Test Case Prioritization based on Extended Finite State Machine Model

Muhammad Luqman Shafie and Wan M. N. Wan Kadir
Department of Software Engineering, Faculty of Computing,
Universiti Teknologi Malaysia, 81310 Johor Bahru, Johor, Malaysia.
luqman_1993@yahoo.com.my

***Abstract*—Regression testing is done to test the modified version of a software, however re-testing all test cases are very inefficient. Test Case Prioritization is one of the techniques used to overcome this problem. It prioritizes the test cases in the test suite by ordering them according to a desired objective goal like revealing faults earlier and has various approaches in performing it. One of them is model-based approach which utilizes the system model to make prioritization. The advantages of this approach are cheaper execution cost and lesser prioritization time compared to code-based prioritization. In this paper, we propose a model-based test case prioritization approach using extended finite state machine. The proposed approach will be based on several related existing approaches with an extra criterion of degree of code changes. The contribution of the proposed approach is it overcomes the identified limitations from the related works and improve the prioritization result.**

***Index Terms*—Extended Finite State Machine; Model-Based; Regression Testing; Test Case Prioritization Technique.**

I. INTRODUCTION

A software system that has passed the development phases cannot be counted as completely done because changes in a software system will continuously occur over time and are inevitable due to ever changing environment [1]. Parts of the system such as architectures, requirements and functions are modified, added and discarded to satisfy the changes. When changes are implemented to a system, its test cases are rerun for testing purpose to ensure that no new defects are introduced during the modification. This phase is necessary to ensure that the quality of the system is in top notch [1] and is particularly known as Regression Testing. The prime purpose of Regression Testing is to make sure that modification and changes made to the particular software system did not create any negative impacts on it [2].

The main issue in Regression Testing is that it has been proved to be among the costliest phases in a software development life cycle [3]. Hall et al. [4] claims that almost 80% of testing budget is dominated by Regression Testing. This circumstance arises mostly because software always undergoes modifications and new version is released from time to time to cope with these changes. As a result, the test suite will have the tendency to grow in size because new test cases might be added to cover the modified or added elements for the testing purpose [5]. As a consequence, the testing cost will obviously escalate endlessly and there will be a point where re-testing all the test suite will not be relevant anymore. Furthermore, Regression Testing also requires a lot of times in the process. A report of an industrial collaborator is the prove of these issues stating that one of its products consisting of 20,000 lines of code involves an extremely long period of seven weeks for the whole test suite to be executed [6]. This inconvenient situation without a doubt will jeopardize the testing phase significantly in many factors.

For these particular reasons, researchers had come up with assorted techniques in order to solve this issue. In their survey, Yoo et al. [2] explain and cluster Regression Testing techniques into three main categories which are Test Suite Minimization (TSM), Test Case Selection (TCS) and Test Case Prioritization (TCP). TCM discards any outdated or unnecessary test cases permanently from the test suite [7] while TCS select relevant test cases from the test suite according to certain criteria [8]. Lastly is the TCP which re-arranges the test cases from the original test suite based on a specified purpose in a manner that the test cases that serve the purpose the most are given the highest priority [9]. All of these techniques possess their advantages and limitations in Regression Testing. However, in this paper we will be focusing on TCP technique, more specifically the model-based approach.

Korel et al. [10] are among the earliest researchers that propose Model-based TCP which implement a different approach in prioritizing test cases than the commonly used code-based techniques. In this particular approach, the system model is utilized to prioritize test cases instead of the system code itself. State machine diagram, activity diagram and sequence diagram are some of the basic models used in model-based prioritization. The advantage of model-based over code-based is cheaper execution cost [10]. Analyzing models would be faster than the source code and early feedback can be achieved since models are made before source code is implemented [3]. Despite that, the number of papers published related to developing new model-based approach is relatively low throughout the years. According to a study conducted, only four papers proposed or used model-based approach between 2001 and 2009 and six were published during 2009 and 2010 period [3]. Therefore, the aim of this paper is to propose a model-based test cast prioritization approach using Extended Finite State Machine.

The remainder of this paper presents our technical paper of the model-based Test Case Prioritization approach. A comprehensive elaboration of Extended Finite State Machine and Model-based Testing is given in Section II and III respectively. A detailed elaboration regarding model-based TCP is presented in Section IV. Section V shows the related works in model-based approach. Our proposed technique was presented in in Section VI while the empirical study is shown in Section VII. Lastly, Section VIII will conclude the technical paper.

II. Extended Finite State Machine

The model that we are focusing in this study is the Extended Finite State Machine. The traditional Finite State Machine (FSM) is extended by Extended FSM (EFSM) by context variables, input parameters, output parameters and conditions when a transition can be fired. EFSM comprises of states and transitions between states [10]. Transitions consist of an event, a condition and a sequence of actions. A particular transition is executed when a specified event occurs and when an enabling condition related to the transition evaluates to true. As a result of a transition's execution, actions associated to it are also executed. The formal definition of an EFSM taken from Tahat et al. [11] is a 7-tuple $M = (\Sigma, Q, Start, Exit, V, O, R')$ where Σ denotes the set of events, Q denotes the set of states, $Start \in Q$ denotes the start state, $Exit \in Q$ denotes the stop state, V denotes a finite set of variables, O denotes the set of actions and R′ is the set of transitions. Each transition T is denoted by the tuple: $T = (E, C, A, S_b, S_e)$ where $E \in \Sigma$ is an event, C denotes an enabling condition described over V, A is a series of actions, $A = \langle a_1, a_2, \dots, a_j \rangle$, where $a_i \in O$, $S_b \in Q$ denotes transition's starting state, $S_e \in Q$ is the transition's exiting state.

For instance, an example of EFSM quoted from Tahat et al. [12] is shown. Figure 1 illustrates an EFSM model of a simplified ATM system which provides three types of functionalities: balance inquiry, withdrawal and deposit. These experimental functions do not represent a real-world operation of an ATM system. To start a transaction, a user must enter a bank card which contains the actual PIN number and money balance represented by the event $Card(x, y)$ where transition T_1 is triggered. To enter the main menu, a matching prompted Pin number with the actual one must be entered with a maximum of three attempts. For example, transition T_2 is executed when the system is at state S1 and event $PIN(p)$ is received, the enabling conditions are true when a mismatch PIN is entered where $(p! = pin)$ and $(attempts < 3)$, next the series of actions are displaying error message, increment $attempts$ value by one and prompt user to enter another PIN number.

III. Model-based Testing

The aim of testing in the context of software engineering is to show that whether the intended and the actual behaviors of a system differ or not and to gain confidence that that they do not [13]. In general terms, failure detection is the major goal of testing which is done by searching the noticeable distinction between the behaviors of implementation and the planned behaviors of the system under test (SUT), as indicated by its requirements. Shafique et al. [14] states that software testing which is the evaluation of a SUT by spotting its executions on valued inputs is probably the most commonly utilized verification technique. Black-box testing and white-box testing are the two main categories of software testing depending on whether they rely entirely on the specifications of the SUT or exclusively on its implementation. Model-based Testing (MBT) is a branch of software testing that relies on the exact behavior models that encode the intended behaviors of the SUT which is a black-box testing. In MBT, a SUT models can be utilized to automatically generate test cases, unlike conventional testing where each test case must be coded by the test engineer [15].

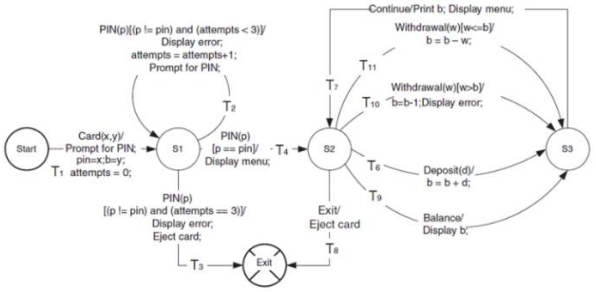

Figure 1: Example EFSM model of a simplified Automated Teller Machine system

The process of MBT consists of several crucial steps in it [13]. The first step is (1) building the models which are often called as test model are built from the informal requirements or specification documents of the SUT. It is crucial that the test models to be simpler (more abstract) than the SUT or else the attempt to validate the models would be equivalent to the efforts of validating the SUT itself. The second step as addressed by Utting et al. [13] is (2) to choose test selection criteria. This step is done to guide the automatic test generation in order to produces a quality test suite, the one that can satisfy the test policy described for the SUT. After the test selection criteria are chosen, they are (3) transformed into test case specifications that describe the notion of test selection criteria and turn them to be operational.

The fourth step in MBT is where (4) a set of test cases is generated using automatic test case generator given model and a test case specification, which aims is to satisfy all of the test case specifications. Finally, (5) once the test suite has been generated, it is run. The execution may be manual which is run by a person, or may be automated by a test execution environment that support the ability to automatically execute the tests and record their verdicts. In the process of running the test cases, the test inputs are first concretized and then the concrete data are sent to the SUT. Then, the resulting concrete outputs from the SUT will be abstracted to obtain the high-level actual result. This actual result will be compared with the expected result to determine the verdict. This process of concretization and abstraction is the duty delegated to a component called adapter. Figure 2 illustrates the overall process of MBT with the corresponding steps labelled.

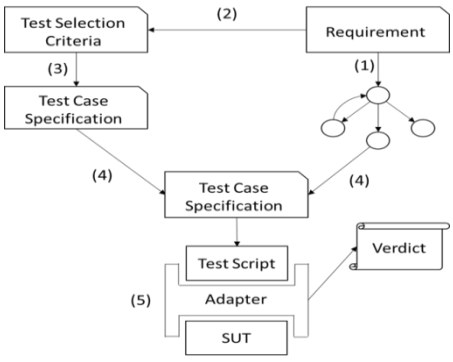

Figure 2: MBT overall process

IV. Model-based Test Case Prioritization

Test Case Prioritization is a technique under regression testing in which test cases are re-ordered from the original test

suite according to a particular purpose in a manner that the test cases that serve the purpose the most is given the highest priority [9]. We took the definition of Test Case Prioritization problem proposed by Elbaum et al. [16] into consideration for this systematic review which is stated below:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real number.

Problem: Find T ∈ PT such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')] \quad (1)$$

In this definition, PT serves as the set of all possible sequences of T, while f is the function when implemented to any of the sequences, yields an award value for that particular sequence. In short, the definition expect that the higher award values are more preferable than the lower ones. There are a number of possible goals when referring to prioritization in this context. Elbaum et al. [16] also states some of the goals in their study which are (1) to increase the rate of early faults detection when executing test suite, (2) to increase the code coverage under test at a faster pace when executing test suite, (3) to increase their confidence in the system's reliability at a faster rate and (4) to increase possibility of revealing faults associated to particular code changes earlier in testing process.

Over time, researchers have proposed numerous approaches for Test Case Prioritization. All of these approaches can be divided into two main categories which are code-based and model based. In code-based Test Case Prioritization, test cases are prioritized by utilizing the source code information of the software system. A survey conducted by Mahdian et al. [5] states that the vast majority of test selection strategies are code-based. A study carried out by Catal et al. [3] also proved that the most investigated prioritization method was coverage-based that conquered 40 percent of all the various techniques they had gathered. Coverage-based is a kind of code-based prioritization where the more coverage achieved by a test suite, the more chances faults can be revealed earlier during testing process. Coverage in this context means the code coverage of the software system for example statement, function or code block. The downside of code-based is that code knowledge is needed in order to prioritize test cases [5] which means prioritization cannot begin until the source code is available. Another drawback of code-based is that most of them are language dependent [5] so testing process will become troublesome in cases where the program is written in various programming languages.

On the other hand, model-based prioritization manipulates the model of the software system to perform prioritization [17]. Generally, any kind of Test Case Prioritization approach that uses the system model in it can be categorized as model-based approach. Some examples of system models are use case diagram, sequence diagram, state machine diagram and activity diagram. The primary advantage of model-based prioritization is that execution of the system models is rather faster than the execution of the system codes itself during testing [10]. This is because system models are at a higher level of abstraction thus capturing system's behavior and structure is less complex compared with the source code [11]. Therefore, model-based prioritization is considerably inexpensive compared to code-based prioritization which is both resource-wise and time-wise [10]. Nevertheless, model-based prioritization also possesses their own weaknesses. One of the major flaws is its dependence on the correctness and completeness of the system models [18]. As the space is limited, this topic regarding model-based weaknesses will not be discussed in this study.

V. Related Works

Al-Herz et al. [19] proposed three approaches in their study. One of them is named Degree Measure Method (DMM) which utilized the Object Relation Diagram (ORD) model which represents the design structure of web application. This particular technique ranks components based on fan-in degree then prioritizes test cases that cover the highest ranked components. Fan in degree in this context means the number of components that lead to this particular component. The rationale behind this technique is that most of the other components will fail to get services if this high fan-in degree component break down [19]. The flaw in this technique is which one should be given highest priority when two components have the same fan-in degree. Their suggestion to solve this problem is by adding more criteria such as component type and fan-in edge type.

In addition, an approach namely Model Dependence-based Test Prioritization was invented by Korel et al. [10] which make use of Extended Finite State Machine (EFSM) to perform prioritization. This approach was elaborated by them in further details in their extended version of studies for modification made both in software system and models and for modification for which models are not modified (only source code is modified) [11, 12]. In brief, this approach utilizes the model dependence analysis to determine the patterns of how added and deleted transitions communicate with the modified model and lastly utilizes this information to prioritize test cases. Despite that, this approach increases execution time because it needs more analysis and gather extra information from the model from other models proposed by them. Furthermore, the whole model execution trace must be stored to compute the interaction patterns thus raising resources usage.

Another approach in model-based TCP is by using the Structural Aspects of Use Case & Activity Diagram proposed by Sapna et al. [18]. In their approach, the UML model use case diagram and activity diagram are used as the input for prioritization. The process starts with capturing data from all use case diagrams to calculate use case priority. Next, scenarios are extracted from activity diagram and assigned weights to their nodes and edges. The weight of path (scenario) is calculated then finally prioritize by summing the sum of the priorities starting at level 1 of the schema and moving down adding the weights of all the nodes up to the scenario weight. The downside in this approach is its dependence on the correctness and completeness of the use case diagram and activity diagram. For example, if the activity diagram is not complete, there will be possibilities where some requirements are not captured. As the result, the scenarios will not be generated and this will affect the overall prioritization.

VI. Proposed Approach

Before elaborating the proposed approach, some related existing approaches proposed by Tahat et al. [11] are clarified for further understanding in the sub sections. The formal

definitions which will be used throughout the following sub sections are described as follow. TC_H is the set with high priority tests, TC_L is the set with low priority tests, $TC_H \cap TC_L = \emptyset$, $TS = \langle t_1, t_2, \dots, t_N \rangle$ is an ordered test suite of size N tests, t_i is a test case, i is the test case number, TS_P is a prioritized ordered test suite, $R' = \{T_1, T_2, \dots, T_j, T_O\}$ is the set of all transitions of size O transitions and j is the transition number, MT is a set of all modified transitions, $S(t_i)$ is a sequence of transitions traversed by t_i, $A(t_i)$ is a set of modified transitions executed by test t_i. Example 1 described below is utilized to show the example of possible result of prioritization for the approaches mentioned later.

Example 1: Suppose that $MT = \{T_1, T_2, T_3, T_4, T_5\}$, $TS = \langle t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \rangle$. For each test the following modified transitions are traversed, $(t_1: T_1, T_2, T_1, T_2, T_3)$ $(t_2: T_3, T_4, T_5)$ $(t_3: T_3, T_4)$ $(t_4: T_5)$ $(t_5: T_1)$ $(t_6: T_1, T_2, T_1, T_2)$ $(t_7: T_2, T_4)$ $(t_8: T_3, T_4, T_2, T_4)$ $(t_8: \emptyset)$ $(t_9: \emptyset)$. Suppose $A(t_1) = \{T_1, T_2, T_3\}$; $A(t_2) = \{T_3, T_4, T_5\}$; $A(t_3) = \{T_3, T_4\}$; $A(t_4) = \{T_5\}$; $A(t_5) = \{T_1\}$; $A(t_6) = \{T_1, T_2\}$; $A(t_7) = \{T_2, T_4\}$; $A(t_8) = \{T_2, T_3, T_4\}$; $A(t_9) = \emptyset$; $A(t_{10}) = \emptyset$.

A. Selective Test Prioritization #1

In this approach, high priority is assigned to tests that executed modified transitions in the model. Which means if a test case includes at least one modified transition in its execution, the test case will be given high priority. On the other hand, low priority is assigned to the test cases that does not execute any modified transition in its execution. In case if during the execution of a test t_i, transition T_j which is a modified transition, $T_j \in MT$, is traversed, the t_i will be put into the high priority set, $t_i \in TC_H$. If not, the test case is assigned to low priority set, $t_i \in TC_L$. All test cases in high priority set will be executed first and randomly then all the test cases in low priority set will be executed randomly. . Therefore, a possible prioritized test suite using this particular approach will be $TS_P = \langle t_2, t_1, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \rangle$.

B. Even Spread Count-based Test Prioritization #2

The idea of this approach is that all modified transitions must be given same chance to be covered during testing. Meaning that it tries to balance the number of executions of modified transitions during testing. Higher priority is given to a test that executes a modified transition which is traversed the least number of times at any point of testing.

An additional definition of a function $count(T, S')$ that describes the number of tests in S' on which T is executed is used in this approach. The equation of the function is:

$$count(T, S') = \sum_{j=1}^{m} F\left(T, t_{i_j}\right) \quad (2)$$

where function $F(T, t_{i_j})$ returns 1 if $T \in A(t_i)$, if not 0 is returned. Let assumes T_i and T_k are two modified transitions. T_k would has a higher priority than T_i if $count(T_k, S') < count(T_j, S')$ which means transition with lowest $count(T, S')$ will be the highest priority.

Refer to Example 1. Suppose firstly the approach select and execute test t_4 containing T_5 from its algorithm which is, select randomly test $t_i \in A(t_i)$ for which $A(t_i) \neq \emptyset$ and remove t_i from TS. Then transition counts for each modified transition are then updated as follow: $count(T_1) = 0, count(T_2) = 0, count(T_3) = 0, count(T_4) = 0, count(T_5) = 1$. Then, the approach identified four modified transitions with minimum count, $E = \{T_1, T_2, T_3, T_4\}$. Next, suppose the approach select T_2, the tests that execute T_2 are identified, $T_2: t_1, t_6, t_7, t_8$. Assumes test t_6 is selected and executed, the transition counts for each modified transition are updated again as follow: $count(T_1) = 1, count(T_2) = 1, count(T_3) = 0, count(T_4) = 0, count(T_5) = 1$. Then, the approach identified two modified transitions with minimum count, $E = \{T_3, T_4\}$. Next, suppose the approach select T_4, the tests that execute T_4 are identified, $T_4: t_2, t_3, t_7, t_8$. Assumes test t_3 is selected and executed, the transition counts for each modified transition are updated again as follow: $count(T_1) = 1, count(T_2) = 1, count(T_3) = 1, count(T_4) = 1, count(T_5) = 1$. The approach will continue looping until all tests that triggered at least one marked transition is selected. Lastly, t_{10} and t_9 that do not traverse any modified transition are executed randomly. Therefore, a possible prioritized test suite using this particular approach will be $TS_P = \langle t_4, t_6, t_3, t_2, t_1, t_5, t_8, t_7, t_9, t_{10} \rangle$.

C. Proposed Approach

The proposed approach is inspired by these approaches mentioned earlier while trying to overcome limitations discussed in the related works. The idea of the proposed approach is that higher priority is assigned to tests that executed more modified transitions in the model while balancing the number of executions of modified transitions during testing. A modified transition will also be assigned a degree of code changes where the higher the degree of code changes of a transition, the higher its priority will be. Modified transitions that have high degree of code changes executed the least number of times will be given higher priority. Figure 3 illustrated the overall implementation of the proposed approach for a better visualization.

For clarification purpose, refer to Example 1. Suppose the transition score $ScT(T_j)$ for each modified transition is calculated where $ScT(T_1) = 2$; $ScT(T_2) = 1$; $ScT(T_3) = 1$; $ScT(T_4) = 1$; $ScT(T_5) = 3$. Then the test case score $Sct(t_i)$ for each test case is calculated based on transition score calculated earlier where $Sct(t_1) = 4$; $Sct(t_2) = 5$; $Sct(t_3) = 2$; $Sct(t_4) = 3$; $Sct(t_5) = 2$; $Sct(t_6) = 3$; $Sct(t_7) = 2$; $Sct(t_8) = 3$; $Sct(t_9) = 0$; $Sct(t_{10}) = 0$. It can be observed that $Sct(t_2)$ has the highest value therefore it will be appended first into the last position of the prioritized test suite, $TS_P = \langle t_2 \rangle$. Next the set E where modified transitions that have been appended into TS_P are determined where $E = \{T_3, T_4, T_5\}$. Then the test case score $Sct(t)$ for each test case is updated. If a test case in TS contains the modified transitions in set E, then the transition score of those modified transitions in the test case will be eliminated. Thus, $Sct(t_1) = 3$; $Sct(t_2) = 0$; $Sct(t_3) = 0$; $Sct(t_4) = 0$; $Sct(t_5) = 2$; $Sct(t_6) = 3$; $Sct(t_7) = 1$; $Sct(t_8) = 1$; $Sct(t_9) = 0$; $Sct(t_{10}) = 0$. Based on the updated $Sct(t)$ values, it can be observed that $Sct(t_1)$ and $Sct(t_6)$ has the highest value. Therefore, one random test case between these two is appended into the last position of the prioritized test suite TS_P and assume that t_1 is chosen, then $TS_P = \langle t_2, t_1 \rangle$. The set E will be updated as $E = \{T_3, T_4, T_5, T_1, T_2\}$. The updated test case score will be $Sct(t_1) = 0$; $Sct(t_2) = 0$; $Sct(t_3) = 0$; $Sct(t_4) = 0$; $Sct(t_5) = 0$; $Sct(t_6) = 0$; $Sct(t_7) = 0$;

$Sct(t_8) = 0; Sct(t_9) = 0; Sct(t_{10}) = 0$. Considering that all test cases scores are 0, the remaining test cases in TS will be selected randomly to be appended in TS_P. Therefore, a possible prioritized test suite using this particular approach will be $TS_P = \langle t_2, t_1, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \rangle$.

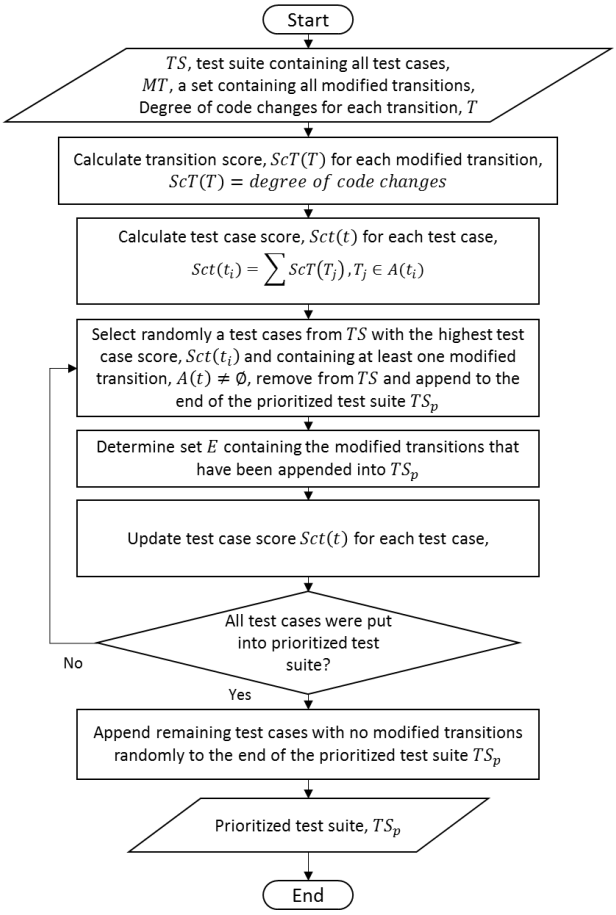

Figure 3: Flowchart of the proposed approach

The results of a possible prioritized test suite for all existing approaches including the proposed approach obtained using Example 1 are shown in the Table 1. Column three shows the number of test cases required for all modified transitions to be covered completely. From the result, it can be observed that the proposed approach requires the smallest number of test cases to cover all modified transition completely which are two test cases. Here we hypothesize that when an approach can produce a test suite that covers all modified transitions faster, all bugs in the system can be detected more effectively. To prove this assumption, an experiment has been done which is explained in the next section.

Table 1
Prioritized test suite for Example 1

Approach	Possible Prioritized Test Suite	Num. of t required for 100% T cov.
#1	$TS = \langle t_4, t_1, t_5, t_2, t_3, t_6, t_8, t_7, t_9, t_{10} \rangle$	4
#2	$TS_P = \langle t_4, t_6, t_3, t_2, t_1, t_5, t_8, t_7, t_9, t_{10} \rangle$	3
P. Approach	$TS_P = \langle t_2, t_1, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10} \rangle$	

VII. EMPIRICAL STUDY

The aim of this empirical study is to evaluate and compare the effectiveness of early faults detection of prioritized test suite resulted from the implementation of the prioritization approaches presented earlier: selective test prioritization, even spread count-based test prioritization and the proposed approach. This experiment is also aimed to prove that when all modified transitions are covered faster in the test suite, the detections of all faults would also be quicker. We created a simple ATM model which we referred the ATM model in Figure 1 and made some modifications. In our model, a user can only deposit money once for a particular session and a user's maximum balance can only be less than or equal 100.

The system's EFSM model is constructed using Spec Explorer tool. A model program which is a set of rules written in Spec# [20], an extension of C# is where the model's behaviours are described. Using a complete model program, the tool produces a set of test cases using path coverage where all transitions in the model will be traversed by at least one test case. Using the test suite obtained, the tool then generates the test code that can be run with the implementation of SUT which is created using C#. Faults are seeded into the implementation by making incorrect modifications while the model is not modified. For each function where the faults are seeded, its corresponding transition is marked as modified transition. Mutation testing technique is used to seed faults using value mutations, decision mutations and statement mutations. Figure 4 shows the partial model of login scenario generated by the tool using the model program written for the ATM implementation. The test input for function $insertCardT1(int\ pin, int\ balance)$ and $promptPinT2(int\ enteredPin)$ are as follow: $pin = 1$, $balance = 50$, $enteredPin = 1{,}2$. Figure 5 shows the test cases produced by the tool by exploring all the possible paths in the model. The full model is not shown because the size is considerably large with many possible paths and states but it will be used for the evaluation purpose.

The full model consists of 10 transitions where each of them represents a function in the implementation, $R' = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}\}$. A total of six faults are seeded into the implementation of the ATM where only one fault is seeded at a time to observe which test cases detect it. A total of 19 test cases are generated from the full model of the ATM. The modified transitions where the faults are seeded are $MT = \{T_5, T_6, T_7, T_8\}$. The transition score for the modified transitions are: $Sc(T_5) = 3$, $Sc(T_6) = 1$, $Sc(T_7) = 1$, $Sc(T_8) = 2$. The modified transitions traversed by each test case is as follow: $A(t_1) = \emptyset$, $A(t_2) = \{T_5, T_8\}$, $A(t_3) = \{T_5, T_8\}$, $A(t_4) = \{T_5, T_7, T_8\}$, $A(t_5) = \{T_5, T_7, T_8\}$, $A(t_6) = \{T_5, T_7, T_8\}$, $A(t_7) = \{T_5, T_7, T_8\}$, $A(t_8) = \{T_5, T_7, T_8\}$, $A(t_9) = \{T_5, T_6, T_7, T_8\}$, $A(t_{10}) = \{T_5, T_6, T_7, T_8\}$, $A(t_{11}) = \{T_5, T_6, T_8\}$, $A(t_{12}) = \{T_5\}$, $A(t_{13}) = \{T_5, T_7, T_8\}$, $A(t_{14}) = \{T_5, T_7\}$, $A(t_{15}) = \{T_5, T_7\}$, $A(t_{16}) = \{T_5, T_6, T_7\}$, $A(t_{17}) = \{T_5, T_6\}$, $A(t_{18}) = \{T_5\}$, $A(t_{19}) = \{T_5\}$. Table 2 depicts the faults with the corresponding test cases that detect them.

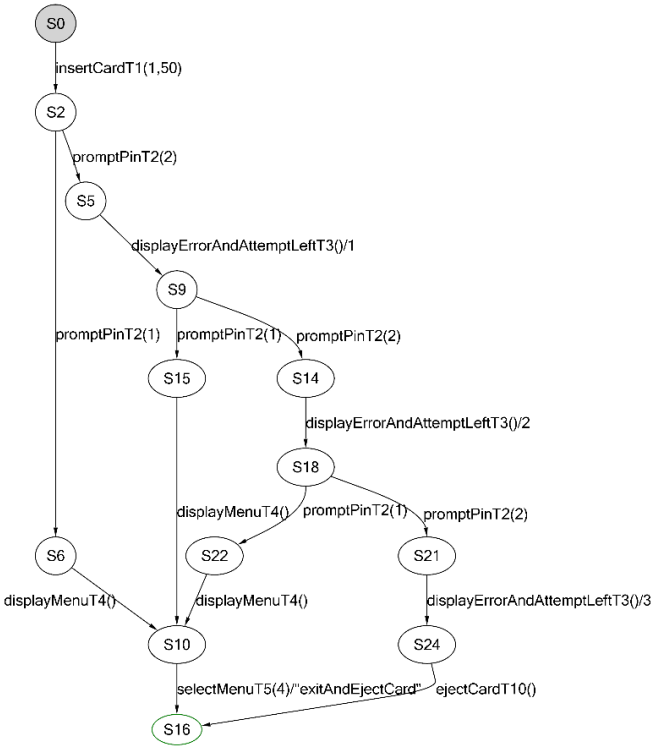

Figure 4: Model of login scenario

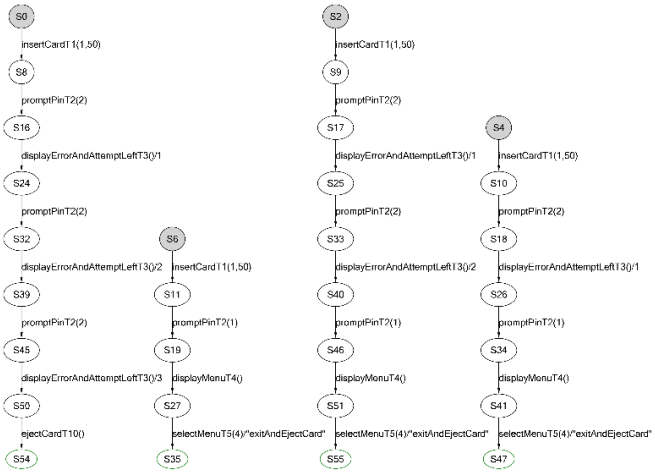

Figure 5: Test cases for login scenario

Table 2.
Faults detected by test cases

		Test case																		
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Fault	1	✓	X	X	X	X	X	X	X	X	X	X	✓	X	X	X	X	✓	✓	✓
	2	✓	X	X	X	X	X	X	X	X	X	X	✓	✓	✓	✓	✓	✓	✓	✓
	3	✓	X	X	X	X	X	X	X	X	X	X	✓	X	X	X	X	✓	✓	✓
	4	✓	✓	✓	✓	✓	✓	✓	✓	X	X	X	✓	✓	✓	✓	X	X	✓	✓
	5	✓	✓	✓	X	✓	X	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	6	✓	✓	✓	✓	✓	✓	✓	✓	X	X	X	✓	✓	✓	✓	X	X	✓	✓

Using the information obtained, we implement the prioritization approaches mentioned earlier to get one possible prioritized test suite TS_P that can be generated by each approach. Then, the prioritized test suites are evaluated by how rapid the they can detect faults using the Average of the Percentage of Faults Detected (APFD) [16]. APFD is a metric used to quantify how effective a prioritized test suite detects faults. The value of APFD result range from 0 to 100 where higher value means better faults detection rates. The equation for calculating the APFD value acquired from Elbaum et al. [21] is shown as follows where *T* is a test suite containing *n* test cases an *F* is a set of *m* faults revealed by *T*. TF_i is the first test case in ordering *T'* of *T* which reveals fault *i* and the APFD value of *T'* is:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \qquad (3)$$

Note that calculation the APFD can only be accomplished when advance knowledge of faults and test cases result is available. Therefore, it can only be used for evaluation purpose to determine the effectiveness of an approach in revealing faults so that in the future the best approach can be utilized to increase the possibility of revealing faults earlier during testing where the position of faults is unknown. Table 3 shows the prioritized test suite resulted from the implementation of the prioritization approaches to the ATM model. Column three shows the number of test cases required for all modified transitions to be covered completely.

Table 3: Prioritized test suite for ATM model

Approach	Possible Prioritized Test Suite	Num. of *t* required for 100% *T* cov.
#1	$TS = \langle t_{13}, t_{15}, t_{12}, t_{18}, t_4, t_2, t_{19}, t_{14}, t_9, t_{10}, t_7, t_8, t_{11}, t_{16}, t_6, t_{17}, t_3, t_5, t_1 \rangle$	9
#2	$TS = \langle t_{18}, t_{17}, t_{15}, t_2, t_{10}, t_{16}, t_6, t_{11}, t_9, t_{14}, t_4, t_7, t_8, t_3, t_{13}, t_5, t_{12}, t_{19}, t_1 \rangle$	4
P. Approach	$TS = \langle t_9, t_{15}, t_{18}, t_4, t_2, t_8, t_7, t_{19}, t_{16}, t_5, t_1, t_{17}, t_{10}, t_6, t_{11}, t_3, t_{12}, t_{13}, t_{14} \rangle$	1

The calculation of APFD for each prioritized test suite is shown as follows:

$$APFD(\#1) = 1 - \frac{1+5+1+9+5+9}{(19)(6)} + \frac{1}{2(19)} = 0.76$$

$$APFD(\#2) = 1 - \frac{3+4+3+2+7+2}{(19)(6)} + \frac{1}{2(19)} = 0.84$$

$$APFD(P.A.) = 1 - \frac{1+1+1+1+4+1}{(19)(6)} + \frac{1}{2(19)} = 0.94$$

From the calculation above, proposed approach has the highest detection rate of 94 % followed by even spread count-based test prioritization and selective test prioritization with 84% and 76 % detection rate respectively. Figure 6 – 8 illustrate percentage of test suite executed over the percentage of faults detected to visualize the progress of test cases execution using the prioritized test suites. By observing Table 3, the proposed approach requires only one test case to cover all modified transitions while the selective test prioritization and even spread count-based test prioritization require nine and four test cases respectively. Based on the APFD result and the graph, the proposed approach clearly outperforms the other two approaches in term of early faults detection. The proposed approach only requires to execute four test cases to capture all the faults seeded while the selective test prioritization and even spread count-based test prioritization require nine and seven test cases respectively. Therefore, the

result approves our assumption that when an approach can produce a test suite that covers all modified transitions faster, all bugs in the system can be detected more effectively.

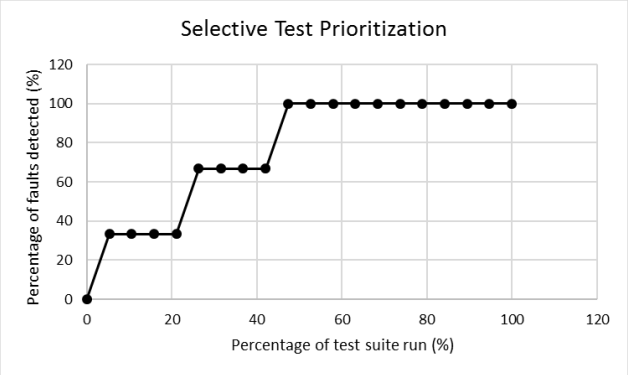

Figure 6: APFD graph for selective test prioritization approach

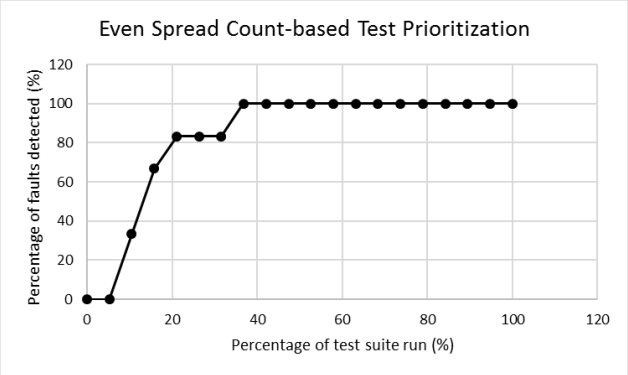

Figure 7: APFD graph for even spread count-based test prioritization

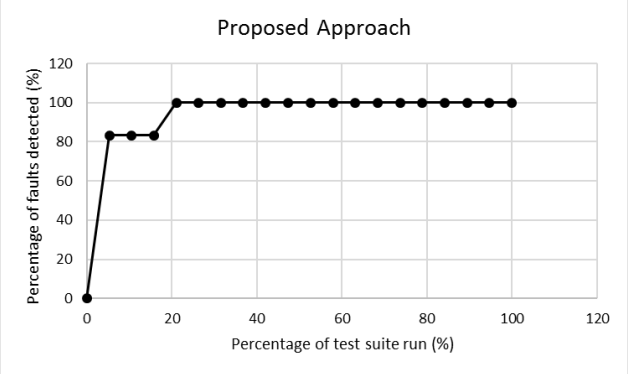

Figure 8: APFD graph for proposed approach

VIII. CONCLUSION

This paper presents a model-based approach in Test Case Prioritization using EFSM. A brief description on each of the related subjects, such as EFSM, model-based testing and model-based TCP is provided. In order to identify the gaps in the existing approaches, a number of related works in model-based TCP are critically reviewed. These approaches are considered as theoretical basis or foundations for the proposed approach. Next, we present our proposed approach that aims at improving the limitations found in the related work. An empirical study is conducted to evaluate the effectiveness of the proposed approach and to prove that when an approach can produce a test suite that covers all modified transitions faster, all bugs in the system can be detected more effectively. The result obtained showed that the assumption is correct and the proposed approach outperforms the other two existing approaches in term of faults detection. For future works, the proposed approach will be experimented using publicly available dataset of larger size and compared will more approaches to evaluate its effectiveness.

ACKNOWLEDGEMENT

The authors would like to express their deepest gratitude to Ministry of Higher Education Malaysia (MOHE) for their financial support under Fundamental Research Grant Scheme (Vot number R.J130000.7816.4F824).

REFERENCES

[1] M. Rava and W. M. Wan-Kadir, "A review on prioritization techniques in regression testing," *International Journal of Software Engineering and Its Applications*, vol. 10, no. 1, pp. 221-232, 2016.

[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] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal*, vol. 21, no. 3, pp. 445-478, 2013.

[4] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1276-1304, 2012.

[5] A. Mahdian, A. A. Andrews, and O. J. Pilskalns, "Regression testing with UML software designs: a survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 4, pp. 253-286, 2009.

[6] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, and S. Kanduri, "Understanding the effects of changes on the cost-effectiveness of regression testing techniques," *Software Testing, Verification and Reliability*, vol. 13, no. 2, pp. 65-83, 2003.

[7] C.-T. Lin, K.-W. Tang, and G. M. Kapfhammer, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information and Software Technology*, vol. 56, no. 10, pp. 1322-1344, 2014.

[8] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler, "An evaluation of combination strategies for test case selection," *Empirical Software Engineering*, vol. 11, no. 4, pp. 583-611, 2006.

[9] Y. Singh, A. Kaur, B. Suri, and S. Singhal, "Systematic Literature Review on Regression Test Prioritization Techniques," *Informatica (Slovenia)*, vol. 36, no. 4, pp. 379-408, 2012.

[10] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 559-568.

[11] 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, 2016.

[12] L. Tahat, B. Korel, M. Harman, and H. Ural, "Regression test suite prioritization using system models," *Software Testing, Verification and Reliability*, vol. 22, no. 7, pp. 481-506, 2012.

[13] 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.

[14] M. Shafique and Y. Labiche, "A systematic review of state-based test tools," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 1, pp. 59-76, 2015.

[15] J. Ernits, R. Roo, J. Jacky, and M. Veanes, "Model-based testing of web applications using NModel," in *Testing of Software and Communication Systems*. 2009, pp. 211-216.

[16] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "*Prioritizing test cases for regression testing*," in *ISSTA '00 Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000, pp. 102-112.

[17] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Application of system models in regression test suite prioritization," in *IEEE International Conference on Software Maintenance*, 2008, pp. 247-256.

[18] P. Sapna and H. Mohanty, "Prioritizing use cases to aid ordering of scenarios," in *Third UKSim European Symposium on Computer Modeling and Simulation*, 2009, pp. 136-141.

[19] A. Al-Herz and M. Ahmed, "Model-based web components testing: a prioritization approach," in *International Conference on Software Engineering and Computer Systems*, 2011, pp. 25-40.
[20] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004, pp. 49-69.
[21] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159-182, 2002.