Computers, Materials & Continua
DOI:10.32604/cmc.2022.023803
Article

Tech Science Press

An EFSM-Based Test Data Generation Approach in Model-Based Testing

Muhammad Luqman Mohd-Shafie[1,*], Wan Mohd Nasir Wan Kadir[1], Muhammad Khatibsyarbini[1], Mohd Adham Isa[1], Israr Ghani[1] and Husni Ruslai[2]

[1]Department of Software Engineering, School of Computing, Faculty of Engineering, Universiti Teknologi Malaysia, 81310, Johor Bahru, Johor, Malaysia
[2]GATES IT Solution Sdn Bhd, WISMA GATES, Jalan Perdana 4, Taman Sri Pulai Perdana 2, 81300, Skudai, Johor, Malaysia
*Corresponding Author: Muhammad Luqman Mohd-Shafie. Email: mluqman24@graduate.utm.my
Received: 22 September 2021; Accepted: 01 November 2021

Abstract: Testing is an integral part of software development. Current fast-paced system developments have rendered traditional testing techniques obsolete. Therefore, automated testing techniques are needed to adapt to such system developments speed. Model-based testing (MBT) is a technique that uses system models to generate and execute test cases automatically. It was identified that the test data generation (TDG) in many existing model-based test case generation (MB-TCG) approaches were still manual. An automatic and effective TDG can further reduce testing cost while detecting more faults. This study proposes an automated TDG approach in MB-TCG using the extended finite state machine model (EFSM). The proposed approach integrates MBT with combinatorial testing. The information available in an EFSM model and the boundary value analysis strategy are used to automate the domain input classifications which were done manually by the existing approach. The results showed that the proposed approach was able to detect 6.62 percent more faults than the conventional MB-TCG but at the same time generated 43 more tests. The proposed approach effectively detects faults, but a further treatment to the generated tests such as test case prioritization should be done to increase the effectiveness and efficiency of testing.

Keywords: Model-based testing; test case generation; test data generation; combinatorial testing; extended finite state machine

1 Introduction

In short, the International Software Testing Qualification Board (ISTQB) defined testing as the planning, preparation, and evaluation of a component or system to ensure that they adhere to specified requirements, prove that they are fit for purpose, and identify faults. Testing determines whether the system under test (SUT) fulfils the specified requirements agreed during the requirements elicitation. At the same time, testing also aims to identify faults in the SUT caused by errors made in the code

during software development or maintenance. One of the essential artefacts in testing is the test cases. Test case generation (TCG) is where test cases are created from the available test basis.

The necessity of effective and efficient testing is becoming more significant in the current fast-moving system developments. In general, being effective means adequately achieving an intended purpose, while being efficient means accomplishing the intended purpose in the best way possible while saving time and effort. Therefore, based on the definition of testing earlier, effective testing can be measured by how many faults can be detected with a given testing strategy [1], while efficient testing can be measured by the time or effort needed to detect faults [2]. According to a statistic by Memon et al. [3], every day at Google, 150 million tests execute on more than 13 thousand projects that require 800 thousand builds. In large scale and fast-paced development projects, manually creating tests is time consuming and susceptible to human errors. Therefore, the transition toward automatic TCG to be more effective and efficient in testing is imperative, even more so in the current industry 4.0 revolution.

Some of the well-known automatic TCG techniques are symbolic execution, model-based testing (MBT), combinatorial testing (CT), adaptive random testing, and search-based testing [4]. Model-based TCG (MB-TCG) is based on MBT [5], which utilizes an automation process and SUT models to generate test cases. By exploiting the system models, test cases representing the expected behaviours of the SUT can be developed promptly with little to zero human effort. This study uses the extended finite state machine (EFSM) model because it has formal definitions that can ease the automatic processes in MBT that require consistency and accuracy.

One of the advantages of using MBT is that the testing activity can be commenced in the early phase of the software development. In addition, with MBT, it is possible to start the actual testing of the system under test (SUT) earlier. Another advantage of MBT is that contradiction and obscurity in the requirements specification and the design documents can be identified and fixed during the design and specification phase. All in all, it has been proved that MBT can decrease the associated development cost while increasing the software quality [6].

Although MB-TCG can increase the effectiveness and efficiency of testing, existing approaches in the literature indicated a significant limitation, particularly regarding test data generation (TDG). TDG is one of the main aspects of MBT [7]. However, it is still one of the main challenges in automating MBT [8]. Manual TDG is still common in existing MB-TCG studies even though it is costly, complicated and laborious. TDG possesses a high potential in the testing cost reduction since it can decrease human effort. The selection of test data is also crucial because it can affect the number of faults detected during testing. A study by Ahmad et al. [9] that reviewed MBT studies using activity diagrams discovered that more than half of the selected studies did not explicitly specify their TDG methods. This finding conveyed that the TDG in MBT was still undervalued, despite its importance. Therefore, more research is needed to improve the TDG in MB-TCG. More discussion regarding the TDG in existing MB-TCG approaches is presented later in Section 3.

CT is a technique that tests the SUT using a covering array test suite that contains all the possible t-way combinations of parameter values [10]. It is based on the idea that instead of exhaustive testing all possible parameter combinations of the SUT, only a subset of them is used, which satisfies some predefined combination strategies. Although only a subset is used, this technique's effectiveness in detecting faults is on par with exhaustive testing that is confirmed to detect all faults. This similarity in effectiveness is achievable because faults seem to result from interactions of only a few variables, so tests that cover all such few-variable interactions can be very effective [11].

CT is a suitable technique to address the TDG generation issue in MB-TCG because its implementation can be automated. So, it can be added on top of the automation in the MBT. In addition,

the CT technique was chosen because both MBT and CT complement each other very well [12]. CT can address the TDG issue in MBT because it deals with the interaction between input parameters. Meanwhile, MBT can address the issue in CT where it has no model and paths that can guide the generation of structured and effective tests to ensure proper coverage of the SUT. MBT can address this limitation by automatically generating paths that act as test cases, which can be used by the CT technique to guide the TDG. Last but not least, the CT technique was chosen because it can provide fewer test data for the proposed approach without reducing the fault detection capability too significantly.

Motivated by the limitation in the existing MB-TCG approaches and the importance of TDG in MBT, this study proposes an automated TDG approach in MB-TCG using the EFSM model and the CT technique. The proposed approach in this study adopts and modifies the approach proposed by Nguyen et al. [12]. Their proposed approach also combined MBT and CT. However, the domain input classifications are manually specified in their study. This study automates the domain input classifications step by taking advantage of the information available in an EFSM model. This improvement can further reduce the human intervention required in automating the TDG in MB-TCG while detecting more faults. The contributions of this study are twofold. First, an automated TDG approach in MB-TCG using CT technique and EFSM model is proposed. Second, an experiment that was done to assess the effectiveness and efficiency of the proposed approach is presented.

The remainder of this paper is organized as follows. Section 2 gives a brief background concerning MBT, EFSM, and CT. Section 3 discusses studies with the TDG issue and related works similar to this proposed approach. Next, Section 4 presents the explanation regarding the proposed approach. Section 5 shows the experiment done to assess the proposed approach effectiveness and efficiency. Lastly, Section 6 discusses the conclusion and future works of this study.

2 Background

2.1 Model-Based Testing

MBT is a branch of testing under black-box testing or functional testing [5]. It relies on the SUT models that visualize the expected behaviours of the SUT in performing testing. Due to its black-box nature, the SUT source code is not required, and the MBT process can be initiated as early as the design phase of software development. In brief, MBT comprises the steps for the automatic generation of abstract tests from the SUT models, the generation of concrete tests from the abstract tests, and the manual or automatic execution of concrete tests. Approaches in MB-TCG generate tests using similar procedures as the MBT steps. Therefore, a good comprehension of how MBT is done is crucial. A brief description of the steps is presented next. A detailed description of MBT can be found in the study by Utting et al. [5].

The first step in MBT is to build one or more test models. EFSM is one of the common models used in MBT [13,14]. Test models are usually created from informal requirements or specification documents. In some cases, design models are used as test models. Next, one or more test selection criteria are decided to drive the automatic abstract test generation. For example, the test selection criteria can be related to the test model's structure, such as state coverage or transition coverage. In the third step, the criteria are transformed into test case specifications (TCSs) that describe the notion of test selection criteria. In the fourth step, abstract test cases are generated to satisfy all of the TCSs. In this step, automation tools are usually used to generate abstract tests given the model and the TCSs [15].

Lastly, the abstract tests are concretized and executed against the SUT in the fifth step. Generating the test data for each abstract test is one of the processes in concretizing the tests. This part is the focus of this study. The test execution process can be done manually or using a test execution tool to execute the tests and record their verdicts automatically. As explained earlier, the models that describe the expected behaviours of the SUT were created using the informal requirements or specification documents, which are usually assumed correct. Hence, the models can be used as the test oracle to compare with the actual behaviour during testing to decide the test verdicts.

2.2 Extended Finite State Machine

An EFSM comprises states and transitions between states [16]. Transitions in an EFSM consist of events, conditions and a sequence of actions. A particular transition is executed when the transition's specified event is triggered and all the transition's conditions are assessed to true. Then, actions associated with that transition are executed. A state could be interpreted as the current values of a set of variables that the SUT has [17]. These variables that dictate which state the SUT is currently in are usually called context variables or internal variables [13,14], in contrast to user variables, which hold the user inputs. When a transition is executed, the context variables of the SUT could change, as instructed by the actions, which then lead to a state change.

For this study, an EFSM is defined as a 7-tuple $M = (\Sigma, Q, Start, Exit, V, P, R)$ where $\Sigma = \{T_1, T_2, \ldots, T_O\}$ denotes the set of all transitions, $Q = \{Start, S_1, S_2, \ldots, Exit\}$ denotes the set of all states, $Start \in Q$ denotes the start state, $Exit \in Q$ denotes the stop state, $V = \{v_1, v_2, \ldots, v_O\}$ denotes a finite set of input and context variables, $P = \{a_1, a_2, \ldots, a_O\}$ denotes the set of all actions, $R = \{c_1, c_2, \ldots, c_O\}$ denotes the set of all enabling conditions. T is a transition denoted by the tuple: $T_i = (G, C, A, S_s, S_e)$ where $G = \{v_1, v_2, \ldots, v_O\}$ denotes a set of input variables, $C = \{c_1, c_2, \ldots, c_O\}$ denotes a set of enabling conditions, $A = \langle a_1, a_2, \ldots, a_O \rangle$ is a series of actions executed, $S_s \in Q$ denotes transition's starting state, $S_e \in Q$ denotes transition's exiting state. S is a state denoted by the tuple: $S_i = (W, T_s, T_e)$ where $W = \{v_1, v_2, \ldots, v_O\}$ denotes a set of context variables with values range that constitute S_i, $T_s \in \Sigma$ denotes the state's incoming transition, $T_e \in \Sigma$ denotes the state's outgoing transition.

2.3 Combinatorial Testing

Combinatorial testing (CT) is a technique that tests a SUT using a covering array test suite that contains all the possible t-way combinations of parameter values [10]. In a covering array, the columns represent the parameters while the rows represent the tests. Assumes an example application is to be tested whether it works correctly on a computer that uses Windows or Linux OSs, Intel or AMD processors, and the IPv4 or IPv6 protocols. These parameters (OS, processor, and protocol) and their values (the possible choices for each parameter) require $2*2*2 = 8$ tests to check each component interacting with every other component at least once if exhaustive testing is used. With the pairwise testing ($t = 2$) technique, only four tests are required to test all possible pairs (two components) of combinations. An empirically derived rule, called the *interaction rule*, claims that most failures came from single factor faults or the interaction of two factors, with fewer failures induced by interactions between three or more factors. This rule is why CT can still be effective even though fewer tests are used than exhaustive testing.

A covering array $CA(N, n, s, t)$ is a form of $N \times n$ matrix where N is the number of rows (array size), n is the number of columns (parameters), s is the level (number of possible values for parameters), and t is the interaction strength. Algorithms to generate a covering array are primarily categorized into

computational and algebraic methods. In brief, computational methods work by directly listing and covering every t-way combination, while algebraic methods operate according to predefined rules in contrast to computational methods. Automatic efficient test generator (AETG) and in-parameter-order-general (IPOG) are the two most used algorithms in generating covering arrays [11], and both are computational methods. However, AETG builds a complete test at a time while IPOG covers one parameter at a time. More details regarding CT can be observed in the study from Kuhn et al. [11].

3 Related Work

Kamath et al. [18] proposed an MB-TCG approach using the activity diagram model. The test data in their approach were assumed to be provided manually by a tester because only abstract test data were presented, and no further detail was given. Andrews et al. [19] proposed an MB-TCG approach for testing Urban Search and Rescue (USAR) robots using the class diagram and the Petri net model. All Combination Coverage (ACC) and Each Choice Coverage (ECC) approaches were used for selecting test data. Both ACC and ECC implementations for generating test data were considered manual based on how test data were created and selected in their approach. Majeed et al. [20] proposed an MB-TCG approach for event-driven software using the FSM model. The TDG in their approach was considered semi-automated because in the first phase, manual test generation and execution must be done first before automated test generation and execution can be achieved in the second phase. This implementation means that the test data were manually generated during the first phase.

Singi et al. [21] proposed an MB-TCG approach for testing from visual requirement specifications, focusing specifically on prototyping. The TDG in this approach was considered a manual process because the test cases only provided templates for the tester to enter appropriate test data. Gutiérrez et al. [22] proposed an MB-TCG approach to automatically generate functional test cases from functional requirements. For the TDG, the equivalence partitioning (EP) method was used after formalizing the information obtained by applying the Category-Partition method to the functional requirements. They stated that this process was manual. Sarmiento et al. [23] proposed an MB-TCG approach to generate test cases from natural language requirements specifications using the activity diagram model. The generation of test data in this proposed approach was identified to be manually done. They also stated that this process required human intervention and did not address it further.

These studies mentioned earlier showed that many existing MB-TCG approaches still used manual TDG. Several studies also still required test data manually provided by the tester. This manual method is inefficient and can result in human error, especially in large-scale testing. Most of these mentioned studies were published in the range of the last decade. This circumstance inferred that manual TDG was still practised in recent approaches. To make the circumstance worse, TDG still has not been given much attention when proposing MB-TCG approaches, as discovered by the latest review study conducted by Ahmad et al. [9]. These were the reasons why this study proposes an automated TDG approach in the MB-TCG.

The closest work to this current study is probably from Nguyen et al. [12] because their proposed approach was adapted and modified in this study. They combined MBT and CT, where test sequences were derived using the FSM model of the SUT and complemented with selective test input combinations. One limitation of their approach was that the domain input classifications were manually specified to transform paths to classification trees. This study enhances the existing approach by automating the domain input classifications to increase further the automation level of the whole test data generation process.

Another similar work to this current study is from Kansomkeat et al. [24]. They proposed an MB-TCG approach using the activity diagram. The input domains were classified based on the guard conditions in the decision points of the activity diagram. Nonetheless, the test suite generated using their approach consisted of all possible combinations of input classes, similar to exhaustive testing. Also, they did not use any CT tool for generating the combinations. In this current study, the conditions in the EFSM model transitions are used to classify the input domains. In addition, the generated test suite consists of the interaction between several input classes only. This implementation decreases the total number of tests generated. Furthermore, this current study uses the existing CT tool to increase the efficiency of generating combinations.

4 Proposed Approach

Fig. 1 shows the framework that visualizes the essential steps in implementing the proposed TDG approach. The proposed approach consists of five steps. They are (1) the generation of test paths, (2) the transformation of classification trees, (3) the generation of test combinations, (4) the removal of duplicates, (5) the execution of tests and incremental refinement of constraints. Each step is explained further in the following subsections.

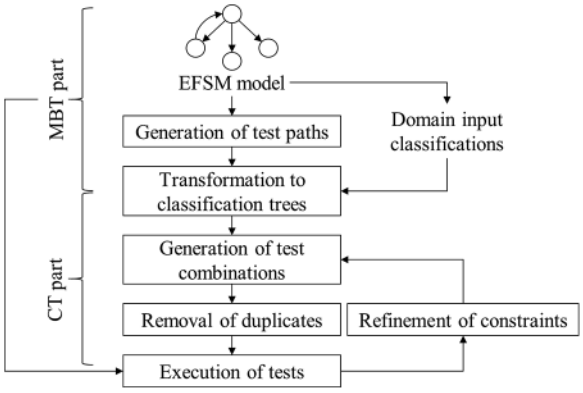

Figure 1: Proposed TDG approach framework

4.1 Generation of Test Paths

A test path is a combination of the SUT states and transitions in the EFSM form. It can be generated from the EFSM using various strategies like state coverage and transition coverage. A complete test path has the start and end states, $\langle Start, T_1, \ldots, Exit \rangle$. In MBT, this part is represented as step two until step four. A complete test path is also an abstract test case that cannot be used yet for testing because it does not have all the necessary information to execute with the SUT.

4.2 Transformation to Classification Trees

The classification tree method is used to interpret and analyze input partitions and test input combinations that will be done later. The complete test paths generated previously are transformed into classification trees where each tree represents a path. After the transformation, each tree will contain (1) a root node that acts as an identifier for the transformed test path, (2) child nodes of the root node that represent the sequence of transitions of the transformed test path, from left to right, (3) child nodes for each of the upper child nodes that represent the parameters for each transition of

the transformed test path, and lastly (4) leaf nodes for each of the upper child node that represent the input classification for each parameter.

Only transitions are transformed into child nodes in the second item contained in a classification tree. The states are excluded because they deal with the verification of certain SUT states, whereas the transitions deal with the users' inputs to the SUT, which are the required information in this approach. Also, only transitions that take user inputs are taken into consideration.

To explain this step further, consider an example of a test path, $t_1 = \langle Start, T_1, S_1, T_2, S_2, T_3, Exit \rangle$. Tab. 1 shows the input parameters and classifications of the transitions traversed by t_1. After the transformation, the classification tree, as shown in the upper part of Fig. 2, is produced.

Table 1: Parameters and input classes for example test path 1

Transition	Parameter(s)	Input classes
T_1	p_1	a, b
T_2	$p_{2.1}$	c, d
	$p_{2.2}$	e, f
T_3	p_3	g, h

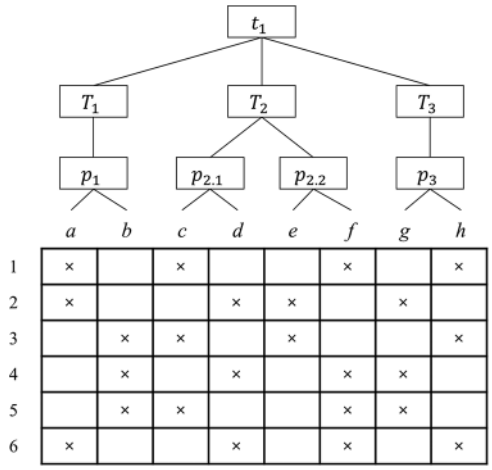

Figure 2: Resulting classification tree and the generated test combinations with pairwise coverage for example test path 1

As mentioned earlier in the introduction and the related work, Nguyen et al. [12] manually specified the domain input classifications. This step is where this research approach differs from their approach. Since this approach uses the EFSM model, the enabling conditions available in each transition will be used and transformed to input classifications. Also, the boundary value analysis (BVA) strategy is used.

The general rule for the input classification in this approach is that, for an enabling condition comparing an input variable with a constant value or a context variable value, the input should be classified into the highest or lowest possible values. Tab. 2 shows the example classifications of input for each type of comparison operator in an enabling condition where x is the input variable.

Table 2: Example classification of input for each type of operator in an enabling condition

Operator	Enabling condition	Input classifications
$>$	$x > y$	$a : x = y + 1$ (Lowest value) $b : x > y + 1$ (Highest value)
$<$	$x < y$	$a : x = y - 1$ (Highest value) $b : x < y - 1$ (Lowest value)
$\geq$	$x \geq y$	$a : x = y$ (Lowest value) $b : x > y$ (Highest value)
$\leq$	$x \leq y$	$a : x = y$ (Highest value) $b : x < y$ (Lowest value)
$=$	$x = y$	$a : x = y$ (Exact value)
$\neq$	$x \neq y$	$a : x = y + 1$ (Above exact value) $b : x = y - 1$ (Under exact value)

For example, an enabling condition of a transition is $c_i = (x \geq 10)$. The inputs will then be classified into $a{:}x = 10$ and $b{:}x > 10$. Assuming that x is an integer type variable, the highest possible value is 2,147,483,647. So, that value will be used for the classification b. Input class for $x < 10$ will not be covered because other transitions will usually cover the range of input values not covered by a particular transition. It is also to prevent incorrect transitions from being executed when a test path has been specified, which will cause the test to be misinterpreted as failed. The same rule applies for enabling conditions having logical operators that combine two or more conditions. For example, if an enabling condition of a transition is $c_i = (x \geq 10\ AND\ x \leq 20)$, then the inputs will be classified into $a{:}x = 10$ and $b{:}x = 20$. Note that only enabling conditions related to user input variables are transformed because this part is all about the input classification. Other types of enabling conditions unrelated to user inputs, such as checking a context variable, are not considered.

4.3 Generation of Test Combinations

After the classification tree for each test path is completed, test combinations are generated. In generating the test combinations, t-way combinations with any desirable interaction strength, such as 2-way (pairwise), 3-way and so on, can be utilized, depending upon the required t-way coverage.

The lower part of Fig. 2 shows the generated combinations for the example earlier. It consists of the classification tree and the generated covering array, which is for pairwise combination. The covering array in Fig. 2 was generated using the CTWedge tool [25], which uses the ACTS tool [26], based on the IPOG algorithm, as the test generator.

4.4 Removal of Duplicates

Some generated test paths traversed the same transitions or states. Because of this circumstance, some generated test cases will possess identical input parameter combinations. Suppose these test cases, containing only identical combinations with other test cases while not having any unique combination, are not removed. In that case, test execution will become inefficient because these redundant test cases are pretty much useless.

The method from Nguyen et al. [12] is followed to identify and discarded these redundant tests. The objective is to identify and discard unnecessary combinations while maintaining the t-way combination coverage within all the generated paths at the global level.

To exemplify, consider two generated paths from a model, $t_1 = \langle Start, T_1, S_1, T_2, S_2, T_3, S_3, T_4, S_4, T_5, Exit\rangle$ and $t_2 = \langle Start, T_1, S_1, T_2, S_2, T_3, S_3, T_4, S_4, T_6, Exit\rangle$. Tab. 3 shows the input parameters and classifications of the transitions traversed by t_1 and t_2.

Table 3: Parameters and input classes for test path 1 and 2

Transition	Parameter	Input classes
T_1	p_1	a, b
T_2	p_2	c, d
T_3	p_3	e, f
T_4	p_4	g, h, i
T_5	p_5	j, k
T_6	p_6	l, m

Tab. 4 lists all the generated combinations and the pairwise combinations that each combination uniquely covers (in this example, it is assumed that testing needs to fulfil pairwise coverage).

Table 4: Generated combinations for test path 1 and 2 with their uniquely covered combinations

Path	#	Test combination	Initial pairwise combinations uniquely covered	Pairwise combinations uniquely covered after test combination four removal
t_1	1	$[b, c, e, h, j]$	$[h, j]$	$[h, j]$
	2	$[a, d, f, h, k]$	$[f, k], [h, k]$	$[f, k], [h, k]$
	3	$[b, d, e, i, k]$	$[b, k], [i, k]$	$[b, k], [i, k]$
	4	$[b, c, f, g, j]$	-	-
	5	$[a, c, e, g, k]$	$[c, k], [g, k]$	$[c, k], [g, k]$
	6	$[a, c, f, i, j]$	$[i, j]$	$[i, j]$
	7	$[a, d, e, g, j]$	$[d, j]$	$[d, j]$
t_2	1	$[b, c, e, h, l]$	$[h, l]$	$[h, l]$
	2	$[a, d, f, h, m]$	$[f, m], [h, m]$	$[f, m], [h, m]$
	3	$[b, d, e, i, m]$	$[b, m], [i, m]$	$[b, m], [i, m]$
	4	$[b, c, f, g, l]$	-	$[b, f], [b, g], [f, g]$
	5	$[a, c, e, g, m]$	$[c, m], [g, m]$	$[c, m], [g, m]$
	6	$[a, c, f, i, l]$	$[i, l]$	$[i, l]$
	7	$[a, d, e, g, l]$	$[d, l]$	$[d, l]$

After the uniquely covered combinations for each test combination have been identified, one test combination that does not cover any unique combination is removed. Then, the uniquely covered combinations for all test combinations left are recomputed because they depend on the remaining test combinations. From Tab. 4, in the fourth column, it can be observed that both test combinations four from the path t_1 and t_2 do not contain any unique pairwise combination. Therefore, any one of them will be removed at random. Assume that test combination four from the path t_1 is removed, the fifth column of Tab. 4 lists all the remaining test combinations and the new pairwise combinations that each combination uniquely covers.

It can be observed that after test combination four from the path t_1 is removed, all the remaining test combinations will cover at least one unique pairwise combination. Therefore, no more test combination can be removed without compromising the t-way combination coverage.

4.5 Execution of Tests and Incremental Refinement of Constraints

In this step, the remaining test combinations can be prepared to be used for testing the SUT. As mentioned earlier in Section 4.1, a complete test path is also an abstract test case, which is not yet usable for testing. These generated test combinations, which subsume the complete test paths, are also considered abstract tests. The test data representing the input data classification must be generated to make them concrete tests, and transitions must be connected with the SUT. This step is pretty much similar to the fifth step in the MBT process explained in Section 2.1. The difference is that test data are usually created manually by the tester in a typical MBT process. In contrast, the test data that adhere to the input data classification and the t-way combination coverage are automatically generated in this approach.

Regarding the execution, there are possibilities that some test paths are infeasible. Some of the reasons are because of conditions conflicts in the path [27], conflicts between context variables and enabling function [28] or because in one state reached, the subsequent available action/event is not accepted by the SUT [12]. These circumstances happen possibly because of (1) faults in the specifications/models, (2) faults in the SUT or (3) there are dependencies between inputs that are missing. For the first case, it is sometimes possible that the specifications/models themselves are faulty. However, it is usually assumed that the specifications/models have been validated and considered accurate before test execution. The second case is an occurrence where a fault in the SUT has been detected.

However, it would mean that one or more dependencies between input classifications have been missed during the generation of test combinations for the third case. To prevent this from happening, path constraints that keep track of the dependency relationship among input classifications are specified to ensure that invalid test combinations are not generated. This step requires human intervention and domain knowledge, so it is done manually. After all the missing constraints have been specified, test combinations will have to be generated again and the subsequent steps repeated.

5 Empirical Study

This section discusses the experiment done to assess the effectiveness and efficiency of the proposed TDG approach in testing. The research questions (RQs) for this experiment are:

RQ1. What are the fault detection capability and code coverage of the proposed TDG approach compared to the conventional MB-TCG?

RQ2. What is the resulting test suite size of the proposed TDG approach compared to the conventional MB-TCG?

RQ1 was designed for the effectiveness assessment by determining whether the test data generated using the proposed TDG approach can detect more faults and achieve higher code coverage than when using the test data from the conventional MB-TCG. RQ2 was designed for the efficiency assessment by comparing the resulting test suite size using the proposed TDG approach and the conventional MB-TCG. To an extent, executing fewer test cases means less testing time and effort, thus higher efficiency. However, it is not always accurate [29]. The generation time and execution time of each test also play significant parts in determining testing time. These aspects will be considered in the future work of this study.

Fig. 3 illustrates the framework of the experiment in general. The parts for the conventional MB-TCG and the proposed approach have been discussed in more details in Section 2.1 and Section 4, respectively. The remaining parts are explained accordingly later.

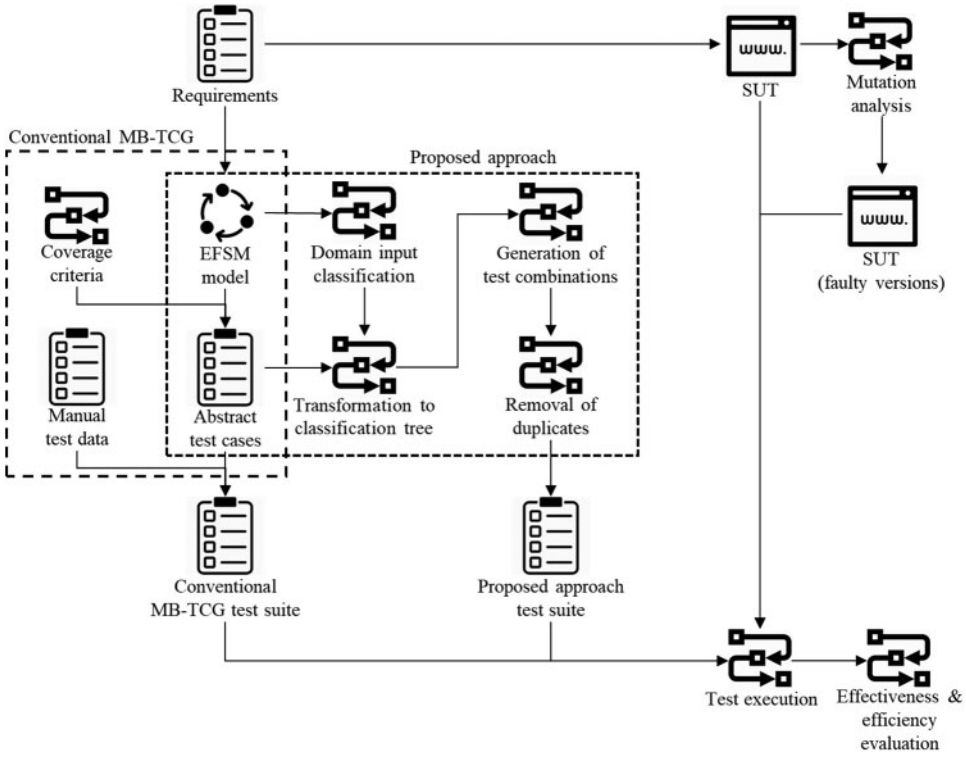

Figure 3: Experiment framework

The TestOptimal tool was used in the experiment for performing the MBT part [30]. It was responsible for implementing the conventional MB-TCG and the proposed approach parts in Fig. 3. TestOptimal was chosen because the end state can be included in the model. This feature was essential for resetting the path generation when the end state is reached and generating several test paths. Another reason for choosing TestOptimal was because it supported data-driven testing (DDT). This feature made it easier for test data combinations to be used alongside the generated test cases.

Another tool that was utilized in this experiment was the muJava tool [31]. It was responsible for implementing the mutation analysis part in Fig. 3. It was used to seed faults into the SUT to measure

the quality of a generated test suite using the mutation score metric. This tool was chosen because the mutant generation was done at the source code level. This approach was preferable because mutants can closely imitate the types of errors programmers might make. Besides, the injected mutants can be clearly described and understood by testers this way. Lastly, the EclEmma tool was used to obtain the code coverage information. It was chosen because it generates code coverage information for programs in Java, the language used to implement the SUT. In addition, it is an open-source tool and is still up-to-date in the year 2021. Usage of open-source and up-to-date tools are recommended because they ease future research to utilize or extend them [32].

5.1 Case Study

System models for real-world commercial software with their respective software systems are not available free [33]. Consequently, this research uses the lift system case study from Kalaji et al. [34]. The case study's specifications and details are represented as the "Requirements" in Fig. 3.

In the context of the EFSM model, the lift system has six states $Q = \{Start,\ Floor_0,\ Floor_1,\ Floor_2,\ Stop,\ End\}$, four context variables, five input variables $V = \{doorOpened,\ inputUpdated,\ floor,\ weightload,\ liftPosition,\ preferredFloor,\ temploadtemperature,\ smoke\}$ and 36 transitions. The SUT for the lift system case study was implemented as a Java program using Eclipse IDE based on its specifications. It is represented as the "SUT" in Fig. 3.

5.2 Benchmark Approach

For this experiment, the test data sequences generated from the proposed approach were compared with those from the conventional MB-TCG. The proposed approach by Nguyen et al. [12] was not included in this experiment because it required manually specifying the domain input classifications. As this step is crucial in implementing the proposed approach, performing it manually would introduce bias that will affect the obtained results. In addition, the study did not provide a systematic way of specifying the domain input classifications. This limitation would introduce variabilities whenever the approach is replicated, which will then affect the obtained result. These issues were the reasons why their approach was not included in the comparison.

The test data sequences from the conventional MB-TCG were arbitrarily or randomly created as long as they adhered to the test case flow. This process can be observed in Fig. 3, where the manual test data are combined with the abstract test cases to produce a concrete test suite for the conventional MB-TCG. In the proposed approach, the difference was that the combinatorial testing approach and the information available in the EFSM model were utilized to generate better test data sequences for fault detection. This process can be observed in Fig. 3, where the EFSM model and the abstract test cases are used to produce a concrete test suite for the proposed approach.

5.3 Evaluation Metrics

Mutation analysis was used in the experiment to assess an MB-TCG approach performance in detecting faults. A fault is introduced by making a syntactic change to the original SUT. One change injected into a copy of the SUT results in one faulty version called a mutant. A mutant is "killed" or detected by a test suite if its result running the mutant is different from the result with the original SUT. This process is also illustrated in Fig. 3 in the upper-right part.

The mutation score is the outcome of mutation analysis. The calculation for the mutation score *(MS)*, taken from De Souza et al. [35], is as follows:

$$MS(P,T) = \frac{KM(P,T)}{TM(P) - EM(P)} \tag{1}$$

where P is the SUT being mutated, T is the test suite being used, $KM(P,T)$ is the number of killed mutants, $TM(P)$ is the total number of mutants and $EM(P)$ is the number of equivalent mutants. Equivalent mutants are mutants that cannot be killed. They are syntactically different but functionally equivalent to the original SUT. Because of this characteristic, automatically distinguishing all of them from the non-equivalent mutants is nearly impossible [36] and often done manually.

Mutation analysis was used as a metric for the effectiveness assessment because it provides a quantitative measure of how well a test suite detects faults. A higher mutation score reflects more faults are detected, which shows that the test suite is good. Furthermore, mutation analysis was used because it resembles real faults typically made by programmers. Empirical results showed that 85 percent of errors caused by mutants were also caused by real faults [36]. In the scarcity of publicly available real faults from the industry, mutation analysis is an alternative to alleviate the threat to external validity regarding fault seeding during the experimentation. In addition, mutation analysis was used because the implementation can be automated using mutation tools.

Code coverage was used as another metric for the effectiveness assessment because higher code coverage means more confidence in the reliability of the software so as not to fail [37]. Code coverage was also used because it is an effective metric in predicting a test suite quality in terms of fault detection [38]. Test suite size was used as a metric for the efficiency assessment because it generally affects the time and effort required to complete test execution. It was also used because it is one of the most common and simplest metrics in measuring the performance of an MB-TCG approach [32].

5.4 Faults Seeding

Mutants were seeded into the lift system Java program using the muJava tool. All method-level operators and class-level operators supported by muJava were used. In total, 767 and four mutants were generated for method-level and class-level, respectively. This study used the mutant sampling method to maximize efficiency without deteriorating mutation analysis effectiveness significantly [36]. A quarter of the total mutants, approximately 25 percent, were randomly picked. Empirical results suggested that a random selection of 10 percent of mutants is only 16 percent less effective than utilizing the complete set of generated mutants in terms of mutation score [36]. This method is an acceptable trade-off between the mutation score effectiveness minimization and the amount of work reduced. After a subset of mutants was selected, equivalent mutants were identified and discarded. One hundred seventy-nine mutants were selected, where 43 were equivalent mutants, and 136 were non-equivalent mutants.

5.5 Generated Test Suites

Abstract tests were generated by simply executing the constructed model using the optimal sequencer provided by the TestOptimal tool. This sequencer generates the least number of necessary abstract tests that can cover every transition in the model. In total, ten abstract tests were generated. For the conventional MB-TCG, random test data sequences were used to concretize the ten abstract test cases. The concrete test cases were then executed against every selected mutant discussed earlier to determine the test verdicts.

The ten abstract test cases were further manipulated for the proposed MB-TCG approach before test data sequences were generated, as explained in Section 4. 74 test data sequences were initially generated for all ten abstract tests after the proposed approach was implemented. After the duplicate removal was done, 53 test data sequences remained. The test data sequences generated from the proposed approach were then used to concretize the ten abstract tests. Each abstract test will be concretized by several test data sequences generated from the proposed approach, unlike the conventional MB-TCG, where each abstract test case only has one test data sequence. The concrete test cases were then executed against every selected mutant discussed earlier to determine the tests verdicts. The process explained in this subsection is also illustrated in Fig. 3, where the test suites from the conventional MB-TCG and the proposed approach, the original SUT, and the faulty versions of SUT are used during the test execution.

5.6 Result Analysis and Discussion

This subsection is represented as the "Effectiveness and efficiency evaluation" in Fig. 3. Tab. 5 tabulates the results summary of the experiment. The result showed that the number of mutants detected by the proposed approach test suite was 106, nine more than the conventional MB-TCG. This number gave the proposed approach a mutation score of 77.94 percent, 6.62 percent higher than the conventional MB-TCG. The code coverage achieved by both test suites was the same. However, in terms of the number of test data sequences, conventional MB-TCG outperformed the proposed approach with ten sequences over 53 sequences.

Table 5: Test suites effectiveness results

Test suite	Test data sequences	Code coverage (%)	Mutants killed	Mutation score (%)
Conventional MB-TCG	10	80.5	97	71.32
Proposed approach	53	80.5	106	77.94

To further analyze the effectiveness of the conventional MB-TCG and the proposed approach, their types of mutants detected were investigated. Fig. 4 illustrates the types of mutants detected. It can be observed that the proposed approach outperforms the conventional MB-TCG by detecting more arithmetic operator insertion (AOI) and relational operator replacement (ROR) mutants.

The test suite generated from the proposed approach detected more mutants compared to the conventional MB-TCG. This finding was obtained because the proposed approach improved the test data used for each test case. By integrating the BVA strategy and transitions' enabling conditions into the data classification step of the proposed approach, faults that happen to be near or at boundary conditions can be detected. Conventional MB-TCG was unable to detect these faults because it mainly used random test data between the acceptable ranges. In terms of code coverage, both test suites achieved identical scores. First, this finding was obtained because the test data from the proposed approach and the conventional MB-TCG used the same test cases. Second, the test data from the proposed approach tested the boundary of a condition without exceeding the condition's range of possible data, while the conventional MB-TCG test data were taken randomly within the

condition's range. Therefore, the execution path was the same for both the proposed approach and the conventional MB-TCG, which resulted in identical code coverage.

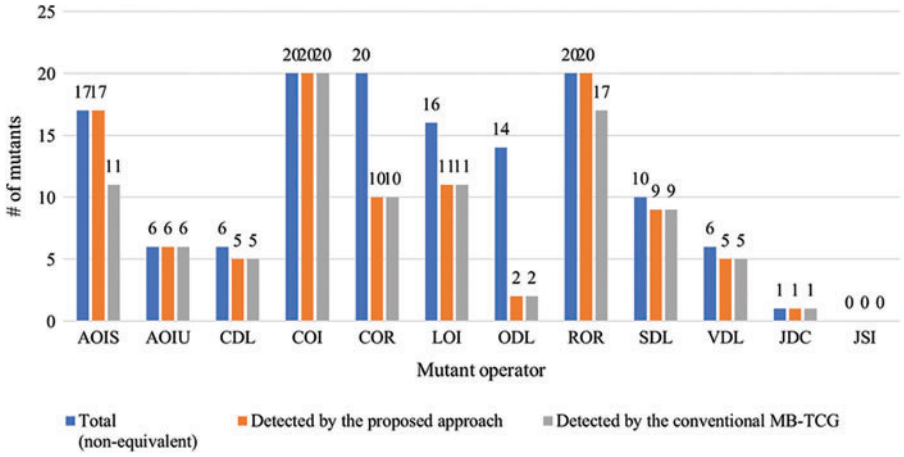

Figure 4: Types of mutants detected by the conventional MB-TCG and the proposed approach

By inspecting the mutants detected by both approaches, the proposed approach can detect AOI and ROR mutants better than the conventional MB-TCG. For AOI, this is logical because an increment or decrement operator might be accidentally used to a variable in a condition statement, thus causing the value to be compared to a constant increase or decrease. So, if the test data test the maximum or minimum boundary of the condition statement, the fault can be detected. It is also sensible for ROR mutants to be detected by the proposed approach. Programmers sometimes get confused or mistaken between using the "greater/less than" or the "greater/less than or equal to" relational operators in a condition statement, thus creating an incorrect maximum or minimum boundary. This kind of error can indeed be detected by testing the maximum or minimum boundary.

The conventional MB-TCG was smaller in terms of test suite size because of their generated test data sequences. This result was expected because the proposed approach was implemented by taking an abstract test case and splitting it into several possible test data sequences to accommodate the combinations of input classifications. This finding was in tandem with the results obtained by Nguyen et al. [12]. The initial test suite size for the proposed approach was even worse, with 74 tests. The removal of duplicates reduced the number of test data sequences by 21. Still, it was improbable that the proposed approach could have the same test suite size as the conventional MB-TCG or less, given how tests are generated for the proposed approach. This finding motivated this study to propose an approach for model-based test case prioritization (MB-TCP) in future work. The generated test suite can be prioritized to further improve the effectiveness and efficiency of the test suite in terms of fault detection.

6 Conclusion and Future Work

This study proposes an automated TDG approach in MB-TCG using the EFSM model. It was motivated by the limitation in the existing MB-TCG approaches and the importance of TDG in MBT. The proposed approach by Nguyen et al. [12] was adopted and modified in this study by automating the domain input classifications step, taking advantage of the information available in an EFSM model. The experiment showed that the proposed approach is 6.62 percent more effective in fault detection

than the conventional MB-TCG based on their mutation scores. However, the conventional MB-TCG is more efficient with 43 fewer tests than the proposed approach.

For future work, an MB-TCP approach using the EFSM model will be proposed as a continuation of this study. This new MB-TCP approach will be applied to the generated tests to increase testing effectiveness and efficiency further. It will also serve as an effort to address the efficiency issue faced by the proposed TDG approach. Furthermore, more related metrics that represent testing effectiveness and efficiency, such as test generation time or execution time, will be used. Last but not least, more case studies will be utilized to increase the confidence level regarding the proposed TDG approach effectiveness. Nontrivial case studies will also be used to reflect real-world systems more, thus making the obtained results more generalizable.

Acknowledgement: The authors would like to express their deepest gratitude to the Software Engineering Research Group (SERG) members and the anonymous reviewers for their constructive comments and suggestions.

Funding Statement: The research was funded by Universiti Teknologi Malaysia (UTM) and the Malaysian Ministry of Higher Education (MOHE) under the Industry-International Incentive Grant Scheme (IIIGS) (Vote Number: Q.J130000.3651.02M67 and Q.J130000.3051.01M86), and the Academic Fellowship Scheme (SLAM).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] T. Y. Chen, F. -C. Kuo and R. Merkel, "On the statistical properties of testing effectiveness measures," *Journal of Systems and Software,* vol. 79*,* no. 5*,* pp. 591–601, 2006.

[2] S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson and D. Sundmark, "A framework for comparing efficiency, effectiveness and applicability of software testing techniques," in *Proc. of Testing: Academic & Industrial Conf.-Practice and Research Techniques (TAIC PART'06)*, Windsor, UK, pp. 159–170, 2006.

[3] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell *et al.,* "Taming google-scale continuous testing," in *Proc. of 39th Int. Conf. on Software Engineering: Software Engineering in Practice Track*, Buenos Aires, Argentina, pp. 233–242, 2017.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software,* vol. 86*,* no. 8*,* pp. 1978–2001, 2013.

[5] M. Utting, A. Pretschner and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability,* vol. 22*,* no. 5*,* pp. 297–312, 2012.

[6] A. Gambi, C. Mayr-Dorn and A. Zeller, "Model-based testing of end-user collaboration intensive systems," in *Proc. of Symposium on Applied Computing*, Marrakech, Morocco, pp. 1213–1218, 2017.

[7] W. Li, F. Le Gall and N. Spaseski, "A survey on model-based testing tools for test case generation," in *Proc. of Int. Conf. on Tools and Methods for Program Analysis*, Moscow, Russia, pp. 77–89, 2017.

[8] A. Saeed, S. H. Ab Hamid and A. A. Sani, "Cost and effectiveness of search-based techniques for model-based testing: An empirical analysis," *International Journal of Software Engineering and Knowledge Engineering,* vol. 27*,* no. 4*,* pp. 601–622, 2017.

[9] T. Ahmad, J. Iqbal, A. Ashraf, D. Truscan and I. Porres, "Model-based testing using UML activity diagrams: A systematic mapping study," *Computer Science Review,* vol. 33*,* pp. 98–112, 2019.

[10] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys,* vol. 43*,* no. 2*,* pp. 1–29, 2011.

[11] D. R. Kuhn, R. N. Kacker and Y. Lei, *Introduction to Combinatorial Testing,* Boca Raton, Florida: CRC press, 2013.
[12] C. D. Nguyen, A. Marchetto and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation," in *Proc. of Int. Symposium on Software Testing and Analysis*, Minneapolis, Minnesota, USA, pp. 100–110, 2012.
[13] Y. Chen, A. Wang, J. Wang, L. Liu, Y. Song *et al.,* "Automatic test transition paths generation approach from EFSM using state tree," in *Proc. of Int. Conf. on Software Quality, Reliability and Security Companion (QRS-C)*, Lisbon, Portugal, pp. 87–93, 2018.
[14] A. Turlea, F. Ipate and R. Lefticaru, "Generating complex paths for testing from an EFSM," in *Proc. of Int. Conf. on Software Quality, Reliability and Security Companion (QRS-C)*, Lisbon, Portugal, pp. 242–249, 2018.
[15] M. L. Mohd-Shafie, W. M. N. Wan-Kadir, M. Khatibsyarbini and M. A. Isa, "Model-based test case prioritization using selective and even-spread count-based methods with scrutinized ordering criterion," *PLOS One,* vol. 15*,* no. 2*,* pp. 1–27, 2020.
[16] B. Korel, L. H. Tahat and M. Harman, "Test prioritization using system models," in *Proc. of 21st Int. Conf. on Software Maintenance (ICSM'05)*, Budapest, Hungary, pp. 559–568, 2005.
[17] T. Shu, Z. Ding, M. Chen and J. Xia, "A heuristic transition executability analysis method for generating EFSM-specified protocol test sequences," *Information Sciences,* vol. 370–371*,* pp. 63–78, 2016,.
[18] P. Kamath and V. Narendra, "Generation of test cases from behavior model in UML," *International Journal of Applied Engineering Research,* vol. 13*,* no. 17*,* pp. 13178–13187, 2018.
[19] A. Andrews, M. Abdelgawad and A. Gario, "World model for testing urban search and rescue (USAR) robots using petri nets," in *Proc. of 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, Rome, Italy, pp. 663–670, 2016.
[20] S. Majeed and M. Ryu, "Model-based replay testing for event-driven software," in *Proc. of 31st Annual ACM Symposium on Applied Computing*, Pisa, Italy, pp. 1527–1533, 2016.
[21] K. Singi, D. Era and V. Kaulgud, "Model-based approach for automated test case generation from visual requirement specifications," in *Proc. of Eighth Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, Graz, Austria, pp. 1–6, 2015.
[22] J. Gutiérrez, M. Escalona and M. Mejías, "A model-driven approach for functional test case generation," *Journal of Systems and Software,* vol. 109*,* pp. 214–228, 2015.
[23] E. Sarmiento, J. C. S. do Prado Leite and E. Almentero, "C&L: Generating model based test cases from natural language requirements descriptions," in *Proc. of 1st Int. Workshop on Requirements Engineering and Testing*, Karlskrona, Sweden, pp. 32–38, 2014.
[24] S. Kansomkeat, P. Thiket and J. Offutt, "Generating test cases from UML activity diagrams using the condition-classification tree method," in *Proc. of 2nd Int. Conf. on Software Technology and Engineering*, San Juan, PR, USA, pp. 62–66, 2010.
[25] A. Gargantini and M. Radavelli, "Migrating combinatorial interaction test modeling and generation to the web," in *Proc. of Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, Vasteras, Sweden, pp. 308–317, 2018.
[26] L. Yu, Y. Lei, R. N. Kacker and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *Proc. of Sixth Int. Conf. on Software Testing, Verification and Validation*, Luxembourg, Luxembourg, pp. 370–375, 2013.
[27] T. Yano, E. Martins and F. L. de Sousa, "MOST: A multi-objective search-based testing from EFSM," in *Proc. of Fourth Int. Conf. on Software Testing, Verification and Validation Workshops*, Berlin, Germany, pp. 164–173, 2011.
[28] S. Wong, C. Y. Ooi, Y. W. Hau, M. N. Marsono and N. Shaikh-Husin, "Feasible transition path generation for EFSM-based system testing," in *Proc. of Int. Symposium on Circuits and Systems (ISCAS)*, Beijing, China, pp. 1724–1727, 2013.
[29] A. Wood, "Software reliability growth models," Technical report, Cupertino, California, USA, 1996.
[30] P. C. Jorgensen, *The Craft of Model-Based Testing,* Boca Raton, Florida: CRC Press, 2017.

[31] Y. S. Ma, J. Offutt and Y. R. Kwon, "Mujava: An automated class mutation system," *Software Testing, Verification and Reliability,* vol. 15*,* no. 2*,* pp. 97–133, 2005.
[32] M. L. Mohd-Shafie, W. M. N. W. Kadir, H. Lichter, M. Khatibsyarbini and M. A. Isa, "Model-based test case generation and prioritization: A systematic literature review," *Software and Systems Modeling,* pp. 1–37, 2021. https://doi.org/10.1007/s10270-021-00924-8.
[33] L. Tahat, B. Korel, G. Koutsogiannakis and N. Almasri, "State-based models in regression test suite prioritization," *Software Quality Journal,* vol. 25*,* no. 3*,* pp. 703–742, 2017.
[34] A. S. Kalaji, R. M. Hierons and S. Swift, "An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models," *Information and Software Technology,* vol. 53*,* no. 12*,* pp. 1297–1318, 2011.
[35] É. F. De Souza, V. A. de Santiago Júnior and N. L. Vijaykumar, "H-switch cover: A new test criterion to generate test case from finite state machines," *Software Quality Journal,* vol. 25*,* no. 2*,* pp. 373–405, 2017.
[36] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering,* vol. 37*,* no. 5*,* pp. 649–678, 2010.
[37] M. H. Chen, M. R. Lyu and W. E. Wong, "Effect of code coverage on software reliability measurement," *IEEE Transactions on Reliability,* vol. 50*,* no. 2*,* pp. 165–170, 2001.
[38] R. Gopinath, C. Jensen and A. Groce, "Code coverage for suite evaluation by developers," in *Proc. of 36th Int. Conf. on Software Engineering*, Hyderabad, India, pp. 72–82, 2014.