score, whereas the test suites optimized at the lowest level and created manually were able to achieve a 95% mutation score by uncovering different interaction faults. We also 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 as it contains a subset of the MBT-generated test cases. 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. Whereas in the case of the MBT-generated test suite, it could not uncover a fault that was detected by both optimized test suites and, hence, provided 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. On the other hand, in the case of the optimized test suites, each test case forces the value of each signal involved and verifies the expected outputs and the 'Reset' test case is only executed at the end of the test scripts. Hence, optimized test suites succeeded in creating different possible scenarios to uncover that specific fault. Moreover, a similar structure of test cases was observed in the case of the manually created test suite that was also able to uncover that specific fault.

2) Test execution efficiency of test suites: For the test execution efficiency of the test suites, we reported the approximate time required by 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 VII. The MBT-generated test suite required the highest execution time of approx. 21 minutes if executed on SUT with no known faults and from 120 to 140 minutes if executed on the mutated SUTs. Whereas in the case of manually created test suites and test suites optimized at the highest level and lowest level 7, 8, and 7 minutes and 10 to 12, 13 to 17, and 10 to 13 minutes of approx. execution time was required 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.

V. Discussion

Our results regarding the effectiveness of the TIGER+ show over 85% of test suite reduction in both cases i.e., optimization at the highest and lowest level. However, during the thorough analysis of the MBT-generated and reduced test suites as well as the FSM model, we observed that the reduction rate of the TIGER+ is dependent on the number of test cases in the MBT-generated test suite generated through random walks, the complexity of guard conditions depicting the behavior of the system, and the size of the FSM model. For instance, generating different test suites in multiple attempts from a random generation algorithm can produce a different number of test cases in each test suite including redundant test cases. Similarly, the size of the FSM model and the complexity of the guard condition can yield different numbers of groups during classification affecting the size of a reduced test suite. Hence, we believe, the TIGER+ can effectively be used to detect and eliminate the redundant test cases in a test suite but these three factors can affect the reduction rate of the proposed approach. However, the further analysis of these factors and the significance of their effect on fault detection is another future work.

We also found the results of mutation analysis in the case of MBT-generated and manually created test suites similar to

TABLE VII
COMPARATIVE MUTATION ANALYSIS BETWEEN MBT-GENERATED TEST SUITE, OPTIMIZED TEST SUITES, AND MANUALLY CREATED TEST SUITE

Test suites	Number of test cases	Mutants killed	Mutants alive	Mutation score	Approx. execution time on SUT in minutes	
					No known faults	Mutated SUTs
Manual	18	38	2	95%	7	10-12
MBT-generated (Original)	307	39	1	97.5%	21	120-140
Optimized at highest level	46	40	0	100%	8	13-17
Optimized at lowest level	22	38	2	95%	7	10-13

our previous study [33] in terms of detected faults regardless of the size of the MBT-generated test suite. These results ratify that a higher fault detection rate can be achieved by utilizing MBT tools. An in-depth analysis of the mutation score and undetected faults shows that the MBT-generated and manually created test suites were only unable to uncover different interaction faults injected based on the Logic Block Replacement Operator, and hence, provided a relatively lower mutation score than the test suite optimized at the highest level. Whereas, the optimized test suite at the lowest level was found least effective in detecting faults injected based on Logic Block Replacement and Logic Block Insertion Operator.

VI. VALIDITY THREATS

This section discusses the threats to the validity of this study along with the strategies we used to eliminate them.

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 with requirements. The modeling of the system is a manual process that requires complete knowledge of the domain, process, and system. The requirements at Alstom are written in natural language using the Given-When-Then format and there exists a chance that misinterpretation of these requirements can impact the conformance of the model and generated test suite. However, to mitigate this threat, we have spent a substantial amount of time for understanding the requirements and the system through carefully analyzing the requirements and test specifications. We have also iteratively created the model by getting continuous feedback on its correctness from a testing team at Alstom.

The threats that can influence the reliability and external validity of the study include the modeling notation, generation algorithm, human experience of modeling, size of the subsystem, and particularities of the test generation tool, MBT model, and test suite specific to Alstom's testing environment and system. The proposed approach and tool implemented, TIGER+, in this study 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 model and test suite created in this study using requirements of a subsystem of TCMS developed at Alstom also contains particularities related to Alstom-specific testing libraries, development, and testing environments that may not apply to other domains. However, we have provided a detailed description of the proposed approach, tool, and experimental methodology for the other researchers working in a similar domain to apply it to other test generation tools and to replicate the study. 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 for the generalization of the tools and knowledge in different contexts.

We used the mutation analysis in this study to evaluate the fault detection effectiveness as it is often recommended in the literature. Furthermore, to select the mutant operator, we have conducted a thorough investigation on the development of the FBD program from the literature, applicability of the selected mutant operators in industrial settings, and dependencies of the tools and recommendation of safety standards i.e., EN 50128, EN 50657 used at Alstom for the development of the selected system.

VII. 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. It leverages different machine learning algorithms to extract, preprocess, 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 having minimal effect on the fault detection rate of the test suite. We also performed a comparative mutation analysis of the test suites reduced by the TIGER+, the MBT-generated test suite, and the test suite created manually in terms of fault detection effectiveness and test execution efficiency in an industrial setting. Our results show a significant reduction of the MBT-generated test suite i.e., approx. 85% and 92% if optimized at the highest and lowest level, respectively. 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, 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. Moreover, a thorough code-level analysis to measure the coverage by test suites at the structural level is also warranted.

ACKNOWLEDGEMENT

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.

REFERENCES

[1] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[3] H. Zhong, L. Zhang, and H. Mei, "An experimental study of four typical test suite reduction techniques," *Information and Software Technology*, vol. 50, no. 6, pp. 534–546, 2008.

[4] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

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

[6] B. Nikolik, "Test diversity," *Information and Software Technology*, vol. 48, no. 11, pp. 1083–1094, 2006.

[7] X. Cai and M. R. Lyu, "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*, 2005, pp. 1–7.

[8] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 560–564.

[9] J. A. Nuh, T. W. Koh, S. Baharom, M. H. Osman, L. Babangida, S. Letchmunan, and S. N. Kew, "Diversity-based test case prioritization technique to improve faults detection rate," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 6, 2023.

[10] D. Mondal, H. Hemmati, and S. Durocher, "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)*, 2015, pp. 1–10.

[11] M. N. Zafar, W. Afzal, E. P. Enoiu, A. Stratis, and O. Sellin, "A model-based test script generation framework for embedded software," in *The 17th Workshop on Advances in Model Based Testing*, February 2021. [Online]. Available: http://www.es.mdu.se/publications/6172-

[12] S. Fletcher, M. Z. Islam *et al.*, "Comparing sets of patterns with the jaccard index," *Australasian Journal of Information Systems*, vol. 22, 2018.

[13] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2007, pp. 291–305.

[14] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, pp. 1–42, 2013.

[15] A. E. V. B. Coutinho, E. G. Cartaxo, and P. D. d. L. Machado, "Analysis of distance functions for similarity-based test suite reduction in the context of model-based testing," *Software Quality Journal*, vol. 24, pp. 407–445, 2016.

[16] N. Kushik, N. Yevtushenko, and J. López, "Testing against non-deterministic fsms: A probabilistic approach for test suite minimization," in *IFIP International Conference on Testing Software and Systems*. Springer, 2021, pp. 55–61.

[17] N. Kushik, N. Yevtushenko, and J. L'opez, "Probabilistic approach for minimizing checking sequences for non-deterministic fsms," in *IFIP International Conference on Testing Software and Systems*. Springer, 2023, pp. 237–243.

[18] R. Sachtleben and J. Peleska, "Effective grey-box testing with partial fsm models," *Software Testing, Verification and Reliability*, vol. 32, no. 2, p. e1806, 2022.

[19] R. Pan, T. A. Ghaleb, and L. Briand, "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)*. IEEE, 2023, pp. 1700–1711.

[20] A. Kramer and B. Legeard, *Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level*. John Wiley & Sons, 2016.

[21] E. Meijer, B. Beckman, and G. Bierman, "Linq: reconciling object, relations and xml in the. net framework," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 706–706.

[22] B. Qin, Y. Xia, S. Prabhakar, and Y. Tu, "A rule-based classification algorithm for uncertain data," in *2009 IEEE 25th international conference on data engineering*. IEEE, 2009, pp. 1633–1640.

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

[24] H. Jiawei, K. Micheline, and P. Jian, "Data mining concepts and techniques third edition," *The Morgan Kaufmann Series in Data Management Systems*, vol. 5, no. 4, pp. 83–124, 2011.

[25] J. Albahari and B. Albahari, *LINQ Pocket Reference: Learn and Implement LINQ for. NET Applications*. O'Reilly Media, 2008.

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

[27] Y. Crama and P. L. Hammer, *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

[28] T. Stubblebine, *Regular Expression Pocket Reference: Regular Expressions for Perl, Ruby, PHP, Python, C, Java and. NET*. " O'Reilly Media, Inc.", 2007.

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

[30] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on software Engineering*, vol. 29, no. 3, pp. 195–209, 2003.

[31] K. J. Hayhurst, *A practical tutorial on modified condition/decision coverage*. DIANE Publishing, 2001.

[32] M. N. Zafar, W. Afzal, E. P. Enoiu, A. Stratis, A. Arrieta, and G. Sagardui, "Model-based testing in practice: An industrial case study using graphwalker," in *Innovations in Software Engineering Conference 2021*, February 2021. [Online]. Available: http://www.es.mdu.se/publications/6101-

[33] M. N. Zafar, W. Afzal, and E. P. Enoiu, "An empirical evaluation of system-level test effectiveness for safety-critical software," in *18th International Conference on Evaluation of Novel Approaches to Software Engineering*, April 2023. [Online]. Available: http://www.es.mdu.se/publications/6638-

[34] A. T. Acree, R. A. DeMillo, T. Budd, R. J. Lipton, and F. G. Sayward, *Mutation analysis*. School of Information and Computer Science, Georgia Institute of Technology ..., 1979.

[35] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 919–930.

[36] Y. Oh, J. Yoo, S. Cha, and H. S. Son, "Software safety analysis of function block diagrams using fault trees," *Reliability Engineering & System Safety*, vol. 88, no. 3, pp. 215–228, 2005.

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

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