UNIVERSITY OF GOTHENBURG

On efficiency and effectiveness of model-based test case generation techniques by applying the HIS method: An experimental research

Bachelor of Science Thesis Software Engineering and Management

SALI EL MASRI

MAHSA ABBASIAN

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2016

On efficiency and effectiveness of model-based test case generation techniques by applying the HIS method: An experimental research

Sali El Masri

Mahsa Abbasian

Examiner: Riccardo Scandariato.

University of Gothenburg

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Göteborg, Sweden June 2016

On efficiency and effectiveness of model - based test case generation techniques by applying the HSI method:
An experimental research

Mahsa Abbasian
Dept. of Software Engineering
Gothenburg University
Gothenburg, Sweden
gusabbama@student.gu.se

Sali El Masri
Dept. of Software Engineering
Gothenburg University
Gothenburg, Sweden
guselmsa@student.gu.se

Abstract—**Researchers and practitioners have extensively studied various testing techniques and their importance on affecting the cost and the quality of software. One of these techniques is Model-Based Testing (MBT). MBT concentrates on test models that are software artifacts exploited for test automation. The goal of this project is to evaluate whether we can reduce the number test cases and time of test case execution in order to detect faults in MBT, by implementing structured test case generation methods, such as HSI method. To test our hypothesis, we conducted an experiment where we compare the efficiency and effectiveness of fault detection between the HSI method implemented by us and the random test case generation method implemented in a model-based testing tool called ModelJUnit. The experiment is done after investigating the existing random walk algorithm in ModelJUnit and implementing the HSI method in the presented tool. Our results indicate that the traditional technique which employs mutation, for some mutants, has better fault detection efficiency than the random walk, regarding the length of the test cases generated. But, concerning the effectiveness, measured by the number of mutants killed, the HSI method only showed better results than the random method for some cases. In these cases, the FSM model of the implementation consists of an increasing number of states, where the random walk cannot reach the deeply injected faults.**

Keywords—Model-based testing, finite state machines, efficiency, effectiveness, HSI method, ModelJUnit, test case generation.

I. INTRODUCTION

Software testing is an essential part of software development and an important activity to improve software quality. However, it is well known that it is costly [16]. Therefore, it should be started as early as possible to make it a part of a process for deciding requirements, and by that to improve the product quality.

In recent years, there has been a growing interest in studying model-based testing (MBT). These studies show how MBT can positively affect the budget and quality of developed software [22, 14, 15]. The main aim of model-based testing is to model the system, in order to show that the expected and actual behaviors of a system differ from each other, or to gain confidence that they do not. In other words, MBT targets a failure detection by finding observable differences between the actual behavior of the implementation and the intended behavior of the system under test (SUT) on a conceptual level, as expressed by its requirements.

1.1. Problem

Systematic testing of software systems is an important and widely demanded technique, which is used to check the quality of systems. Manual testing is usually laborious and costly and hence, automated test techniques have been considered widely in academia and industry. Therefore, the need for test automation for reduced costs and higher quality software has been recognized as a challenging opportunity to researchers to innovate, and propose new methodologies for maximizing the efficiency of the testing techniques [12]. Model-Based Testing (MBT) is such a technique, in which the test cases are generated automatically from a model of the system behavior. An example

of a formal model is Finite State Machines (FSM) [19,20], which are widely used models for describing the behavior of the System Under Test (SUT). There have been several FSM-based MBT methods proposed so far such as W-method [1], Wp-method [2], and HSI-method [3,4].

There have been some testing tools developed for application of different MBT methods and are in use today [12]. ModelJUnit is an open source Java library extending JUnit, which is designed to support MBT. This framework allows for FSM models to be written as Java classes and then tests are generated from those models and ran similar to other JUnit tests.

1.2. Aims and objectives

The primary purpose of this project is to perform a comparison between the efficiency and the effectiveness of the test case generation techniques used by one of the above-mentioned FSM -based methods, such as HSI method that we will implement, and the existing random walks algorithm in ModelJUnit for Java applications. In the first step, the possibility of applying the desired test case generation algorithm on top of the ModelJUnit library should be investigated. In the case of complications with implementing the test case generation algorithm using the library, we focus on generating test cases for applications written using our tool for a small and lightweight subset of Java. As the next step, the efficiency and the effectiveness can be compared to ModelJUnit and the algorithm implemented by us for a set of examples (e.g., by using mutation testing).

1.3. Purpose

The boost in the development of critical applications has demanded stringent methods that guarantee software reliability. Software companies have pursued solutions that have, at the same time, low cost and high effectiveness [20]. Although formal methods and model checking methods have been used to verify the software development, software testing still a widely used complement for these methods concerning executing the SUT and comparing the obtained behavior with the expected one.

The purpose of this study is to advance the current knowledge of using model- based automated tests in the testing and verification of formal specifications, which represent a significant opportunity for software testing since they precisely describe what functions the software is supposed to provide.

1.4. Scope and Limitations

This study will be limited to testing on simple Java program implemented by us and some other SUT models that are provided by the model-based testing tool ModelJUnit.

1.5. Research Questions

From the research goals we derived the following research questions:

Q1: Which of the two model-based testing methods is superior in efficiency?

Q2: Which of the two model-based testing methods is superior in effectiveness?

This paper is organized as follows. Section II presents the definition of effectiveness and efficiency, reviews the background of the various MBT techniques being compared and reviews the related work to our experiment. Section III describes the implementation of the traditional algorithm (HSI method). Section IV depicts the comparison method we adopted, the data collection and the data analysis. Section V describes the results of the conducted experiment. Section VI, discusses the results and their relation to the hypotheses. Finally, we conclude and suggest future work in Section VI.

FF. BACKGROUND

A. Overview

Efficiency has at times been defined as “doing the job right”; and effectiveness has sometimes been defined as “doing the right job” [10]. Therefore, it has been considered to be a major factor of the software development life cycle. There are many ways to measure efficiency and effectiveness of test case generation techniques in model-based testing. One way we will use is mutation testing, which is known as a test that evaluates the test generation techniques’ quality. In order to perform the comparison, we will apply the mutation testing to the random test case generation algorithm (ModelJUnit), and then on the traditional test case generation technique (HSI method).

1. Efficiency and effectiveness

Since we are conducting an experiment to compare the effectiveness and efficiency of model-based testing techniques, it will be constructive to clarify how these words are used in the context of software testing. Effectiveness means the number of faults that are detected by the utilization of an error detection technique [21]. According to Weyuker [22], measuring the effectiveness of testing is not possible since the calculation of this measurement requires information that is not available. However, the formal comparison of testing criteria regarding expected number of failures detected is possible.

Moreover, we are using the mutation analysis technique to inject faults in the original implementation program, these faults investigate and evaluate the effectiveness of the methods regarding the number of faults detected by them. On the other side, the word efficiency will be meaningful if we consider the meaning of the effort and the time that it takes to detect the faults in testing criteria. The efficiency of the test criteria also provides information about their cost-effectiveness. In our experimental study, we obtain the measurement for effectiveness and efficiency on a set of test cases generated by the two different methods, the HSI and random walk method, by using mutation testing analysis [21].

2. Finite state machine

Finite state machines have been widely used to model systems in diverse areas, like sequential circuits and most recently in communication protocols [23], [24]. Testing, with the help of FSM, increases the system reliability, where the FSM models the system to ensure its functionality.

A finite state machine (FSM) is a deterministic mealy machine [20] that produces outputs on their state transitions after receiving some inputs. It composes a 5-tuple: M = (I, 0, S, δ, λ) [25], where:

- I: is a finite set of inputs.
- O: is a finite set of outputs.
- S: is a finite set of states with initial state s_0.
- δ: δ = S x I -> S is the state transition function.
- λ: λ = S x I -> O is the output function.

When the machine is in a current state and receives an input, it moves to the next state that is specified by δ and produces an output defined by λ. An FSM can be represented as a directed graph, where the vertices perform the states, the edges carry out the state transitions and each edge is labeled by the input and output functions. For example, suppose we have a stack, an empty stack corresponds to the state s_0. When we call "push" function, we move from s_0 to s_1. The same occurs for the pop function but in the opposite way, where calling pop will take us back to the previous state.

Figure 1 shows a simple FSM of a stack with "push", "pop" and "peek" functions:

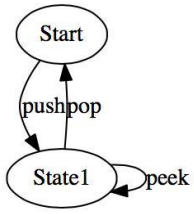

fig 1: Stack FSM.

3. ModelJUnit

Unit testing has become widespread in software quality assurance [5]. ModelJUnit tool extends JUnit to support Model-based testing for exploring FSM based tests and generating test cases. To execute the test cases and model it in a form of an FSM, it has to contain an initialization method, guards (conditions) and actions (method calls). A guard indicates the transition which is enabled and an action indicates the successor state and the corresponding action of the system under test. By using ModelJUnit library, a test can start from simple FSM model and grows to become a more complex model (EFSM), which provides a collection of traversal algorithms to generate the tests out of models. The EFSM defines the transitions and possible states which are likely to test. Also, it works as an adaptor to connect the model to the system under test (SUT) [7].

ModelJUnit offers several testing strategies, (All Round Tester, Greedy Tester, Looked Ahead Tester and Random Tester), and coverage metrics (Action Coverage, State Coverage, Transition Coverage and Transition Pair Coverage). On average, the Random tester follows the "Random walk algorithm" that covers every transition of the model. The Greedy tester follows the "Greedy Random Walk" algorithm that gives priority to unexplored paths. The looked ahead tester follows the "Lookahead Walk" algorithm that does a look ahead of several transitions to find the unexplored paths. In this experiment, we focus on the Random tester with Transition Coverage.

Figure 2 shows a simple FSM model of an SUT implemented in ModelJUnit:

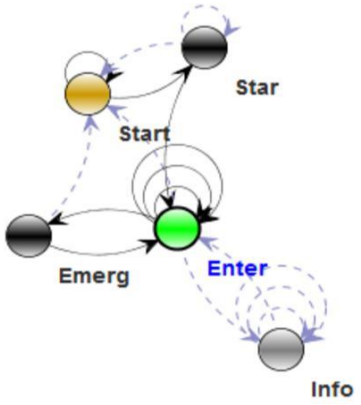

Figure2. ModelJUnit example (GUI example)

4. HSI method

The HSI-method derives a family of harmonized identifiers [8], also referred to as a separating family of sequences. The HSI-method is a traditional algorithm that contains two main parts. The first part (state identification) checks for each state of the specification whether it also exists in the implementation. The second part (transition testing) reviews all remaining transitions of the implementation by monitoring whether the output and the next state conform to the specification [9]. The HSI-method uses the separating family to assay the states in both state identification and transition testing [8]. The separating family can be obtained from a characterization set W, which in the worst case, will be the W set itself. A separating family of sequences of FSM is a collection of sets satisfying the following two conditions: For every pair of states, there is an input sequence that separates them [9]. The HSI-method uses appropriate members of the separating family in both stages of the algorithm. These members examine if the state in the implementation has the same behavior as that in the model.

The first part of the algorithm is called prefix-closed state cover set. This set reaches all the states of the FSM, and it may be created by constructing a spanning tree (see figure 3.b) from the state transition graph of the specification machine M. A spanning tree of FSM M rooted from the initial state s0 is an

acyclic sub graph of the FSM graph and is composed of all the reachable vertices (states) and some of the edges (transitions) of M. A state cover set is then generated from the spanning tree, where all the possible paths from the initial node are traversed. In the second part of the algorithm, building the family of separating sequences can be done by generating the characterizing set (W set) [17] for the FSM model with input set I and output set O. The characterizing set of M is a set W of input sequences such that for every pair of distinct states, there exists an input sequence in W such that each input of each state is different from the input of the other state.
Figure 3 shows an example of how a spanning tree can be generated from an FSM (directed graph).

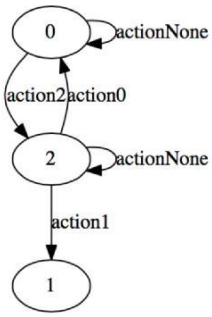

Fig2. a. FSM M

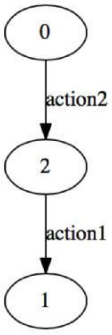

Fig3. b. Generated spanning tree of M

B. Related work

Software Defined Networking (SDN) presented a case study on model-based testing of SDN firewall programs [10]. They investigated the firewall module of Floodlight, one of the most popular SDN platforms in Java. The result showed that the generated model based tests have achieved much higher mutation coverage than the existing JUnit tests in the Floodlight firewall program, which indicated that model-based testing can be a viable option for quality assurance of SDN-based firewall programs.

K. ElFakih et al. performed some experiments on incremental test generation methods, based on HSI test derivation method, that reduces the cost of testing with on a modified system specification by generating tests that only check the corresponding modified parts of the implementation [14]. The experiments were done on an implemented software tool and it clearly showed significant gains in using incremental testing as compared to complete testing, when less than 20 percent of the transitions of the original specifications were modified.

Furthermore, A. Paradkar, conducted three different case studies in order to compare various techniques in terms of a number of generated test cases and their fault detection effectiveness. The application of the case studies was on relatively simple applications (e.g. simple ATM application for withdrawing money) [15]. The results of the case studies showed that MBTG technique which employs mutation technique for generating state verification sequences provides better fault detection effectiveness than those based on boundary values, and predicate coverage criteria for transitions.

The work we are conducting will also involve mutation testing [13], where we test the new algorithm of the HSI method. A similar work was carried out by A. Takeshi Endo and A. Simao [18] where they compared the recent methods (H, SPY, P) that generate the test suites from finite state machines with the traditional ones (HSI, W) in their study. The comparison and analysis of these methods were in different configuration and fault detection ratio and were evaluated by applying the mutation testing to simulate the faults in their experiments. The results of this study showed that the H, SPY, and P methods produce short and more test cases while the HSI and W methods product longer test cases. All methods show a high average of the fault detection ratio over 92 percentages. Though the shorter test cases are easy to debug and execute and are suitable to execute by hand. However, the longer test cases are suitable if the cost is important and a test should be executed automatically.

CCC. IMPLEMENTATION

In this section, we present the implementation of the HSI method, (depicted in Section II). HSI implements the traditional test case generation algorithm. The implementation of the algorithm is done in Java programming language and uses Java and ModelJUnit tool libraries. The implementation of HSI is done under three phases:

1. The state cover set (refer to Section II, 4): To implement this phase, we first use ModelJUnit library to build a graph (FSM) that consists of nodes (states) and edges (transitions). Then, we apply the Depth First Search (DFS) [3,4,25] to traverse the graph and build the spanning tree (fig. 3-b). Subsequently, we apply the topological sort, which gives us a power set of all the edges (transitions) in the spanning tree. Consequently, the power set of the edges is put in a set, which is the state cover set. For example, in figure 3-b a state cover set is: {ε, acion2(), action2(). action1()}, (where ε is the empty input sequence).
2. The characterizing set (W set): Due to time issues, we could not implement the complete characterizing set as in the algorithm description, but a simplified version of the W set was implemented. The simplified version of the W set, consisted of traversing the FSM to get all the distinct calls in the FSM. Assuming the FSM in figure 3-a, the characterizing set is computed as follows: {action2(), actionNone(), action0(), action1()}.
3. The concatenation of the sets: The last phase was computed by the concatenation of the state cover set and the characterizing set. The concatenation of

the two sets is as well the generation of test cases. The test cases generated by the examples mentioned above are: {action2(), actionNone(), action0(),action1(),action2().action2(), action2().actionNone(), action2().action0, etc.}.

IV. Research Method

The research process consists of an experimental research. First, we provide the description of how ModelJUnit works in test-driven development and how we can apply the HSI - method on ModelJUnit. In the next step, we implement a simple program module (*stack*) that is applied to compare the two methods by mutation testing. Then, we implement the HSI method in ModelJUnit, using its library. The last step is the comparison of effectiveness and efficiency through mutation testing analysis with the modified ModelJUnit. To mitigate the threats to validity, we applied the methods on another example (*String Set*) with an FSM model and its implementation in Java. Next, we provide a brief explanation of the two examples on which we have evaluated our approach.

Stack is a simple class with five main functions: push, pop, peek, isEmpty and clear. When the "push" function is called, an element is added to an array of integers with a maximum size. The "pop" function, when called, deletes the last added element from the stack (array) and the "peek" function shows the last element added to the stack. The Boolean function "isEmpty" checks whether the stack is empty or not and "clear" function empties the stack by reassigning a new empty array.

String Set is a simple set class with six functions: add, remove, clear, contains, isEmpty and equals. The "add" function merely adds strings to the set and "remove" function removes string from the set. "IsEmpty" returns true when the set is empty. "Contains" and "equals" are Boolean functions, which check whether the set contains the requested element, respectively compares a specified object with the set for equality.

The upcoming sub -sections are divided as follows: A- represents the design of the experimental research, B- states the data collection procedure and C - outlines the analysis of the gathered data for the experiment.

A. Experiment design

In this subsection, we present a description of the experiment in the laboratory environment. In (a) give a small definition of the experiment, (b) and (c) cover successively the independent and dependent variables. Additionally, (e) presents the experiment steps, (f) covers the measurements instruments we used in the experiment and finally, (g) describes the measurements' objects.

a) Experiment definition

Although several studies have been conducted in order to compare and evaluate different testing methods and strategies, few of them have been validated experimentally [26]. Theoretically, the structured test case generation algorithms, such as HSI, are expected to be more effective concerning killing more mutants than the random algorithm. The efficiency of the two algorithms is as well expected to produce considerable differences with respect to the number of test cases generated before finding the mutants and killing it. The ModelJUnit is predicted to be ameliorated after applying the HSI. This amelioration is anticipated due to the strategy that the random walk follows when it randomly resets after generating some test cases, which may require longer time and number of test cases before detecting the fault. In order to design the experiment, we needed to understand how the test case generation effectiveness and efficiency could be measured.

b) Independent variables

The independent variables are the variables to manipulate in the experiment [11]. In this experiment, the independent variables will be the ModelJUnit algorithm and the HSI-method.

c) Dependent variables

The dependent variables are the outcome of the experiment that we want to study to see the effect of changes [11]. The dependent variables will be the measured efficiency and effectiveness of the tool after the modification.

d) Hypothesis

The general hypothesis of the experiment is that the ModelJUnit with HSI method is more efficient and effective in generating test cases and increasing the coverage metric, i.e. the amended ModelJUnit is assumed to generate more test cases per time unit, and to find a larger rate test coverage. The hypotheses of our experiment are presented below:

$H_{0\text{-}Effcy}$: There is no difference in efficiency measured by the length of the test cases generated before and after the application of the HSI-method to ModelJUnit.

$H_{1\text{-}Effcy}$: There is a difference in efficiency measured by the length of the test cases generated before and after the application of the HSI-method to ModelJUnit.

$H_{0\text{-}Effv}$: There is no difference in effectiveness measured by the number and type of mutants killed before and after the application of the HSI-method to ModelJUnit.

$H_{1\text{-}Effv}$: There is a difference in effectiveness measured by the number and type of mutants killed before and after the application of the HSI-method to ModelJUnit.

e) Experimental steps

The experiment plan was divided into a two-phases process. In the first phase, we implement the HSI algorithm (as described in Section III) in ModelJUnit by referring to its library. To check the complexity of applying it in ModelJUnit, we applied some tests on the simple stack class, which was flexible to convert to an FSM model with a larger number of states, where each time "push" is called, a new state is added. In the second phase, we applied the mutation approach (read more in section C) to the implementation part of the SUT. Afterward, we extended the tests to apply it to programs with larger FSM models provided by ModelJUnit, in order to obtain more comparative insight about the examples.

f) Measurement Instruments (mutation testing)

Measuring efficiency and effectiveness is done by mutation testing analysis. Mutation testing provides an indication of the fault detection ability of a test set [13]. Mutation testing analysis introduces small syntactic changes in the source code of a program to produce mutants (for instance replacing a plus sign with a minus sign). The aim of mutation testing is the evaluation of a test set, and to do that, all the mutants should be killed. In order to kill the mutants, the test set should reveal a difference between the original and the mutated program [13]. There are several tools introduced for automatically create mutants and evaluate the test sets (e.g. Pitest, MuJava), but for ModelJUnit we created them manually since the mentioned tools are largely used for JUnit testing evaluation and do not support MBT tools.

g) Measurement objects

The efficiency is measured by the number of test cases generated until it detects the fault. The effectiveness is measured when the faulty implementation (the mutant) is killed during a test execution. We analyzed the effectiveness of our mutation operators (defined in C) on their ability to kill faulty implementations.

B. Data Collection

The information for describing the experiment's instruments will be gathered from literature review through research papers related to ModelJUnit, random walks, HSI-method, mutation testing. Furthermore, the supervisor provides the technical specifications of the new algorithm that we need to implement in ModelJUnit for the experiment.

For the experiment, the type of data to collect will first be testing the ModelJUnit on a small program implemented by us in java (e.g. Stack, Queue). ModelJUnit also has some examples of various sizes with both FSM models and their implementations applied in these examples as well.

In the next step, we test the same program with the HSI-method. Then, we check the both techniques using mutation testing and compare the results of efficiency by measuring the number steps required to kill a certain mutant.

The effectiveness will be measured by the number of mutants killed (within a given time bound). For each test program, the number and type of mutants generated and killed by the test cases generated are recorded and analyzed. The data obtained for the tested programs were separately analyzed to determine the effectiveness and efficiency of the two distinct methods. The results are compared and analyzed in the discussion phase.

C. Data Analysis

By following the mutant generation approach, we obtained a set of mutants when injecting faults in the original program using a set of mutant operators (refer to Section A-f). We manually created mutants in the *Stack* program, implemented by us and the existing *String Set* program in ModelJUnit. After the mutants have been created, we ran the ModelJunit random tester and then the HSI traditional tester to compare the number of test cases generated by each of the two methods before killing the mutant and the number of mutants killed.

Furthermore, according to mutation testing structure, we tried to change the implementation part of the SUT and inject manual errors in the code through some changes. Because of the nature and small size of the programs, the only applicable mutation operators that we used, and the types of faults injected by them were the following [21]:

- Literal change operator (LCO): the type of fault injected was changing increment to decrement and vice versa, changing and removing statements.
- Control flow disruption (CFD), (or value mutation): The type of fault injected was incorrectly placing block markers (curly brackets) and changing return values.
- Statement swap operator (SSO), (or statement mutation): the type of fault injected was swapping the order of statements in the same scope or removing it.
- Variable replacement operator (VRO): The type of fault injected was replacing a variable with another from the same type.
- Missing condition operator (MCO), (or decision mutation): the type of fault injected was removing a condition in a conditional statement.
- Relational replacement operator (LRO): the type of fault injected was replacing true with false, and replacing greater than, equal to or less than and vice versa.

We could not produce an exhaustive list of mutants for the examples. Based on the code size, we determined that 25 mutants would be adequate for the *Stack* program application, and 20 for the *String Set* program application. The *String set* program had fewer mutants than *Stack* program because it does not have too many arithmetic operations and literal constants since it uses functions from the set that is built in Java library.

For the *stack* example, to provide more focused and elaborated evidence, we extended the mutation based approach to include deep faults in the implementation, which target the FSM states of the SUT model. An example implicates, inserting a faulty conditional statement that pushes a wrong element to the stack after certain steps (we increased the number of states of the SUT from 10 to 199, and we pushed a wrong number at state 99). We did not consider the bugs that would lead to crash when running the program. Table 1 shows the distribution of the different types of mutation operators and faults injected by them.

Table 1
The mutation operators' distribution on Stack and String Set

Mutation operator	Nr of mutants on Stack	Nr of mutants on String Set
LCO	4	2
CFD	3	3
SSO	2	1
VRO	3	4
MCO	4	3
LRO	9	7
Deep Faults	2	0

V. RESULTS

A. Stack Program

We applied the generated test cases from the two methods on the original and on each of the 25 faulty versions of stack program. Subsequently, we measured the measurement objects (Section A-g). The results are shown in Table 2. As can be seen, most of the faults injected into the *stack* implementation were killed at the same time, by the two methods. These mutants, were mostly of types LCO, where all the increments of the array's indexes in "push" method, were altered to decrements instead and vice versa (for example, stackArray[top++] statement is converted to stackArray[top--] in push). Another example was the LRO mutation operator, this operator was distinctly revealed by the two methods we well (for example, in all the existing conditional statements in the implementation of *stack*, we replaced the cases of = = to <= /! =/>= or </>).

On the other hand, the table 2 also shows that some other mutants were not revealed forthwith by the two methods (late killed mutants). These types of mutants (mainly VRO) required more testing effort from both the random walk method and the HSI method (i.e. needed to generate more test cases). For instance, the delayed coverage of some statements was in the part of the code which assigns the last element added by "push" to the stackArray, this statement was modified to assign another value instead. Furthermore, one noticeable issue was that the ModelJUnit, when generating the random walk tester, could only kill the late detected mutants after a certain amount of tests, depending on the length of the FSM states (since the stack FSM is an increasing number of states, when the length of states was 10, the number of test cases had to be raised to generate about 50 tests before failing). When the length of tests generated is below the specified number, the mutants were not covered. However, since the HSI method follows a more structured strategy than the random one, the amount of test cases were not changeable. But it also had to execute a similar number of test cases the ModelJUnit ones before failing with the same type of mutants. Additionally, six defects went undiscovered by any of the methods. The defects were of different types (2 of type SSO, 1 of type VRO, 1 CFD and 2 of MCO) . As shown, the survived mutants were of various types, this was surprising since the tests from both methods were covering all the code. After we checked these mutants, we discovered that they were largely not of a big matter to failing. An example is, when we swap a certain order of statements, it does not need to be an error.

A useful observation of this experiment is that the deep faults that were later appended to the implementation (see the example in Section III-C) were not revealed by the random method. We injected these errors to see if the random walk will still kill the mutant after generating some test cases. But, the results out of this test showed that the random walk did not detect the mutant even when we increased the length of test cases to 5000. However, the HSI method discovered the mutant after generating a pretty long sequence of test cases in order to reach that far state.

Table 2
Results for Stack program

The methods (Stack)	Mutants total	Mutants early killed	Mutants late killed	Mutants survived	Mutation Coverage (%)
ModelJUnit method(random algorithm)	25	14	4	7	72%
HSI method (traditional algorithm)	25	16	5	4	84%

B. String Set Program

We followed the same process for the *String Set* example after injecting the 20 faulty versions in the original program module. The results are shown in Table 3. As we can see, similar results were obtained for the *String set* model, which killed most of the injected faults, that are from different types of mutant operators, by both the random method and the traditional method. Various mutants were applied on this example, where mostly the mutants that were instantly killed were of type VRO, MCO and LCO. These faults were mainly removing statements, swapping others and changing values from the same type, especially when we check if a string exists in the set. As other examples were when we change the true to false and vice versa.

No significant differences, concerning effectiveness and efficiency, were shown between the random and the traditional methods when they run some test cases before realizing the mutants and killing them. However, the random method needed to generate some more test cases than the traditional method before failing (for the String set, it needed to produce 27 test cases before the fail, while the traditional test case generation method fails after 10 test cases). Further, we investigated the survived mutants, and we observed that these mutants involved some implementation variables which were not present in the specification so that the tests could not cover it.

Table 3
Results for Stack program

The methods **(String Set)**	Mutants total	Mutants early killed	Mutants late killed	Mutants survived	Mutation Coverage (%)
ModelJUnit method(random algorithm)	20	10	5	5	75%
HSI method (traditional algorithm)	20	12	4	4	80%

VI. DISCUSSION

A. *Efficiency and effectiveness*

Tis experiment tries to provide a more effective method, regarding MBT. As mentioned before in the report, the efficiency of a test case is its ability to find the fault with the least effort, and the effectiveness is the ability of this test case to find the fault in a program. The results have shown that there is no significant difference in efficiency between the random walk in ModelJUnit and the structured HSI method, that was measured by the number of the test cases generated before detecting the mutants. Hence, we could not reject the hypothesis

$H_{0\ effcy}$, which claims the no difference in efficiency between the the random walk method and the HSI method.

Furthermore, the effectiveness of the methods was dependent on the type of the implementation. For example, when we ran the experiment on the *String set* model, which consisted of limited states of true and false options, each time we add or remove a number, the two methods were either detecting the mutants and killing them at the same time, or at all not recognizing that the implementation was faulty. Whilst in the stack model, the states of the FSM are not options, but numbers that increase each time we push an element in the stack. In this case, when we increased the number of states to 100 and injected some severe faults in the original implementation of the stack (e.g. in state 99 when we pushed a wrong element), the random walk could never reach the state 99 because of the random reset that the random walk method calls with a chance of 5% while generating the test cases. But, the HSI could recognize the fault and fail when reaching the state 99.

However, to reach the state 99 in ModelJUnit, we had to decrease the chance of calling the reset function to 0.001 %. This reduction helped us to detect the deep mutant and kill it but rose another issue. Namely, in models and implementations with numerous branches this could reduce the effectiveness, because the random walk may be stuck in a particular branch without resetting to cover other branches. Thus, we could conditionally reject the hypothesis H_{0effv}, which claims the difference in effectiveness between the random method and the HSI method. According to this condition, the random walk could compete in effectiveness with HSI only if we know exactly how the model looks like.

In fact, we did not put a major attention to investigate the reasons behind all the killed or survived mutants, as our experiment is to examine whether there is an improvement, with respect to efficiency and effectiveness, after applying the HSI method to ModelJUnit not how effective or efficient the methods are.

B. *Threats to validity*

No data is perfect and no analysis can be 100 percent trustable. Specially, in any experimental study, it's crucial to identify the threats to validity and carefully assess the likelihood of such threats and their potential consequences [27]. Some factors that affect the validity of the results of this experiment can be classified into two primary types: internal threats and external threats. The internal threats determine whether the conditions of the experiment and the evidence offered support what the experiment claims to provide. Further, the experiment time was one of the major threats to validity, the entire duration of the experiment was about three months, which was quite short for such a study. Due to time constraint, we have not gotten the opportunity to complete the application of the full HSI method in ModelJUnit, so an implementation of a simplified W-set of HSI would not be sufficient to validate the experimental results. Moreover, the examples we were running the experiment on, were of a small size, but this experiment may require sufficiently large test examples in order to covers as many mutants as possible for the results to be more valid and interpreted correctly. On the other hand, our basic understanding of the model-based testing, since we are bachelor students, can be as well treated as a threat to the validity of the results.

The external validity is related to our ability to generalize the results of the experiment [27]. Since we did not have access to automated tool support for the methods we are comparing, we chose only two modest examples (stack and string set). Thus, for a broader applicability of the results, it will be important to repeat these experiments on other examples where real faults have been introduced and where the complete HSI method has been implemented. Indeed, we do not believe the results will be impacted by the simplicity of the chosen programs, but only on the size.

C. *Limitations of the experiment*

Lack of proper tool-support for obtaining better results may be a hindrance in applying this experiment. For example, the mutation operators of different types were injected manually,

since there are no tools for automatically inserting faults in the implementation part of the models. On the other hand, time restricts and lack of experience of the researchers were as well limitations to provide better conclusion of the conducted work.

VII. Conclusion

Testers are usually interested in the fault-finding abilities of different software techniques and models (effectiveness) and their effort requirements (efficiency). In this paper, we review two important model-based test generation methods: the random walk method modeled by ModelJUnit and the HSI structured method. Moreover, we report the results of an experimental study, that compares the two reviewed methods, regarding their effectiveness and efficiency. We found that the two methods required a nearly equal effort to detect bugs in the implementation, while the HSI achieved more effective fault detection than the ModelJUnit, only when the chance for random reset was about 5% or more. But if we do not know what the right random reset percentage is, we are more likely to either miss a lot of bugs or not cover the all the states. Thus, we conclude that the random walk could be a competitor with HSI only when we exactly know how our model consists on an increasing number of states. However, it is not clear if these results can be generalized to wider examples of larger complexity, so we believe that more experiments are needed in the future for more accurate results and better validity.

Acknowledgment

We would like to express our sincere gratitude to our supervisor, Mr. Mohammad Mousavi for the continuous support and for his patience, motivation and immense knowledge, and for providing us with the necessary knowledge throughout this thesis period in order to conduct our experiment properly. We would like to thank him too for his constant feedback during the entire phase of the research's development. We would also like to thank Mr. Imed Hamouda for his feedback which helped us in constructing the thesis proposal and final report. Special thanks to Miss. Mahsa Varshosaz for providing us with all the necessary facilities for the research, in particular to the implementation of the HSI method. Last we would like to thank all the volunteers who have helped in achieving the goals of this research.

References

[1] T. S. Chow. Testing software design modeled by finite-state machines. IEEETrans. Softw. Eng., 4(3):178–187, 1978.J.

[2] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite State Models. IEEE Trans. Softw. Eng. 17(6), 591-603, 1991.

[3] K. Sabnani and A. Dahbura. A protocol test generation procedure. Comput. Netw. ISDN Syst., 15(4):285{297, 1988.

[4] N. Yevtushenko & A. Petrenko (1990): Synthesis of test experiments in some classes of automata . Automatic Control and Computer Sciences 24(4), pp. 50–55.R.

[5] J. Link and P. Fröhlich. Unit Testing in Java: How Tests Drive the Code. Morgan Kaufmann Publishers, Inc., 2003.

[6] C. Artho. Separation of Transitions, Actions, and Exceptions in Model-based Testing,12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009.

[7] M. Utting and B. Legeard. Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers, 2007.

[8] A. Petrenko, N. Yevtushenko, A. Lebedev, A. Das. Nondeterministic state machines in protocol conformance testing. In: Proceedings of the IFIP TC6/WG6.1 Sixth InternationalWorkshop on Protocol Test systems, vol. VI, pp. 363–378 (1994).

[9] G. A. Nemeth, Z. Pap, G. Kovacs, M. Subramaniam, FSM-based Incremental Algorithms for Test Generation and Testing, Computer Science Department, University of Nebraska at Omaha, Omaha, NE 68182, USA

[10] I. Alsmadi, M. Munakami, D. Xu, Model-Based Testing of SDN Firewalls: A Case Study, 978-1-4673- 9581-6/15, 2015, IEEE.

[11] C. Wohlin, and P. Runeson, Experimentation in Software Engineering: An Introduction. 2000, Boston: Kluwer Academic Publishers.

[12] M. Ramachandran, And R. Carvalho, (2010). Handbook of research on software engineering and productivity technologies. Herhey, PA:Engineering Science Reference.

[13] L. du Bousquet and M. Delaunay, (2008). Towards Mutation Analysis for Lustre Programs. Electronic Notes in Theoretical Computer Science, 203(4), pp.35-48.

[14] K. EI-Fakih, N. Yevtushenko and G. Bochmann, (2004). FSM-based incremental conformance testing methods. IEEE Transactions on Software Engineering, 30(7), pp.425-436.

[15] A. Paradkar, (2006). A quest for appropriate software fault models: Case studies on fault detection effectiveness of model-based test generation techniques. Information and Software Technology, 48(10), pp.949-959.

[16] P.K. Kapur; G. Singh; N. Sachdeva And A. Tickoo, Measuring software testing efficiency using two-way assessment technique, Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 2014 3rd International Conference on, Oct. 2014, IEEE

[17] M. Broy (2005). Model-based testing of reactive systems. Berlin: Springer. 87-111.

[18] A. T. Endo And A. Simao (2013). Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods. Information and Software Technology, 55(6), 1045-1062. doi:10.1016/j.infsof.2013.01.001

[19] A. Takeshi Endo, A. Simao, Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods, June 2013

[20] A. Takeshi Endo and A. Simao. Experimental Comparison of Test Case Generation Methods for Finite State Machines, Fifth International Conference on Software Testing, Verification and Validation, 2012, IEEE.

[21] A.Gupta & P.Jalote, An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing, Published online: 8 January 2008.

[22] E. Weyuker: Can we measure software testing effectiveness?In: Metrics '93: Proceedings of 1st International Symposium on Software Metrics,IEEE Computer Society, Washington (1993).

[23] A. D. Friedman and P. R. Menon, Fault Detection in Digital Englewood Cliffs, NJ: Prentice-Hall, 1971.

[24] Z. K'ohavi, Switching and Finite Automata Theory, 2nd ed. New York McGraw-Hill, 197:8.

[25] D. Lee and M. Yannakakis, Principles and methods of testing finite state machines : A survey, Poceedings of the IEEE, Vol. 84, No. 8, August 1996.

[26] N. Juristo, A.M. Moreno, S. Vegas: Reviewing 25 years of testing technique experiments. Empirical Softw. Engg. 9(1–2), 7–44 (2004).
[27] J. H. Andrews and Y. Labiche: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Transactions on software engineering, Vol. 32, NO. 8, August 2006.