FSM Testing Based on Transition Trees and Complete Round Trip Paths Testing Criteria.

by

Hoda Ahmed Khalil

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Doctorate of Philosophy

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

State machine testing has received genuine attention in the literature. Among the studied testing strategies are complete round-trip paths and transition trees that cover round-trip paths in a piecewise manner. Although trees are claimed to be equivalent to complete round-trip paths testing, there is anecdotal evidence that this is not the case. The core contribution of this thesis is therefore an empirical comparison between the effectiveness of the complete round-trip paths test suites and the transition trees test suites on the one hand and the effectiveness of the different techniques used to generate transition trees (breadth first traversal, depth first traversal, and random traversal) on the other hand. The comparison is conducted using four experimental objects and a significant number of test suites. Other contributions include constructing a reusable experimental setup and developing our automatic toolchain including novel algorithms to generate the different types of test suites. We also evaluate the effect of the structure of the transition trees as well as the state machine on the fault detection capability of the resulting test suites.

Our results demonstrate that covering round-trip paths in a complete manner is not equivalent to covering them in pieces. In addition, using statistical analysis, we prove that the different types of transition trees and even every single tree has different fault detection capability. Then, we show that neither the average length of the test cases in a test suite, nor the standard deviation of the test cases length, nor the number of complete round-trip paths included in a transition tree affects the effectiveness of the test suite. We also show that the effectiveness of the complete round-trip test suites is proportional to the connectivity of the finite state machine of the system under test.

Acknowledgments

Success is the joy of the journey, not merely the destination. Now that my Ph.D. journey is coming to a successful end, evaluating this journey in retrospective, I admit I enjoyed every bit of it thanks to the support and encouragement of many great people that I was lucky to have in my life and who always pushed me to pursue my dreams.

I would like to express my sincere gratitude to my supervisor, Dr. Yvan Labiche for his understanding during my down times, for his appreciation and encouragement during my up times, for all kinds of support he generously provided and for the thorough reviews and intellectual conversations that I enjoyed throughout my thesis work.

I would also like to thank all my examiners. The insightful suggestions of Dr. Samuel Ajila and Dr. Stéphane Somé were of great help during my thesis proposal. The detailed review and interesting debate with Dr. Douglas Howe during my thesis defence added a valuable dimension to my work. I appreciate the effort of my committee chair; Dr. Alan Cai to make the defence process smooth and easy. I would also like to thank Dr. Alexandre Petrenko for the very valuable and to the point reflections on my thesis manuscript.

Last but definitely not least, I would like to thank my family. I can never thank my parents enough for all what they did to make me who I am, for their unconditional love and for the values they instilled in me that helped, and continue to help me, through the different challenges of life. I thank my husband for his love and support, for boosting my confidence during the hard times, and for encouraging me to pursue this work. Finally, I thank my

three children for always making me proud, and for being my role models in ambition, persistence, and ethics.

- Hoda

Table of Contents

List of Tables

List of Illustrations

Introduction

Finite state machines (FSMs), being intuitively understandable and suitable for modeling in many domains, are used by many software designers and are particularly useful for modeling and testing various pieces of software [4, 5, 15, 30, 31].

There are several FSM testing criteria. Among these criteria is the transition tree method. Briand and colleagues show that the transition tree method is a good compromise between the cheap but ineffective all-transitions criterion and effective but expensive all-transition-pairs criterion [10].

Binder implicitly claims that transition trees are equivalent to complete round-trip paths testing [8]. However, there is anecdotal evidence that trees are not equal for instance in terms of effectiveness at finding faults [47].

The cost effectiveness balance of the transition tree criterion provides a strong motivation to evaluate this criterion empirically. The aim of my work is to evaluate the different approaches for developing adequate test suites for the transition trees/round-trip paths (RTP) to find ways to maximize the effectiveness of the studied criterion.

Chow [16] introduced the W-method that generates test suites using a transition tree traversal of the FSM representing the system under test (SUT). The collection of paths composing the tree is then used as a collection of test cases. Binder later introduced the concept of round-trip paths [8], paths that are prime paths of non-zero length that start and end at the same node [1]. A path from node n_i to node n_j is a prime path if it is a simple path and it does not appear as a proper sub path of any other simple path [1]. A simple path does not contain any repeating nodes, except possibly for the first and last nodes of the path

[8]. However, Binder's method covers the round-trip paths in pieces by adapting the transition tree solution of Chow [16]. The works of Chow and Binder among other research literature is discussed in Chapter I and Chapter II. Binder's way of covering round-trip paths in his N+ method raises the first of my four research questions (Chapter III): Is covering complete round-trip paths equivalent to covering them in pieces using transition trees? Binder argues that either a breadth first search (BFS) or depth first search (DFS) can be used to create a tree and generate round-trip paths. This triggers the second research question: Are BFS and DFS trees different from a testing point of view. Last, since there are typically several BFS/DFS trees for an FSM we ask: Are all BFS (resp. DFS) trees equivalent from a testing point of view?

To answer those questions, we use four experimental objects (Chapter IV), generate all possible BFS and DFS trees, following Binder/Chow's algorithms (we generate a total of 102,113 test suites for all experimental objects combined), as well as random test suites and test suites that exercise round-trip paths in their entirety. To facilitate experiments, we deployed an automation toolchain [43], which is described together with the general experimentation setup (Chapter V and Chapter VI). We compare test suites by measuring their effectiveness at finding faults. Results in Chapter VII lead us to develop four hypotheses, which we validate in Chapter IX [44]. We then discuss threats to validity, and finally, we state the conclusion.

The main contributions of this dissertation include building a robust chain of tools that can be reused for similar experimental research to automatically generate and run BFS, DFS, random and RTP test suites [43]. This includes novel algorithms to generate the test suites as well as documented procedures for replicating the experiments [45]. The thesis

also contributes to the testing literature by performing the experiments using a huge number of test suites, collecting the significant amount of resulting data, performing statistical analysis using descriptive and inferential methods (ANOVA, t-test, Kruskal-Wallis, and U-test), and drawing conclusions based on that. Our work proves that complete RTPs detect faults that cannot be detected using piece-wise RTPs. The empirical work provided by the thesis also proves that complete RTP test suites outperform other test suites, especially for more connected FSMs. We also prove that the method used to generate the transition trees (BFS, DFS, or random) can affect the number of detected faults, and that test suites generated using any one of these testing methods are not equal in terms of effectiveness at detecting faults [44]. This leads us to investigate the structural characteristics of test suites that could be used to explain effectiveness results and can therefore be used to select a test suite among many. The thesis evaluates the relationship between the mean paths lengths of the test cases for transition trees test suites and their effectiveness. The results lead us to test another hypothesis to find a possible relationship between the extent to which a tree is balanced and the effectiveness at finding faults. The thesis also studies the correlation between the structure of the FSM and the nature of the SUT on the one hand and the detection of mutants belonging to different mutation operators on the other hand. The different contributions of the thesis, as well as the future research opportunities, are summarized in Chapter XI.

Chapter I Background

This chapter discusses three main concepts that are essential to comprehend before proceeding to the research discussed in this manuscript. FSM is explained, then different criteria used for testing an FSM based system are discussed, and finally a brief background of mutation analysis is provided.

I.1 Finite State Machines

The term finite automata were first introduced as a mathematical concept to model hardware systems such as sequential circuits [63]. FSMs are the engineering application of finite automata[8]. A state machine is one of the most powerful ways to represent a control system where a number of stimuli (inputs) is received from the application and actions (outputs) are produced to affect the application [83]. State machines are used to describe behaviors of sequential systems where outputs depend on inputs and the current state. This is to be opposed to combinational sequences where the output is only dependent on the set of inputs. There are four building blocks in an FSM. Those are states, transitions, events, and actions [8].

The first building block of an FSM is a state or a collection of states that represent all the possible situations in which the FSM maybe in. The system goes through a sequence of events to reach a certain situation or state. Therefore, a state is some kind of a memory that represents the history of the model [83]. From a software point of view, a state is a set of specific values for a collection of variables [1].

The second building block is the group of transitions in an FSM. A transition is an allowable two-state sequence that may result in an output action and must specify an

accepting state and a resultant state [8]. A transition occurs in zero time, and usually means a change in the value for one or more variables [1].

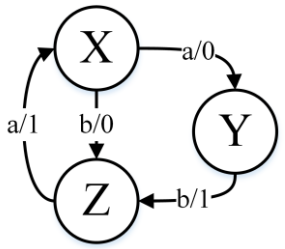
The third block is the group of events. An event could be either an input or an interval of time. In the latter case, if the interval of time passes, the transition takes place [8].

Finally, the fourth building block is the set of actions that may be associated with the different transitions. Actions are the result or the output produced when a certain event is triggered [8].

There is more than one way to represent an FSM. One way is the mathematical modeling. There is more than one way to model an FSM mathematically depending on the application. For example, a transducer FSM is represented as a sextuple ($\sum$, Γ, S, s0, δ, ω), where $\sum$ is a finite non-empty set of symbols that composes the input alphabet for the FSM, Γ is the finite set of output alphabet, S is the set of states, s0 is the initial state of the system ($s0 \in S$), δ is the function for transitions ($\delta: S \times \sum \rightarrow S$), and ω is the output function ($\omega: S \times \sum \rightarrow \Gamma$). For a parser FSM, a quintuple ($\sum$, S, s0, δ, F) is a more suitable representation since on the one hand, there is no need for an output function and on the other hand, a symbol F is added as the set of final states [83].

An FSM can also be represented by a transition matrix, where rows represent the source state and columns represent the destination states. Table cells store the transition number or name where there is a transition starting at the row state and ending at the column state [83]. There is no agreed convention on showing actions in a transition matrix. For compatibility, in Figure 1, the input/output (event/action) pair is stored in the cells of the transition matrix.

$\sum = \{a, b\}$
$\Gamma = \{0,1\}$
$S = \{X, Y, Z\}$
$s0 = X$
$\delta (X, a) = Y, \delta (X, b) = Z, \delta (Y, b) = Z,$
$\delta (Z, a) = X$
$\omega (X, a) = 0, \omega (X, b) = 0,$
$\omega (Y, b) = 1, \omega (Z, a) = 1$



From\To	X	Y	Z
X	-	a/0	b/0
Y	-	-	b/1
Z	a/1	-	-

Figure 1 Different representations of an FSM

While a transition matrix is basically a table and therefore easier to draw and to fit into a smaller area, the third way of modeling an FSM which is a transition diagram is usually the most comprehensible state machine representation [83]. In a transition diagram, states are represented by nodes or simply circles, while transitions are mapped into arrowed arcs. The tail of the arc is at the source state, while the head is at the destinations state. Figure 1 shows the explained mathematical, graphical and transitions matrix representations of the same state machine.

In real life applications, the simple mathematical representation of state machines may become too complex. The transitions are triggered by events rather than simple input symbols and states might have complex actions performed due to the triggered transition and not just one output.

There are different types of actions (outputs) associated with an FSM as a result of the triggering of a certain event (input) : entry action (done when the state machine enters the state), exit action (done before the state machine leaves the state), input action (done when input condition is true), and transitions action (an action performed during the change of states) [83].

There are two main types of state machines: Moore machines and Mealy machines. In a Moore machine, the output depends only on the current state of the system, while the

outputs of a Mealy machine are determined by both the current state and the input [88]. There is no general rule on which model is better and it depends mainly on the application, but the selected model affects the design. Hardware systems, for instance, are best represented by a Moore model. In this manuscript, the focus is on Mealy machines since it is the one mostly used to specify software systems [81]. In a Mealy machine, each transition in the FSM is labeled with a single event/action (input/output) pair. When used for testing purposes, each transition in the Mealy machine is a single step in a test case where the input is the stimulus sent to the system under test (SUT) and the output is the expected response by the SUT [81].

The idea of the final state as in a parser state machine is very limiting since a state machine does not need to stop, and it can work continuously [83]. Therefore, the FSM (Mealy machine) used in this research does not have a final state. The used FSM is also composed of a finite set of states and produces outputs on state transitions after receiving inputs as explained in the preceding paragraph when discussing input actions of a Mealy machine [21, 38].

I.2 State-based Testing

FSMs are particularly useful for testing protocols, telecommunication, concurrent systems, system failure and recovery, system configuration, and distributed databases [3, 5, 61]. In addition to all the application domains where state machines are commonly used, control tasks that are ideally modeled using FSMs are present in all software [83]. Because of that, and the fact that state machines are intuitively understandable and therefore adopted

by many designers, state machine models can be used in most software nowadays. FSMs are widely used as a functional testing tool as well as a design programming tool [5].

That being said, FSM testing has been widely covered in the testing literature. Strategies for state-based verification target the use of state machines for software structure, software behavior and software specifications [5].

FSM based testing belongs to a wider category of software testing called Model-based testing. Model-based testing is a black box testing technique based on a formal model that is usually built to facilitate automation [49, 55]. Model-based testing has proven to have high fault detection effectiveness while maintaining reduced costs [12].

An FSM is basically a graph where the nodes are states and the transitions are edges connecting the nodes of the graph. Therefore, one can claim that all graph testing strategies can be applied to state-based testing. However, data flow criteria, which are testing criteria based on the flow of data through the SUT [1], is more troublesome when applied to state-based testing. Most representations of state machines nodes are not allowed to have variable definitions (defs) or variable uses for example since all actions are supposed to happen on the transition. At the same time, the semantics of the triggering of events imply that the effect happens immediately to the variable by taking the transitions, which means that uses are reached directly after defs. For all this, there have been few attempts (e.g. [7, 60]) at applying data flow criteria such as all defs and all uses to FSM. To the contrary, control flow based testing strategies can be easily mapped to state-based testing strategies [1].

There are different types of faults that can occur in a control system based on a state machine. The faults types are:

1. Missing or wrong transition that results in an incorrect target state,
2. Missing or incorrect event that results in a valid event being ignored,
3. A missing or incorrect action that results in the wrong behavior,
4. An extra, missing or corrupt state,
5. An illegal event that gets accepted when it should not, and
6. The implementation accepts undefined event.

Different testing strategies aim at detecting the above-mentioned classes of faults. Binder [8] classifies major state-based testing criteria as follows:

1. Piecewise testing strategies: A piecewise strategy test suite exercises each state, each event, or each action at least once. Such a strategy is not dependent on the control flow of the system. So, if it finds faults, this is basically accidentally. This method is easy to satisfy. It is possible for instance to exercise all states while missing some actions. Therefore, a piecewise strategy is not an effective way to find faults.
2. All transitions testing: In this strategy, every transition is exercised at least once. This, of course, exercises all states, all events, and all actions. This method cannot show that an incorrect state has resulted. It cannot reveal the presence of extra transitions.
3. All transition k-tuples: All n-transition coverage is satisfied when all sequences of n-transitions are exercised. This can reveal some corrupt states but not necessarily all of them. This is equivalent to n-switch coverage described by Chow where a 0-switch refers to transition coverage, and 1-switch refers to covering all sequences of two transitions in the FSM [16, 81].

4. All round-trip path: This criterion is satisfied when all paths that represent loops are exercised. The round-trip path test strategy will be discussed in detail in the foundational work section of Chapter 2.
5. M-length signature: This strategy assumes that the SUT is opaque, i.e., that there is no means to determine the current state of the system at any point in time. Therefore, the only way to make sure that the system is working correctly is through observing the outputs produced by the SUT and comparing them with expected ones (from the FSM). So, to conclude that the SUT is in a certain state, one applies the signature or the distinguishing sequence of inputs that produces different

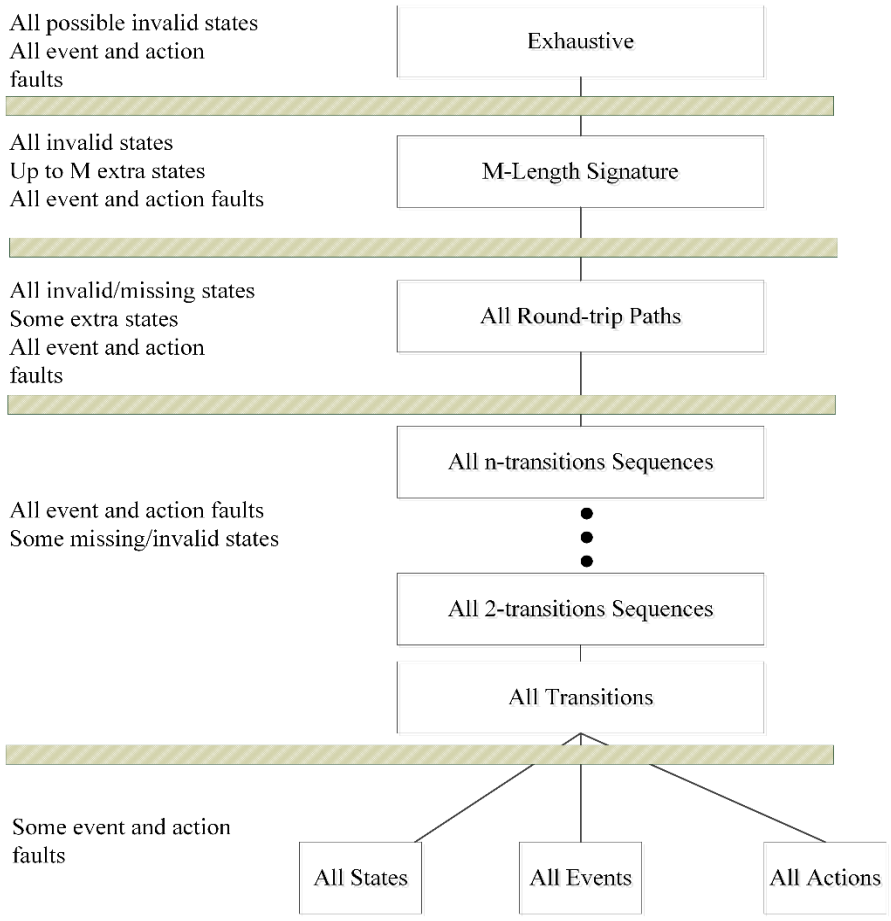


Figure 2 State-based Testing Techniques Power Hierarchy and Fault Detection

outputs for each state. If the result is as expected, then the system is in the correct state. Otherwise, it is not [8]. The most classic method based on this criteria is the W method [16, 82]. Other methods that are more or less based on the W method include Wp [65], UIO [72], HSI [68], H [20], SPY [75], and P [74].

6. Exhaustive Testing: This is the most expensive strategy since it attempts to exercise all possible scenarios that the software can go through as specified in the corresponding FSM.
7. A seventh criterion that could be added to Binder's classification is the random walks (e.g., test sequences with fixed length that are randomly generated) that are surprisingly effective at finding faults and usually the least expensive to generate [11].

Figure 2 is adapted from [8] with minor clarifying modifications, to illustrate what the author calls the power hierarchy of the different testing strategies and the kind of faults they succeed to detect. The power hierarchy maps to what is defined as the Subsumption relationship in [1]. Note that the criterion referred to as all round-trip paths in Figure 2 accounts for simple paths as well as round-trip paths of the FSM. A testing criterion C1 subsumes criterion C2 if and only if any test suite that satisfies C2, also satisfies C1 [1].

I.3 Mutation Analysis

Another concept that is used in this research as an aide to evaluate and compare different test suites, is fault-based testing. Fault-based testing is defined as a software testing criterion that uses test data designed to demonstrate the absence of a set of pre-specified faults. Mutation analysis is one of three types of fault-based testing [45, 61]. The

other two types are error guessing and fault seeding. Error guessing depends on the test engineer experience where one can guess where and what type of faults might exist. Then, the test engineer designs tests specifically targeting those faults and the test stops when all the faults guessed by the testing engineer have been discovered [38]. In fault seeding, an assumption is made that a test suite that finds seeded faults is likely to find other faults too [61].

Mutation analysis can be used in conjunction with other testing techniques to find weaknesses in test data [61], or in other words to evaluate the quality of test cases [1].

The process of mutation testing can be summarized in the following steps:

1. An automation tool accepts as an input the SUT.
2. The tool creates mutants of the SUT, and possibly eliminates equivalent mutants; those are mutants that are syntactically different from the original program, but not semantically different. Therefore, they cannot be detected or killed by any test suite although they are covered [40].
3. The set of test cases is executed against the original program.
4. The same set of test cases is then executed against the mutated program.
5. If the output of the mutant program is different from that of the original program, the mutant is said to be killed.

A mutation score is the ratio of killed mutants over the total number of unique mutants [1]. This ratio gives an evaluation of the fault measuring power of the test suite being analyzed [57]. It has been proven through experimental investigation that mutation seeding and analysis is a reliable way of evaluating test suites [2].

Chapter II Related Work

In this chapter, we discuss two groups of work from the software literature that are related to my thesis. Section II.1 focuses on research work that forms a foundation on which the contributions of the thesis are based. Section II.2 is concerned with the second group of research work, which is the group of studies that empirically compare different state-based testing criteria similar to what we propose.

II.1 Foundational Work from the Literature

As mentioned in Chapter 1, Chow is one of the most influential authors in the area of state-based testing. The method he proposed [16] is the foundation of many other state-based testing strategies. Chow suggests the so-called "automata-theoretic" strategy as a valid and reliable testing strategy for testing the control structure of a design. The method is guaranteed to reveal any error in the control structure if the machine is completely specified, minimal, starts with a fixed initial state, and every state is reachable. The author claims that the "automata-theoretic" strategy is more effective than n-switch coverage and n-switch set coverage criterion. The first step of the strategy deals with the state diagram as a graph and generates a transition tree from that graph. The author provides a procedure for generating the tree which mainly states that each node is branched only the first time the traversal of the FSM hits that node. There could be more than one transition tree for the same FSM. Each path in the transition tree that starts with the initial node and ends at a leaf node is considered a testing sequence. However, state-based testing cannot be effective without the ability to determine the resultant state after applying each test sequence. Hence, in the second step, to generate the test sequences a characterization sequence is appended

to each path in a generated transition tree. Chow defines a characterization set, containing characterization sequences, as a set consisting of input sequences that can distinguish between the behaviors of every pair of states in a minimal automaton [16]. W is a valid characterization set of a state machine if for any two states that are distinguishable, there exists an input sequence in W such that:

(i) It is acceptable in the two states and

(ii) Two different sets of output sequences are produced when this input sequence is applied to these states [66].

By doing this, Chow assumes that the system is opaque; i.e., there is no other mean of knowing the current state of the system without observing the output after applying the characterization set. This might be true for some systems, but not necessarily a limitation for most software systems. Binder refers to the characterization set by the term State Revealing Signatures as one of three approaches of determining the resultant state after applying each test sequence [6]. The three methods are:

1- State Reporter Method: This is considered the most reliable technique. The SUT implements a method that reports the state of the system. One simple way is a method that evaluates the state invariants of the SUT and returns a Boolean value that indicates if the object is in the state indicated in the input parameter.

2- Test Repetition: Repeating the test sequence and comparing the actions is suggested by Beizer [6]. This could report some corrupt states.

3- State Revealing Signatures: More expensive than the state reporter method. This is used when a state reporter method is not feasible for any reason. This method identifies a signature sequence for all states. When the sequence is applied, each state

produces different results. There are advanced techniques to produce the signature sequence. Using the signature sequence increases the size of the test suites.

The State Revealing signatures (a.k.a., the characterization sequence) is a smart way of determining the current state without the need to expose the internals of the system. Binder claims that objects can be opaque, but little modification can be applied to the system design to make it testable by simply adding observability of the state without compromising efficiency or encapsulation [6].

Based on this claim, he suggests a new method N+ [8]. The N+ method assumes that the test oracle has the capability to inquire about the current state of the SUT at any point in time. In his method, Binder adapts Chow's W-method which produces a transition tree from the state diagram. However, Binder does not append the characterization set to the resulting test cases as it is assumed that there is a mean to get the current state of the SUT without the need to observe the final output. The resulting tree includes all sequences that start and end in the same state without repeating other states. Thus, the paths in the transition tree are called Round Trip Paths (RTP). However, not all RTPs are covered in their entirety; some are covered in a piecewise manner [36, 44].

The procedure to generate all round-trip paths is as follows

1. The initial state of the state diagram becomes the root node of the transition tree.
2. An edge is drawn for every transition from the initial state, and target nodes for this added edge are added to the tree.
3. Step 2 is repeated for every node that has not already been added to the tree.

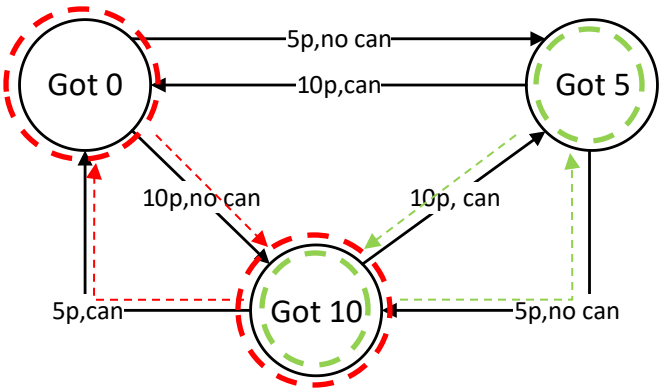


Figure 3 FSM of a vending machine

4. A leaf node is marked as a terminal node if it represents a final state or a state that has been previously added to the tree. In such a case, no more transition is added to the terminal node.

Binder mentions that his procedure gives a breadth first traversal of the FSM graph, but a depth first traversal is also possible. A depth first traversal will result in longer but fewer test sequences. Binder claims that the structure of the tree should not make a difference in producing the round-trip paths since in all cases, all round-trip paths will be included in the resulting tree (though possibly in a piecewise manner). Binder also mentions that how the different traversal trees might affect the effectiveness of the test suites is an interesting problem, but he does not explore this possibility any further.

The above claim, stating that all round-trip paths will be included in any resulting transition tree, is not always true. To illustrate this, the example in Figure 3 is a state diagram of a simple vending machine [89]. The machine only accepts 5p and 10p coins, and only vends 15p cans. If an amount of money less than 15p is deposited to the machine, no action is taken, and the machine waits for more money. Only when 15p are received,

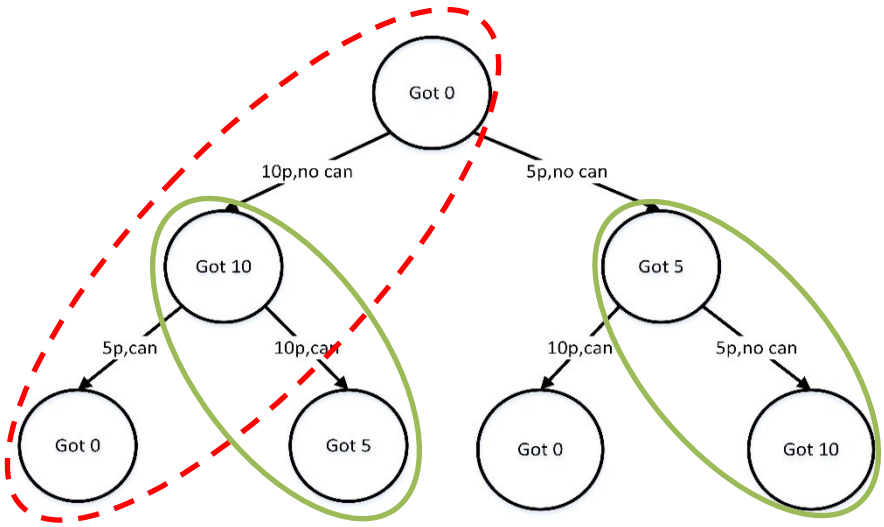


Figure 4 Breadth Traversal of the Vending Machine

the machine vends a can to the customer. In Figure 3, the path [Got 0, Got 10, Got 0][1] highlighted in red is a complete round-trip path that starts and ends at the same state (Got 0), while the path [Got 5, Got 10, Got 5] highlighted in green is also a complete round-trip path that starts and ends at the state Got 5.

Figure 4 shows a breadth first tree traversal of the vending machine, while Figure 5 is a depth first tree traversal of the same FSM. On the one hand, the tree in Figure 4 misses the round-trip path [Got 5, Got 10, Got 5] for example (highlighted in dotted red), although the pieces [Got 5, Got 10] and [Got 10, Got 5] of this RTP are in the tree. On the other hand, the tree in Figure 5, misses the round-trip path [Got 0, Got 10, Got 0], although the pieces [Got 0, Got 10] and [Got 10, Got 0] of this RTP are in the tree. It is not always the case that BFS trees fully cover round-trip paths. As marked in green in Figure 4 and

[1] Through-out the manuscript, the notation [s_1, s_2, s_3,…,s_x] is used as a representation of a path that starts at state s_1, ends at state s_x, and passes by s_2, s_3, etc [1].

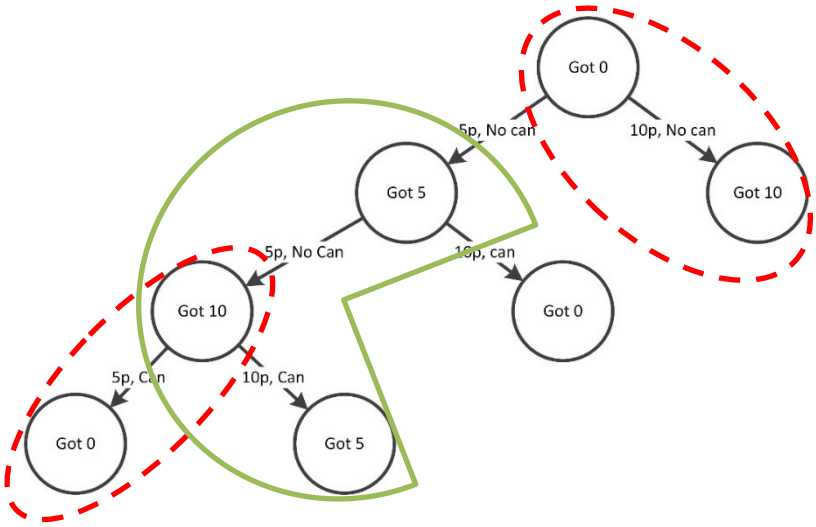


Figure 5 Depth Traversal of the Vending Machine

Figure 5, the complete round-trip path [Got 5, Got 10, Got 5] is fully covered by the DFS tree (Figure 5), and covered in pieces by the BFS tree (Figure 4).

We conclude that the different traversal trees do not necessarily cover all round-trip paths, or at least do not cover them fully. Consequently, the effectiveness of the test suites derived from the traversal trees might vary. This has been confirmed experimentally, though not in a systematic way [36, 44].

This conclusion is a motivation for an interesting area of research, where one can compare the effectiveness of the different traversal trees in testing FSMs, as well as test suites that cover all the round-trip paths in a complete manner. This is the focus of this thesis.

II.2 Similar and Complimentary Research

There are many studies dedicated to comparing the different methods for state-based testing. Few of those studies include recently developed strategies like SPY [75] and P [74]. Endo and Simão [23] compare different state machine testing strategies that are

mainly an improvement of the W method with respect to choosing the characterization sets and the cost of the test suites. Those methods are W, HSI, H, SPY, and P. The comparison is done based on the number of resets (number of test cases) in each test suite, test case length (i.e., the number of symbols in the path used as a test case), and test suite length which is the sum of all test cases lengths in one test suite. The fault detection ratio is also compared using mutation analysis. The authors conclude that recent methods (H, SPY, and P) produce fewer but longer test cases on average than the W method. SPY generates the longest test cases, while HSI generates the shortest. The P method can generate shorter test suites too. The methods are ranked from shortest to longest test suites as follows: P < H < SPY < HSI < W. Based on their observations, the authors mention in the conclusion of their paper [23] that SPY and P have the highest fault detection, while H and HSI have the lowest. The study is done using 100 randomly generated state machines with different configurations. The authors mention the fact that the similarity between these FSMs and the real FSMs used in practice is a threat to the validity of the results [20, 21]. Mutants of the specifications were used to simulate faults in the implementation of the state machines. The operators used were change initial state, change output, change tail state, and add extra state [24].

Simão and colleagues compare FSM testing techniques mainly state coverage (SC), transition coverage (TC), initialization fault (IF), and transition fault (TF) coverage [73]. TC and SC are structural coverage criteria that are explained in section I.2. IF and TF are coverage on potential faults in the FSM implementation. For the IF criteria, the test requirements address the faults that can cause a state to be wrongly used as the initial state in an FSM implementation. The TF, on the other hand, addresses the faults that might cause

a transition to produce the wrong output or end in the wrong state [73]. This is done experimentally using randomly generated FSMs using a tool implemented by the authors. The tool generates connected FSMs with a given number of states, transitions, inputs, and outputs. Experimental data on the length of tests generated using these criteria is provided as a measure of cost. Instead of generating test suites for each criterion, a test suite adequate for all considered criteria is generated, then it is minimized for each criterion separately using a minimization algorithm. The criteria under experimentation are also measured for finding mutants introduced in the FSM specifications. The authors also investigate the effect of some FSM parameters such as the number of states and the number of inputs on the length of the test suites. Results show that the number of states has the greatest impact on the length of the test suites [73].

Dorofeeva et al., [19] compare the W, WP, UIO, UIOv, DS, H, and HSI test derivation methods using 50 randomly generated FSMs and one real-life example of a protocol called Simple Connection Protocol (SCP). The Distinguished Sequence method (DS) can be thought of as a particular case of the W method [19]. The DS works in two phases, in the first phase, it makes sure that all states in the specification map to implementation states, while the second phase verifies the transitions and the outputs [28]. For the UIO and the DS, a single sequence is supposed to discriminate a state from all other states in the FSM [28]. While the W and WP methods, for instance, are always applicable to any FSM that satisfies the usual conditions (minimality, reachability of all states from the initial state, and complete specifications), the UIO, UIOv, and DS are applicable only when there exists the single distinguishing sequence [28]. The FSMs considered in the experimentation are of small size with two to six states and two inputs. However, the real-

life example has 26 states, 11 inputs, 12 outputs, and 286 transitions. All the above methods depend on the characterization sequence to distinguish between the different states and thus all states in the randomly generated FSMs used for experimentation are distinguishable. As expected, the authors conclude that the W method has the lengthiest test suites, while the rest of the test suites lengths are ordered as follows: Wp and HIS have the same test suite length, followed in decreasing order by the DS and the H, then the UIO method.

The test suites generated are compared for their completeness. A complete test suite detects every faulty implementation with respect to the considered fault model. The UIO method produced incomplete test suites in almost half of the experiments, while the DS was only applicable to 19% of the generated FSMs. In the paper, two types of implementation faults are considered; those are transfer faults and output faults [19]. The two types of faults, transfer faults and output faults, map to the TF and IF mentioned earlier respectively. However, it is not clear how the effectiveness is measured, or the faults are seeded.

Holt and colleagues [33] study the cost-effectiveness impact of the state machines selection strategies, test model and test oracle. They also investigate the application of sneak-path (extra transition) testing and how it influences the cost-effectiveness of the test suite. The evaluation is done using 26 real faults applied manually to a real case study model representing a safety monitoring component in a control system. The criteria evaluated are all transitions, all round-trip paths, all transition pairs, 2-switch, 3-switch, and 4-switch coverage. The experiment showed that all transitions pairs is the most expensive criterion while all RTPs provides the best tradeoff between cost and fault detection followed by all paths of length three. The experiments use two different test

oracles. The more rigorous test oracle increased the fault detection significantly (24% to 37%), but with a tradeoff in cost. Sneak-path testing proved to detect all faults that are undetected using conformance testing.

El-Gendy and Amer [22], study the transition tour, distinguishing sequence (DS) method, the W method, the UIO, and the UIO set methods. A transition tour is a minimum-length circular path through the FSM that visits every transition at least once [81]. The effectiveness of the methods is measured after applying them to several simulated FSMs. The authors conclude that fault coverage of the transition tour is far less than the fault coverage of the other methods, while the UIO sequence set shows slightly higher effectiveness than the rest of the methods.

Mouchawrab and colleagues [60] compare RTP testing to structural (white box) testing in terms of cost and effectiveness. Although there is no significant difference in the effectiveness of each of the methods compared, the effectiveness improves remarkably when the two testing methods are combined. The effectiveness is evaluated based on mutation analysis where the mutants are seeded in three experimental objects using MuJava [64]. Results show that there is no significant consistent difference in the cost of the two testing techniques.

Khalil and Labiche [47] study the question raised by Binder of whether the traversal trees generated using the W method are equivalent in terms of cost and effectiveness. Two experimental objects are used in our work. The authors study breadth first search (BFS) traversal, depth first search (DFS) traversal, as well as a proposed novel graph traversal algorithm. The research concludes that the effectiveness of transition trees varies, but with

low statistical significance. However, further experimentation is required to generalize the findings and compare tree generation techniques.

Chapter III Research Questions

The thesis aims at answering four Research Questions (RQs). RQ1 is related to Binder's assumption that a test suite exercising all round-trip paths in their entirety is equivalent to a test suite derived using a transition tree that covers round-trip paths in a piecewise manner [8]. RQ2 and RQ3 are linked to the question raised by Binder about whether the algorithm used to generate transition trees affects the fault detection capability of the generated test suite [8]. Khalil and Labiche [47] explored whether the transition trees used in the W-method are equivalent regardless of the used traversal algorithm. They performed their experiment on twelve test suites only. The authors concluded that the results did not have enough statistical power. Therefore, they recommended performing the study on more test suites too. The question raised by Binder has therefore never been adequately answered. RQ4 is a critical research question that if answered may lead to a possible contribution of finding a new way to generate transition trees that achieve better and more consistent fault revealing capabilities.

RQ1: Are all the faults revealed by a round-trip paths test suite also revealed by transition tree test suites?

RQ2: Does the algorithm used (BFS, DFS, or Random) to generate the transition tree affect fault detection?

RQ3: Do distinct trees generated using one algorithm differ in their effectiveness at finding faults?

RQ4: Is it possible to derive a common trend in the trees generated that helps achieve higher fault detection?

To answer the research questions above, we perform systematic experiments on four experimental objects of different sizes and from different fields. The experimental objects are described in detail in Chapter IV. To conduct the required experiments, developing automation techniques for generating and running the test suites was paramount, and therefore as part of the thesis, we implemented the required tools (Chapter V). Also, a mutation seeding and analysis framework was used as described in Chapter V. In Chapter VI, we describe the algorithms used to generate the different test suites. In Chapter VII and Chapter VIII, we present the results of the different experiments, analyze the results, draw the links between the different outcomes, and attempt to answer the above-listed research questions. In Chapter IX we come forward with four different hypotheses and analyze them. Finally, Chapter X and Chapter XI cover the threats to validity and the reached conclusions respectively.

Chapter IV Experimental Objects

Determining the effectiveness of finding faults of the test criteria mentioned earlier cannot be performed by analytical means. Like other types of criteria (e.g., data flow criteria [27]), experimental evaluations are required and as illustrated earlier, experimentation has been the preferred comparison method in the field. The Case study research methodology studies a phenomenon in its natural context. The case study research is an exploratory methodology that is well suited for many kinds of software engineering research [71]. Case study research has been followed in previous work to explore whether the different trees vary in their effectiveness using only two experimental objects [47]. In this thesis, we use exploratory research with more systems and test suites to be able to reach statistically significant results. In addition, we use explanatory research to empirically study the relationship between the nature of the trees and the effectiveness of the produced test suites at finding faults.

The five steps for conducting the empirical evaluation are experimental objects objective definition and design, preparation for data collection, collecting evidence, analysis of collected data, and reporting. The research questions chapter (Chapter III) covers the objective definition in the first step of the empirical research. The experimental setup in Chapter V is a detailed explanation of the second step; preparation for data collection. Chapter VI covers step three, while Chapter VIII and Chapter IX cover steps four and five of the methodology. This chapter covers the design of the experimental objects in the first step of the research methodology.

In the experimentation part of the thesis, four experimental objects with various characteristics and from different areas are used. The four experimental objects are a data structure representing an ordered set, an embedded controller represented by a cruise control simulator, an Automated Teller Machine (ATM), and an electromechanical device represented by a Videocassette Recorder (VCR). The four experimental objects are implemented in Java. The cruise control experimental object was originally described by Gomaa [30]. The ATM code was first generated using the State Machine Compiler [78]. The VCR experimental object is based on work done by Q. Lin [54], while the ordered set was used by others [36, 44] and is available from the Software Artifact Infrastructure Repository [18]. Table 1 summarizes the structural characteristics of the state machines of each experimental object. Note that for non-specified inputs, we assume that the FSM remains in the current state and that the output is null. This assumption augments the specification to ensure that the FSM is completely specified [68]. The FSM of each experimental object was created. In some cases, the code we received was not the exact implementation that maps to the FSM describing the experimental object. Therefore, the code needed to be debugged and modified to match the corresponding FSM. It was also necessary to make sure the code is robust enough to handle the fault seeding process. For example, try and catch statements were added to the code in order to make sure the SUT does not crash when the mutants are seeded to the code. The steps followed to create an experimental object are listed in section V.3.

In terms of complexity, we argue that the experimental objects we use are similar to the FSMs that practitioners use. For example, Lilius and Paltor use a standard case study (a production cell plant) that is provided by an industrial partner to compare formal

methods. The authors model the case study using a 16 states FSM. The connectivity of the FSM is very simple compared to the VCR and the Ordered set [53]. Other researchers use industrial experimental objects that are similar in size and complexity to the experimental objects we use in our research in different fields such as common public radio interface [77], network communication services [26], and others [31, 49, 68].

As shown in Table 1, the complexity roughly increases from the cruise control to the ATM, to ordered set, then to the VCR. The lines of code (LOC) measurement mentioned in the table measures the actual Java instruction lines and excludes comments and blank lines.

In the following four sections, each experimental object is described in detail. FSM diagrams of all experimental objects are also provided (appendices A.1 ,A.2, A.3 and A.4).

Table 1 Size of the four used experimental objects

	Cruise Control	***ATM***	***Ordered Set***	***VCR***
No. of States	5	10	9	17
No. of Transitions	29	22	35	65
Lines of Code	211	473	273	1493

IV.1 Cruise Control

The cruise control experimental object, the FSM of which is shown in appendix A.1, is a simulation of the cruising controller of a car engine. The FSM of the cruise control has four states in addition to the start state and twenty-nine transitions. The cruise control states have many self-loops, and the FSM is cyclic in nature with a cluster of transitions between the cruising and the standby states.

The Java implementation used of the system is composed of three classes; Car Simulator, Controller, and Speed Control. The Car Simulator class simulates the starting or stopping of the car engine, accelerating and breaking, as well as running the cruise control. The controller represents the cruise control and contains the speed controller. The speed control monitors and adjusts car speed when the cruising is enabled. It monitors the car speed and adjusts its throttle accordingly to maintain the speed selected for cruising. Because of the presence of many self-loops in the cruise control FSM (see appendix A.1), it is not possible to list all the transitions names on the diagram itself. So, a legend is added to map to the transition numbers to their names. The cruise control events do not have parameters (appendix A.1), and the code of the system has 211 LOC.

IV.2 ATM

The ATM presented here allows the user to either choose a saving account or a checking account. For each account, the user can either withdraw, deposit, or simply check the balance. When the user gets the receipt, the system reverts to the previous state where the user again can choose one of the actions to perform on each account. The user may enter an invalid Personal Identification Number (PIN) twice. If the third trial is an invalid PIN as well, the system quits. The ATM implementation has two classes the ATM, and the ATM context.

The FSM of the ATM (appendix A.2) has 10 states and 22 transitions. Transitions going in and out of the three validate states are in the same direction going away from the initial start state, while the five center states of the FSM form two identical clusters of states and transitions based on the choice of account either checking or saving. The ATM state

machine is implemented in 473 LOC. Some ATM events triggering transitions have input parameters such as a PIN to be validated and the amount of money to be withdrawn from or deposited to the account (appendix A.2).

IV.3 Ordered Set

The ordered set experimental object is a single java class that simulates a data structure storing a bounded set of integers. Integers can get added to the ordered set without exceeding the maximum set size. Whenever the maximum set size is reached, the data structure gets resized. However, there is a constant number of resizes allowed. In this experimental object, the ordered set size is originally two, while the maximum set size is six. Hence, the maximum number of allowed resizes is two.

The ordered set FSM that is used in the experimentation and shown in appendix A.3 has 9 states and 35 transitions. Each state except for the start state has two self-transitions. Most cycles in this state machine are either between two states or a self-transition. Every two consecutive states have two transitions between them in two opposite directions. The ordered set code has 273 LOC. The ordered set event parameters are the integer elements to be added or removed from the data structure (appendix A.3).

IV.4 VCR

With 1493 LOC as shown in Table 1 and 30 classes, the VCR is the largest experimental object used in this experiment. Fourteen out of the 30 classes are state classes, while eleven are event classes as the design follow the state design pattern.

When the VCR is in a Stopped state and the FFButton or the RewButton is triggered, the VCR changes state to FastFF or FastRew respectively. Otherwise, if the VCR

is in any other state and the FFButton or the RewButton is triggered, then the SlowFF or the SlowRew will be the next state respectively. However, in the model used during experimentation, the FF and SlowFF were combined into one state and the Rew and SlowRew were combined into one state.

The large code of the VCR reflects a large FSM diagram (appendix A.4). The strongly connected FSM has 17 states and 65 transitions. The high connectivity is evident since most states can be reached from nearly any other state in the diagram. For example, every state may accept the EjectButton or the stop button to be triggered. Therefore, from any state the destination states of Stopped or TapeAbsent can be easily reached.

The VCR with its relatively large size and high connectivity adds to the variety of structure that is already present in the experimentation process through the three other experimental objects. This wide range of FSMs enriches the empirical study and makes the results induced more representative. There is no need to specify input parameters for the VCR object (appendix A.4).

Chapter V Experimental Setup

This chapter discusses the tools used to conduct the empirical study. In the last section of the chapter, the steps needed to create an experimental object and to perform the required experiments on each of them are listed. The experimentation process needed is composed of four main independent steps. The first step is to generate the test suites based on the four algorithms previously mentioned. This is done using an automation tool that we implemented. The second step is to produce JUnit files that can trigger a test driver that runs the test suites generated in the first step. A test oracle determines whether a program under test conforms with the specification or not, while a test driver is a software module used to invoke the module under test, often providing test inputs, and that controls and monitors the SUT [70]. The test driver also uses a test oracle to verify that the current state of the SUT is the same as the state expected from the specification. The third step of the experiment is seeding the code with faults. The fourth and final step is running the JUnit tests produced in the second step using the mutated code produced in the third step in order to detect the faults covered and killed by the different test suites.

The first two steps of the experimental procedure are described in section V.1. The third and fourth steps are described in detail in section V.2. The last section (section V.3) of this chapter goes through the whole process of conducting the experiment on one object.

V.1 State-based Test Suites Automatic Generation and Execution Tool: STAGE

STAGE is a multi-step framework toolchain developed as part of this thesis [43]. To study all possible adequate[2] test suites for a criterion, automation is paramount; alternatively, producing these test suites manually is very time consuming, likely error-prone, and sometimes nearly impossible: e.g., the VCR FSM used in the thesis experiments, with 17 states and 65 transitions, leads to 101,947 different adequate test suites for the depth first traversal technique. Generating and executing the test suites or even a representative subset of them is impossible without automation tools to aid the process.

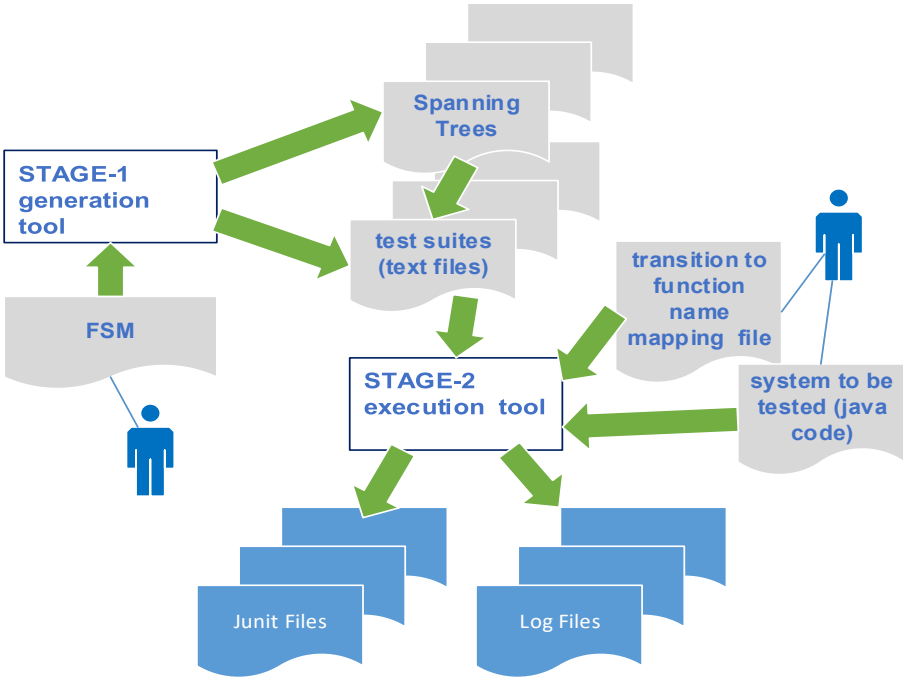


Figure 6 STAGE General Overview

[2] An adequate test suite for a criterion is a test suite that achieves 100% of test requirements mandated by the criterion. When some of the test requirements are infeasible, then adequacy is obtained by achieving 100% of the feasible requirements.

The set of features provided by STAGE (Figure 6) gives the user control over the test criteria to reach a balance between cost and effectiveness. It also allows comparing the results achieved by different test suites for drawing general conclusions on how to choose the testing criteria based on the nature of the SUT.

The toolchain (Figure 6) consists of two tools: STAGE-1 and STAGE-2. Section V.1.1 focuses on STAGE-1 that gives the user the option to produce test suites according to different state-based adequacy criteria, so far using either random, round-trip, breadth first, or depth first graph traversal algorithms. The second step of the toolchain, which is described in section V.1.2, automatically executes one or more generated test suites on the code of the SUT. It also encloses a test oracle that is responsible for checking whether the SUT conforms with the specification provided by the input FSM, or not.

The SUT is expected to implement a getState() method that evaluates the state of the system to be one of a set of enumerated state values and returns a constant indicating which state the system is currently in. Note when mutating the code, one has to make sure that the getState() method is not mutated. The getState() method is responsible for evaluating the current values of state variables, and based on that, it confirms which state the SUT is currently in. Table 2 lists the state variables evaluated of the four experimental objects we use. The state invariants for all experimental objects are available in Appendix B. For example, the getState() method of the ordered set checks the conditions in Table 3 to confirm that the current state of the system is PartiallyFilled1, which is the state of the system when there is only one element in the data structure (last=0). (see invariant descriptions in Table 2 and FSM in aAppendix B)

Our oracle relies on the current state of the system returned by the getState() method. Then, after executing each transition the test oracle is responsible for comparing the state of the system to the state expected (as modeled by the FSM) after executing each transition. If the actual state of the SUT does not conform to the expected state, i.e., the state returned by getState() is not the expected one, an exception is issued.

STAGE-2 also can run in an alternative mode to generate JUnit tests instead of directly executing the test suites. This is useful, as the JUnit tests may be used by other tools to run the test suites as is the case with the Major tool [41] that we used in experimentation as explained in section V.2.

V.1.1 STAGE-1

Step one of the automation solution generates test suites for a given state diagram. Given a graph description language (DOT) file of the state diagram representing the system

Table 2 State variables for all experimental objects.

Experimental Object	State variable	Description
Cruise Control	ignition	current ignition state true if the car is started false otherwise
	speed control	enabled/disabled
	pedal	brake setting 0...10, allowing different levels of braking
	speed	simulated car speed
ATM	PIN	the personal identification number entered by the user.
	number of attempts	the number of times the user entered an invalid PIN.
Ordered Set	resized times	the number of times the array has been resized
	actual size	the actual number of elements in the ordered set
	last	the index of the last element in the ordered set
	overflow	a flag indicating whether the data structure is overflowing
	set size	the maximum current set size; in other words, the maximum allowed number of elements before a resize.
VCR	tape	is the tape object valid or null.
	isTapePulledToDrum	true if tape puled to the drum, and false otherwise.
	isTapeWriteProtected	true if the tape is write protected, and false otherwise.
	tapePosition	the current position of the tape.
	isTapePresent	true if there is a tape, and false otherwise.

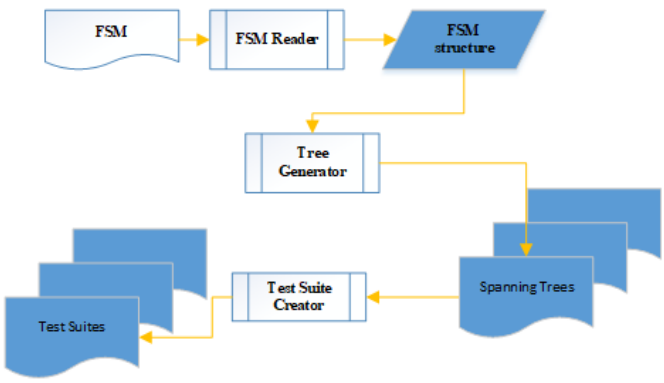


Figure 7 STAGE-1 Overview

to be tested (top left of Figure 7), the tool reads the (DOT) file and transforms it into an internal representation of the FSM graph. As shown in Figure 7, spanning trees of the input graph are then generated. The spanning tree of a connected graph G with n vertices are subsets of n-1 edges that form a connection between all vertices of G and equivalently contains no cycle [48]. The user chooses one of the four algorithms implemented as part of the tool to generate the spanning trees. Although the tool currently implements the breadth first, depth first, random and round-trip tree traversal algorithms, other algorithms can easily be plugged into the current implementation. The user or test engineer has the choice to generate the spanning trees using any of the available algorithms.

For one input state diagram, there is one or more generated traversal trees using either the breadth first, depth first, round-trip, or random traversal algorithms. Those traversal trees are then saved in DOT files, one file per tree, i.e., one file for each test suite. Finally, each of the generated trees is converted into a collection of test cases, where each test case is one complete path in the tree starting by the root node of the tree (i.e., the initial node of the state diagram) and ending at one of the leaf nodes.

The outputs produced by STAGE-1 are:

Table 3 Ordered set state invariant conditions for PartiallyFilled1 state (the invariant is a conjunction of the conditions in the table)

Invariant	Condition
resized times	< max_accepted_resizes
set size	>0
last	=0 && < set size-1
actual size	< set size
overflow	false

1. Depending on the tree generator algorithm, a collection of (DOT) files representation of generated trees, possibly reduced to one element, where each file represents one spanning tree produced by applying the chosen algorithm. In the case of the depth first traversal and breadth first traversal we have implemented, more than one (DOT) tree file is typically generated; our implementations produce all the possible traversals, given a specific stopping criterion. In the case of our implementation of a random traversal of the FSM graph, we obtain a constant number of distinctive (DOT) files. For the experimentation in this thesis, the constant number is set to ten unique random traversals.

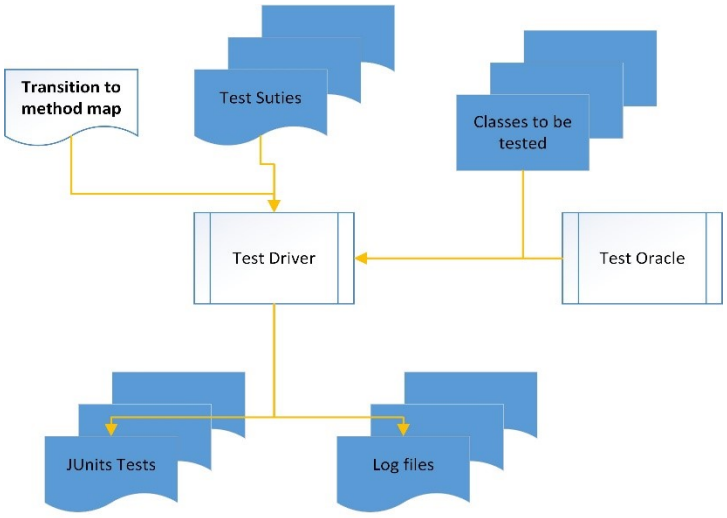


Figure 8 STAGE-2 Overview

2. A collection of text files where each file has a list of test cases. We obtain as many text files as (DOT) tree files. Each test case maps to one path in a tree. The test cases in one file collectively form one possible test suite.

V.1.2 STAGE-2

Step two of the solution shown in Figure 8, runs in two different modes. The first mode automatically executes the test suites produced in the first part of the solution on the code of the SUT. To execute the tests, STAGE-2 needs the following as inputs:

1. A text file produced from step one with the listed test cases. If arguments need to be passed to functions, this is manually done by the tester as automating this part is beyond the scope of this tool. The tester would insert the parameters in the test suite file between parentheses to signify that they are parameters. If no arguments are needed for the methods, then the test suite file that was generated by the first phase can be directly fed to the second phase.

```
0-Atm
1-Atm-String Integer Integer
2-pin-String
3-pin-String
4-pin-String
5-pin-String
6-pin-String
7-pin-String
8-returnCard
9-returnCard
10-savings
12-checking
7-exitAccount
8-withdrawal-Integer
9-deposit-Integer
10-balance
```

Figure 9 Transition to Function Map File

2. A file mapping each transition number to a function name with a listing of the types of the required parameters for this transition is required to be created manually and provided as an input to the system. The solution at this point is generic enough to handle overloaded methods. Figure 9 shows part of the ATM transition to function mapping specification provided as an input to STAGE-2. As shown in the example of Figure 9, the second constructor accepts three parameters, the first of which is of type String while the other two are Integers. Such mapping is necessary when there is a difference of abstraction levels between the FSM and the code.

The tool takes the inputs above, executes each test case, checks (using the test oracle) after executing each transition that the current state of the system under test is the same as the state expected from the provided specification (FSM), and reports the success or failure of each test case.

The other mode that the second part of STAGE-2 operates in, takes the same first set of inputs described above which is the file path or directory containing the group of text files produced by STAGE-1. From this collection of files representing the test suite, STAGE-2 automatically generates a group of JUnit test files one for each test suites. The purpose of this mode is that some mutation analysis tools work on Junit files to analyze and compare the results of executing different test suites. Hence, introducing this mode facilitates the integration between STAGE-2 and other research tools to ease conducting different types of experiments.

V.2 Major Mutation

As mentioned in section I.3, to compare the different testing strategies studied in this research fault detection effectiveness, specifically mutation analysis is used. Even though a seeded mutant does not necessarily represent a state fault, a detected mutant certainly represents a state fault. Indeed, by design, the test oracle detects state faults only since it relies on getState() that evaluates state invariants (Section V.1.2).

Although a reliable method [2], mutation analysis is a time consuming and not easy to use technique [41]. Therefore, Major is used as an automatic mutation framework to facilitate the process. There are a couple of reasons that justify using Major as a mutation tool for test suite evaluation. The first reason is that Major uses a fewer number of mutation operators to avoid the creation of many easy to find mutants. Having many easy to find mutants complicates the task of comparing the different test suites since it is likely that most test suites will detect the easy to find mutants and hence the difference between the mutation scores (explained in section I.3) of the different test suites will be too small.

The second reason is that Major is a compiler integrated tool as opposed to other tools available that are either source code mutation tools (e.g., Jester and MuJava) or bytecode mutation tools (e.g., Javalanche and Jumble). Source code mutation has the disadvantage of adding compilation overhead for each mutant in addition to introducing incompatible mutants, whereas bytecode mutation generation cannot map the mutants to the source code which makes it difficult to analyze the mutants. In addition to this, lack of type information on the source code level leads to generating invalid mutants in the case of source code mutation, while bytecode mutation has the disadvantage of introducing

nonrealistic mutants that could have never been introduced in the code by a developer [30, 32].

Major determines mutation coverage, that is the number of mutants which are hit (i.e., the line of code where they are seeded is executed) by test suites. Major also calculates the mutation score which is the ratio of killed mutants over the total number of unique mutants [1]. This ratio gives an evaluation of the fault revealing power of a test suite [80].

By measuring both mutation coverage and mutation score, Major is identifying test-equivalent mutants for a test suite, which are mutants that when executed using the test suite do not lead to an infected execution state or a state that is different from the expected state. These mutants are equivalent with respect to the executed test suite only, but not necessarily test equivalent for any other test suite. Note that test-equivalent mutants are not the same as the equivalent mutants mentioned in section I.3 [40]: a test-equivalent mutant is equivalent for a test suite but not necessarily test-equivalent for another test suite, and is therefore not necessarily equivalent (as per the definition in section I.3) whereas an equivalent mutant (section I.3) is necessarily test-equivalent for any test suite. Figure 11 illustrates these notions; it shows three test suites with different colors as an illustrative example. As shown by the legend on the right of Figure 11, a rounded corner rectangle represents the set of covered mutants, a circle with horizontal line shading represents the set of mutants killed, and a solid yellow circle is a mutant. All elements of the same color (red, blue, or green) belong to one test suite. Figure 11 illustrates the difference between the different mutants. For example, X is a test-equivalent mutant for test suites T_2 and T_3, but it is not test-equivalent with respect to test suite T_1. Therefore, X is not an equivalent

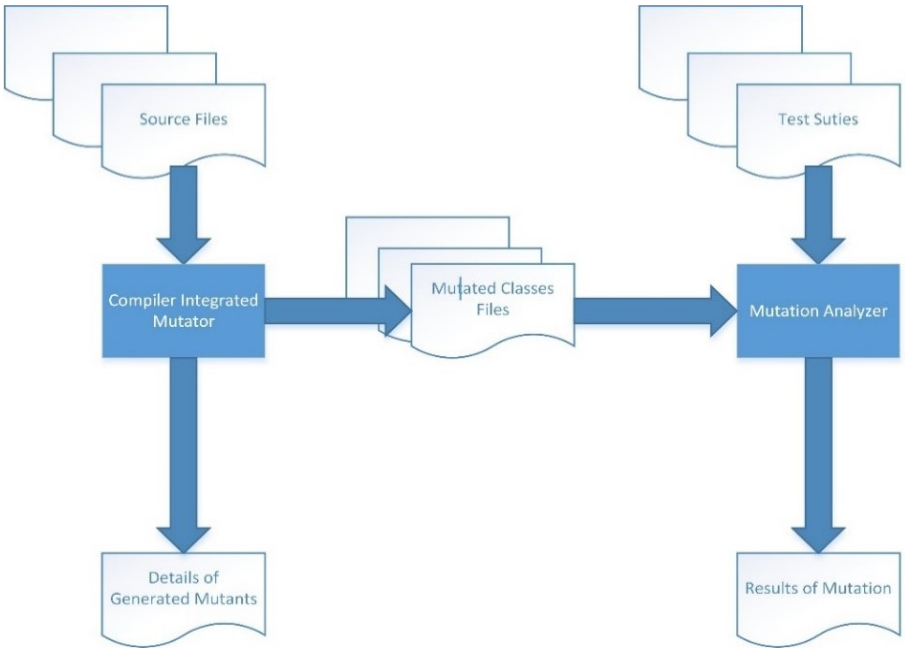


Figure 10 Major Framework Main Functionality

mutant although it is test-equivalent with respect to some test suites. Mutant Z is killed by all test suites, and hence is neither equivalent, not test-equivalent. However, mutant Y is an equivalent mutant, since it has not been killed by any of the test suites. In other words, when executing any test suite against mutant Y, the execution state of the SUT will not be

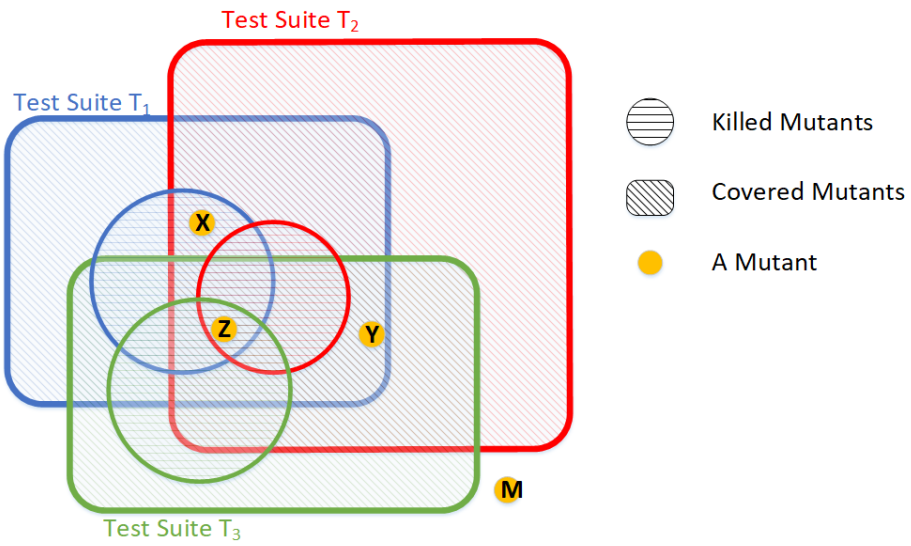


Figure 11 Covered, killed, test-equivalent and equivalent mutants.

different from the state expected by the requirements. M also is an equivalent mutant, but a one that has not been covered by any test suite.

Detecting equivalent mutants is not an easy task and is in general undecidable [3, 36, 63]. In our experiments, we consider that an equivalent mutant is a mutant that is test-equivalent with respect to each test suite executed in our experiments. We consider this a reasonable generalization since we execute a large number of test suites. After running all the test suites for each experimental object, we identify mutants that are test-equivalent for all test suites, and we consider the resulting set as the set of equivalent mutants.

When calculating the effectiveness of a single test suite, we divide the number of killed mutants by the number of seeded mutants for that test suite:

$$\text{mutation score} = \frac{killed\ mutants}{seeded\ mutants}$$

Equation 1

The existence of mutants such as Y in the previous example will affect the results:

$$\text{mutation score} = \frac{killed\ mutants}{seeded\ mutants - Y}$$

Equation 2

Including equivalent mutants in measuring the number of seeded mutants and not subtracting them (**Equation 1**), may give the illusion that the effectiveness of the test suite is low. In our study, the aim is to compare a group of test suites and not to evaluate each test suite individually, and therefore including the equivalent mutants in calculating the effectiveness will not affect our results since the dominator is the same when calculating the mutation score of the different test suites. For example, if

$$\text{mutation score for test suite 1 (MS1)} = \frac{killed\ mutants\ by\ suite\ 1}{seeded\ mutants - equivalent\ mutants}$$

Table 4 Types of mutants implemented by Major.

Operator	Description	Example
AOR	Arithmetic operator replacement.	a + b ⟶ a - b
LOR	Logical operator replacement.	a ^ b ⟶ a | b
COR	Conditional operator replacement.	a || b ⟶ a && b
ROR	Relational operator replacement.	a == b ⟶ a >= b
ORU	Operator replacement unary.	a ⟶ ~a
STD	Statement deletion operator.	foo(a,b) ⟶ no op
LVR	Literal value replacement.	0 ⟶ 1, 0 ⟶ -1
EVR	Expression value replacement	return a ⟶ return 0 int a=b ⟶ int a=0

and

$$\text{mutation score for test suite 2 (MS2)} = \frac{killed\ mutants\ by\ suite\ 2}{seeded\ mutants - equivalent\ mutants}$$

and MS1 > MS2, the inequality will not change whether the used formula is Equation 1 or Equation 2. In summary, eliminating equivalent mutants in our study is just a matter of scale of the measurement (mutants) not a matter of validity. However, we calculated the equivalent mutants and listed them as this piece of information can be useful for further analysis (Chapter VII).

Although some covered but alive mutants may not be equivalent, as this depends on the tests we use, we considered a mutant that is covered and alive with all our test suites as equivalent. Given the proof above and the large, to the very large number of test suites the thesis experiments with, this is a reasonable assumption. Plus, since test suites are compared between one another, this assumption should not introduce a significant threat to validity. In the experiments proposed, all the mutation operators available with Major are used, when applicable.

Major runs in two independent steps. The first step compiles the source code of the SUT, generates mutants, and embeds them in the code during compilation. Major's mutator is integrated into the OpenJDK Java compiler. The second step runs the mutation analysis. For the mutation analysis step, Major has an analysis backend that extends Apache Ant's JUnit task [39]. Figure 10 illustrates the main components, inputs, and outputs of Major.

Major supports a set of commonly applied mutation operators. This includes five major categories; binary operators (e.g. arithmetic, logical, shift, conditional and relational operators) replacement, unary operators (e.g. negation) replacement, constant value replacement, branch condition manipulation, and statement deletion [39]. Table 4 lists the mutation operators supported by Major and illustrates them by examples.

V.3 Running Experimental Objects

This section summarizes the steps that can be used to replicate the work done to implement and run an experimental object. This is useful to extend our work by other researchers. All the required resources are available via our GitHub repository [46]. The itemized steps below correspond to the numbered activities of Figure 13.

1. An FSM diagram representing the experimental object is created using the Graph Description Language (DOT). It can be visualized using Graphviz [90]. The file that models the experimental object is then fed to STAGE-1, and a traversal technique is chosen (Breadth, Depth, Round-trip, or Random).
2. STAGE-1 generates all the traversal trees of the FSM diagram. From each generated tree, STAGE-1 produces one test suite that is composed of all tree paths from the root node to a leaf node (test sequences).

```
[0] Start-87(1 2 3 4 5)-PF5-37(6)-F6-41(8)-PF7-42(8)-PF6-44(6)-xxPF6
[1] Start-87(1 2 3 4 5)-PF5-37(6)-F6-41(8)-PF7-42(8)-PF6-43(7)-xxxxxPF7
[2] Start-87(1 2 3 4 5)-PF5-37(6)-F6-41(8)-PF7-42(8)-PF6-45(9)-xxxPF6
[3] Start-87(1 2 3 4 5)-PF5-37(6)-F6-41(8)-PF7-42(8)-PF6-46(6)-xxxxxxxxPF5
```

Figure 12 Parametrized Test Suite Sample

3. The researcher/test engineer adds parameters when needed for each test suite. Figure 12 shows a sample of how the parametrized test suite looks like. Each line corresponds to one test case (tree path). For example, in test case 0 of Figure 12 PF5 is a state name, while 37 is a transition number that maps to a method that takes an integer parameter. The test engineer, in this case, specified the parameter value to be 6.
4. The parametrized test suites, as well as the experimental object code, are fed to STAGE-2, and the mode that exercises the test suites on the experimental object code is chosen for this step. There are few details in this step that have been omitted from Figure 13 for simplicity. They are:
    a. An experimental object class inherits from a provided abstract java class that has the getState functions, and each experimental object implementation implements this function. This is to guarantee that the test oracle can use the getters to check the current state of the object.
    b. When an experimental object is implemented, or the code is supplied to STAGE-1, one has to make sure that the code has some error handling mechanism in order not to crash when the mutants are seeded. For instance, each small block of statements can be surrounded by try and catch so that the SUT can be resilient enough to survive the mutants inserted by Major without crashing.

c. In addition to the test suites and the code, STAGE-2 needs a way to map the transition numbers to actual functions. Therefore, it accepts a transition to function map file as the one shown in Figure 14. The file also specifies the type of arguments for each transition/method as shown by the provided format. Parameters are separated from functions by a '-', while spaces are used as separators between the different parameters of the same function. Since STAGE-2 supports arrays as parameters, two square brackets "[ ]" are used to indicate that the parameter is an array.

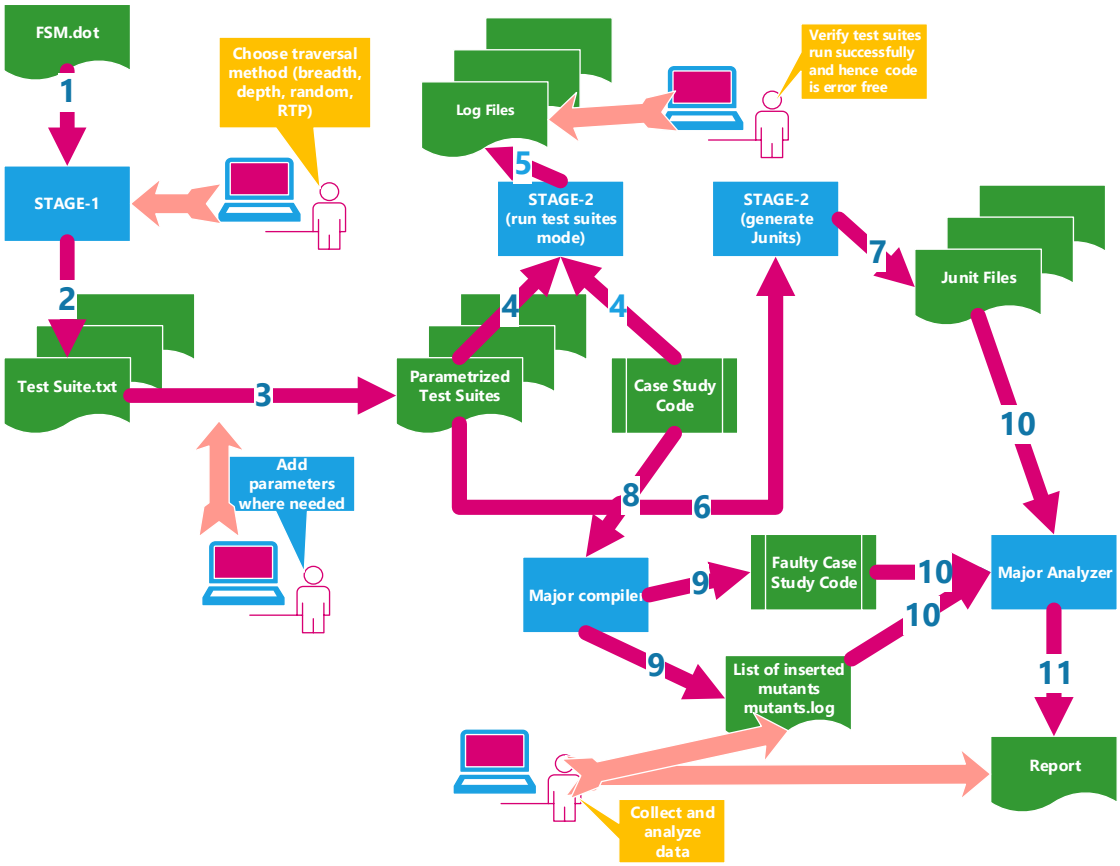


Figure 13 Running a Case Study System

5. STAGE-2 runs using the inputs listed in the previous steps (test suites and transition to function map) and it produces log files. If the log file is empty, this means all test cases passed and the experimental object code is error-free. Otherwise, each reported failure in any experimental object should be resolved before moving to the following step.
6. The parametrized test suites are fed again to STAGE-2, but this time to produce all JUnit files.
7. STAGE-2 produces all the JUnit files, where each JUnit file maps to one test suite.
8. The experimental object code is fed to the first step of Major described in section V.2.
9. Major produces the faulty version of the experimental object code files. It also provides a log file that specifies the seeded mutants and their locations in the code.

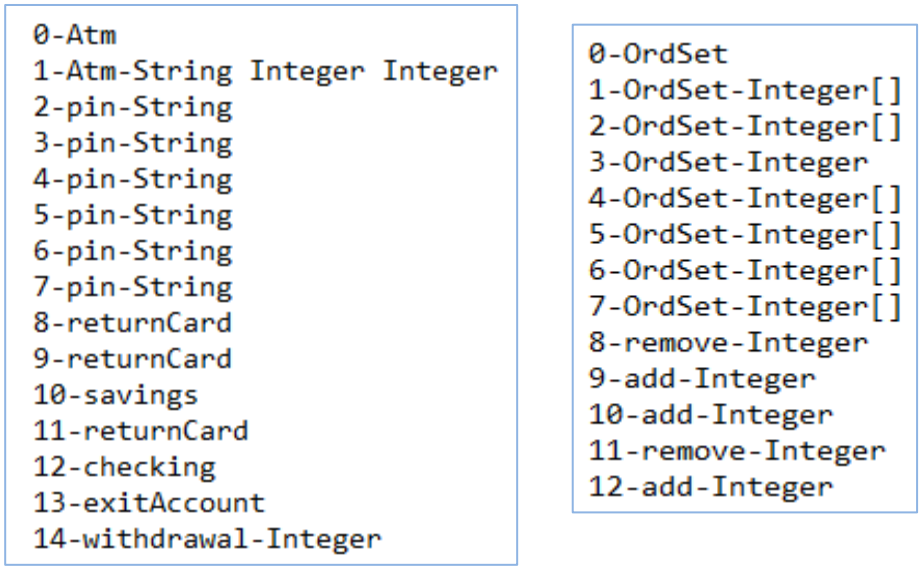


Figure 14 ATM and Ordered Set Transition to Function Map Snapshot.

10. The faulty version of the code (from step 9), the mutation log file (from step 9), and the JUnit files (from step 7) are the inputs to the second step of Major described in section V.2.
11. Finally, Major runs the experimental objects and based on assertion failures (in the JUnit tests) produces a report indicating which mutants have been killed and which mutants have been covered by the test suites.

Chapter VI Test Suites Generation Algorithms

The thesis compares four types of test suites. Three of them are deduced from transition trees; those are the BFS, DFS, and random traversals, while the fourth is the round-trip paths. This chapter details each algorithm. BFS and DFS traversals are explained in sections VI.2 and VI.3 as they are based on similar concepts and they produce all possible trees for each traversal technique. Sections VI.4 and VI.5 describe the random and the round-trip algorithms respectively.

The graph BFS and DFS traversals algorithms we use in this thesis differ from the conventional graph traversal algorithms. What the devised algorithm is trying to achieve is not just one breadth first/ depth first traversal of the input digraph, but rather a collection of all the possible breadth first traversals of that graph. The motivation behind creating all possible breadth first/ depth first traversal trees in the case of generating test suites for a SUT is the need to compare the effectiveness of all possible breadth first traversal trees test suites of the original state diagram.

A directed graph may have many spanning trees. The proposed algorithm is based on the intuition that the reason behind having more than one spanning tree of a single directed graph with a known original node is having nodes with more than one incoming edge; i.e., nodes with indegree > 1 [45]. Each of those nodes is referred to as a Multi Incoming Edges Node (*MIEN*). Nodes 1, 3, and 4 in Figure 15 are MIEN nodes, while nodes 0, 2, 5, and 6 are not.

An important observation is that if the MIEN node is a terminal node of the graph (i.e., it does not have outgoing edges), then that MIEN node is not significant for creating

a different traversal tree, since the traversal path terminates at this node anyway and it will never form a root of a subtree. To be able to present the algorithms, some background information needs to be solidified.

VI.1 Background and Related Work

In a graph G, where there are vertices v_1, …, v_n an edge is noted by $e_i = v_{i-1}v_i$ for i=1, ..,n. A walk in G is a sequence of edges. A walk for which v_i are pairwise distinct is called a path. A trail, on the other hand, is a walk for which e_j are pairwise distinct [37]. For example, in graph G_1 of Figure 15 (e_1, e_4, e_9, e_3, e_7) is a trail, but not a path since Node 1 is traversed twice.

A spanning tree vertex has only one incoming edge. In some applications, a spanning tree that covers all edges, as well as all vertices, is required [15]. Since the spanning tree generation algorithm introduced in this thesis (BFS, DFS, and random) are developed to produce test suites for software artifacts, the desired generated trees are supposed to cover all edges and all vertices, since edges, in this case, represent operations in the SUT. A full spanning tree of a graph G contains all the edges in *G* and has no cycles. Because *G* represents an FSM, which is assumed to be connected, this ensures a full spanning tree also contains all the nodes. While the traditional spanning tree is a collection

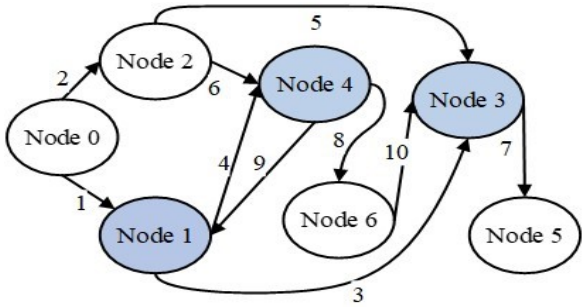


Figure 15 Directed Graph G_1

of paths, the full spanning tree is a collection of trails. For the sake of simplicity, throughout the rest of the document "spanning trees" is used to refer to full spanning trees.

Usually, there are many spanning trees for a connected graph. A reasonable way to generate spanning trees of a graph is to start with one spanning tree, add an edge to it to form a circuit, then remove any other branch to break that circuit. This is called elementary tree transformation [62].

One of the very early attempts to generate all spanning trees of a graph was Char's algorithm that generates all combinations of $n-1$ edges, then examines each combination to decide whether it forms a tree or not [14]. The algorithm runs in $O(E+N+N(T+T'))$ where T is the number of generated trees, while T' is the number of generated combinations of edges that do not form trees [36].

Mayeda and Seshu developed an algorithm that is based on elementary graph transformations and generates the trees without duplicates. The idea is to start with a root spanning tree to generate all the other spanning trees by an exhaustive search when replacing each edge [56].

Minty introduced another idea by generating a partial tree and adding one edge at a time to that tree. If the added edge converts the tree to a graph by forming a cycle, all the edges forming that cycle are removed from the graph. The algorithm's time complexity was $O(N \times E+N \times E \times T)$ where T is the total number of generated trees, while the space complexity is $O(N \times E)$ [36].

Hakimi introduced two methods for generating the trees of a given undirected graph. The first algorithm produces a duplicate tree [31], while the second has high time complexity as it was not the result of elementary transformations [21].

Gabow and Myers introduced an algorithm for generating all spanning trees of directed and undirected graphs [29]. The time complexity for this algorithm is $O(N+E+E\times T)$ and the space complexity is $O(N+E)$.

Matsui's algorithm is based on the idea of generating one spanning tree of an undirected graph and then adding an edge to form a cycle. Removing another edge of the cycle introduces a new spanning tree. The algorithm runs in $O(N+E+T)$ time, and $O(N+E)$ space complexity [55].

Another improvement to the generation of spanning trees of graphs is by Kapoor and Ramesh who present three algorithms for enumerating all spanning trees in directed, undirected, and weighted graphs. They use the concept of computation tree where each node represents one spanning tree of the undirected graph. They use edge replacement to transform a tree and add a child tree to the computation tree. The first algorithm handles undirected graphs and runs in $O(N+E+T)$ with space requirement of $O(N2\times E)$. The second algorithm is for directed graphs and runs in $O(TN+N3)$. The third algorithm generates spanning trees of weighted graphs in $O(N\times E+T\times logN)$ time and $O(N\times E+T)$ space [42].

An algorithm that uses an initial minimum spanning tree and generates the rest of the trees in order of increasing cost using edge exchange is presented by Sörensen and Janssens. Time complexity is $O(T\times E\times logE+T^2)$ and space complexity is $O(T\times E)$ [76].

Another algorithm requires $O((N+E) \times (T+T'))$ time where T' is the set of generated graphs that contain cycles and thus do not qualify as trees [13]. The algorithm is based on classifying the graph edges into two categories; pendant edges that are not part of a graph cycle, and circuit edges. Pendant edges are included in all trees. Circuit edges are parts of cycles in the graph and are analyzed to obtain all possible trees [13].

Knuth in volume 4 of his famous series "The Art of Computer Programming", lists the steps to visit all the spanning trees of a directed graph represented by a doubly linked list. He uses a technique called "dancing links" to remove and restore items from and to doubly linked lists to produce a new tree. So, the idea is very similar to swapping edges of an initial tree to get the rest of the spanning trees [48]. The complexity of the algorithm is not calculated.

Through researching work that covers generating all spanning trees we claim that to the best of our knowledge there is no available work in the literature that generates all BFS distinctive trees neither all DFS trees. The algorithms surveyed express the complexity in terms of the number of generated trees which is an output, not an input parameter. The BFS algorithm proposed for example in my thesis runs in $O(E+T)$ if expressed in terms of the output which is better time complexity than any of the mentioned previous algorithms. However, our solution can be improved in terms of space complexity as described in section VI.2.2.2. It is also worth mentioning that there is another difference between our algorithm and previous algorithms as the generated trees span all edges as well as all vertices. All the trees generated by the proposed algorithms are unique.

VI.2 Breadth First Traversal

A Breadth First Traversal of a graph is defined as follows: Given a graph $G = (V, E)$ and an initial vertex s, BFS systematically explores the edges of G to find every vertex reachable from s. It produces a "breadth first tree" with root s that contains all reachable vertices [17].

Figure 15 represents an example graph G_1, that is used throughout this section to explain the BFS traversal algorithm.

VI.2.1 The Algorithm

To produce two different BFS trees, the graph should have two edges originating from two different nodes on the same level (at the same distance from the initial node) and leading to the same non-terminal destination node. However, that destination node must be on a different level that is further from the root node (deeper) than the two source nodes, because if the destination node was not deeper, this means it would have been branched earlier anyway since we are working with BFS trees only. The algorithm starts by creating

```
Algorithm 1: CreateInitialBreadthTree
Input: A directed graph G with root node n_root
Output: A breadth first spanning tree T

1   BFST(G)
2   n_root ← GET_INITIAL_NODE(G)
3   T← SET_ROOT(n_root)
4   ENQUEUE (Q, n_root)
5   Do
6     n ← DEQUEUE(Q)
7     do if (not LEAF(n))
8       n_current ← Copy(n)
9       edges ← GET_OUTGOIN_EDGES (G, n)
10      SET_BRANCHED (n, true)
11      for each e ∈ edges
12        n_d ← GET_DESTINATION_NODE(e)
13        ADD (T, n_current, e, n_d)
14        If (not BRANCHED (n_d)
15          ENQUEUE (Q, n_d)
16        Else
17          ADD (MIENs, n_d)
18        Endif
19      Endfor
20  EndifWhile (not EMPTY(Q))
```

Figure 16 Creating Initial BFS Tree

an initial BFS tree. Then from that tree, a collection of all other possible unique breadth first traversal trees, that are modified copies of the first tree, are generated. The BFS algorithm used runs in two main stages as described below.

VI.2.1.1 Creating the Initial Breadth First Traversal Tree

The first step of the BFS Trees (BFST) generation algorithm is to create an initial breadth first traversal tree. Figure 16 is the pseudocode for the conventional algorithm used for this step [49]. The algorithm starts from the initial node of the input directed graph and adds that node to a first in, first out data structure (FIFO) referred to as Q (Figure 16: line 4). Then, the algorithm starts consuming node by node from Q (line 6). For each consumed node, the algorithm iterates on the outgoing edges of that node and inspects one edge at a time (line 9). The destination node n_d of the currently processed edge is retrieved (line 12). If n_d has never been processed (i.e., branched before), n_d gets added to the FIFO data structure Q in order to process it at a later stage (line 15). This way, this node will never be processed before all the nodes that are already in Q. At any point in time, the stored nodes in Q are of a higher level or closer to the root of the tree than the node being added at the moment by the BFST algorithm. When a node n_1 is referred to as being closer to the root of the graph than another node n_2, this means that to reach n_1 from the root node, fewer edges are needed than the edges needed to reach n_2. If n_d has been added to the resulting tree T before, this indicates that n_d is a MIEN node and that this is not the first time the

Create Sister Tree
Input: A tree t, two MIEN nodes : n_{sec} and $n_{origina}$
Output: A tree identical to the input tree except for the subtrees rooted at n_{sec}, $n_{original}$. The two subtrees are switched in the new t_{new}.

CREATE_SISTER_TREE (t, n_{sec}, $n_{original}$)
1 $t_{new} \leftarrow$ Clone(t)
2 $t_{new} \leftarrow$ Swap_Leading_Edges(t_{new} ,n_{sec}, $n_{original}$)

Figure 17 Creating Sister Tree

current traversal reaches n_d. In such a case, n_d is marked as MIEN, a prefix is added to its name to indicate that it is a copy of a previously traversed node, and it gets added to the list of MIEN nodes (line 17). Then the algorithm adds n_d to the resulting tree T. The colony creation algorithm of Figure 18 makes use of this data.

Finally, the breadth first traversal algorithm terminates when all the nodes in Q are consumed. The following section describes how to make use of this initial tree to generate all possible breadth first traversal trees of the graph to construct a breadth first colony of trees.

VI.2.1.2 Constructing a Breadth First Colony

The algorithm concerned with generating all unique BFS trees, the pseudocode of which is shown in Figure 18, starts by calling the BFST of Figure 16 algorithm to create an initial BFS tree (Figure 18: line 1). After creating the initial tree, the algorithm loops on all MIEN nodes of that tree that were identified in the first step of the algorithm while creating

```
Algorithm 2: CreateBreadthColony
Input: A directed graph G with root node n_root
Output: C: A colony of all breadth first spanning trees of the input
graph


CREATE_BREADTH_COLONY (G)
1     t_i ← CreateInitialBreadthTree (G)
2   ADD (C, t_i)
3   nodes ← GET_ALL_MIEN_NODES(t_i)
4   for (each n ∈ nodes)
5     TreesToUpdate ← SIZE(C)
6     n_original ← n
7     If (not HAS_DUPLICATE_PREFIX (n_original))
8       n_sec ← ADD_PREFIX(n_original)
9       do
10        If (SAME_LEVEL (n_sec, n_original)
11          for (each t ∈ TreesToUpdate)
12          t_sis ← CREATE_SISTER_TREE (t, n_sec, n_original)
13          ADD (C, t_sis)
14          End for
15        n_sec ← x+ n_sec
16      while (not GET_NODE (t, n_sec) ≠ null)
17    Endif
18  End for
```

Figure 18 Creating Breadth First Colony

the initial BFS tree (line 4). For each MIEN node ($n_{original}$), the algorithm loops on all of the other copies of that node (line 9). Other copies of MIEN nodes refer to the unordered pairs of versions listed for each MIEN node in the first step of the algorithm. In other words, the algorithm loops on all the different ways to reach that node (line 10 to 15). If a copy n_{sec} is at the same level of $n_{original}$, the subtrees whose roots are $n_{original}$ and n_{sec} gets swapped to create a new tree (Figure 17) and that new tree gets added to the list of trees constructing the colony *C*.

If the two copies of the same node are not at the same level in the initial breadth first traversal tree, no switching takes place, and we move to the next copy of the node to examine it. The reason for this is that if the two nodes are not at the same distance from the root, the BFS traversal, by definition, reaches the node through the path that is shorter from the root before the other longer path that is further from the root node. Switching the subtrees means allowing the traversal to reach the MIEN node through a longer path which violates the BFS traversal concept. After examining all the copies and performing the required switches another node of the original tree is examined until all the nodes belonging to the initial tree are consumed.

It is worth noting that based on the BFS performed in the first step, the order in which nodes are consumed from the queue is based on their distance from the root. So, no child node is consumed before its parent. Otherwise, children will be swapped in trees where parents are not swapped yet and this may result in having duplicate trees and missing BFS trees.

VI.2.2 Complexity Analysis

In this section, we calculate the time complexity and the space complexity of the BFS traversal algorithm.

VI.2.2.1 Time Complexity

Theorem #1: BFST runs in *O(E)* time.

Proof: The algorithm in Figure 16 has two nested loops. The outer loop has a number of iterations equal to the number of nodes in the graph, while the inner loop has a number of iterations equal to the outer degree of each node. Therefore, the complexity of the algorithm is $O(N \times D)$ where D is the average outdegree of the vertices. However, $N \times D$ is the average number of edges outgoing of each node multiplied by the number of nodes, which is simply the total number of edges in the graph. Thus, $N \times D = E$ where E is the total number of edges in the graph. This proves theorem #1.

This is also logically intuitive since the BFST visits each edge of the original graph G once.

Theorem #2: The algorithm CreatingBreadthColony in Figure 18. has time complexity of $O\left(E+\frac{E^N}{N^N}\right)$ or in function of the output trees $O(E+T)$ where T is the number of generated trees.

Proof: The algorithm calls Algorithm 1, the time complexity of which is *O(E)* from Theorem #1, then it goes through three nested loops. The outermost loop iterates on MIEN nodes that are not leaves. Assuming a worst-case scenario, where all nodes in the graph are MIEN nodes, the number of iterations for this loop will be N where N is the total number

of nodes in the original graph. The middle loop loops on all copies of each MIEN node. Given the previous worst-case scenario assumption, the worst-case scenario for the number of iterations for this loop will be the average number of edges per node. This assumption guarantees that the complexity calculated here is the highest possible complexity for our algorithm, which strengthens our argument that our algorithm has lower complexity relative to what is available in the literature (section VI.1). This number is smaller than $\frac{E}{N}$. The most inner loop iterates on all previously created trees *t*. As discussed in the introduction to this chapter (Chapter VI), the reason for having more than one tree is the different possible positions that a MIEN node can take in each tree. Therefore, the number of possible trees is equal to the total number of combinations for all possible locations for the MIEN nodes, and hence *t* is always less than that number. In the worst-case scenarios (all nodes are MIEN), this number in the worst-case scenario is equal to $\frac{E^N}{N^N}$. Creating a sister tree (Figure 18: line 12) clones the original tree by creating a copy of it. The complexity of this step is *O(N)*.

Hence, the total complexity of the algorithm is $O\,(E + N \times N \times \frac{E}{N} \times \frac{E^N}{N^N}) = O\,(E + E \times N \times \frac{E^N}{N^N}) \approx O\,(E + \frac{E^N}{N^N}) \approx O\,(E + \frac{E^N}{N^N})$. This again can be logically deduced since the total number of iterations the algorithm goes through is equal to the number of generated trees. This number is at the maximum equal to all combinations of copies of MIEN nodes as explained in the introduction to Chapter VI. Assuming that each MIEN node has $\frac{E}{N}$ (average indegree of a node) copies, and that all the nodes in the tree are MIEN nodes. These assumptions again give the worst-case scenario possible which guarantees the

complexity we deduce will never be exceeded for any given graph. Then, all possible combinations are $\frac{E^N}{N^N}$ which is the number of trees generated. When we add the iterations on all edges to get the MIEN nodes, the result is $O\ (E + \frac{E^N}{N^N})$ as suggested by theorem #2.

VI.2.2.2 Space Complexity

Theorem #3: The resulting trees of the algorithm occupy $\frac{E^N}{N^N}$ space.

Proof: As explained in the previous example the number of trees generated is equal to the maximum number of all combinations of ingoing edges per node. The average number of ingoing edges for each node is $\frac{E}{N}$. Therefore, all possible combinations are equal to $\frac{E^N}{N^N}$.

However, the space occupied can be drastically improved as each tree differs from another by only a swapped edge and therefore a swapped subtree. A modified algorithm can store the difference between the trees only rather than the whole set of trees. This will result in a space complexity of $O(\frac{E}{N})$.

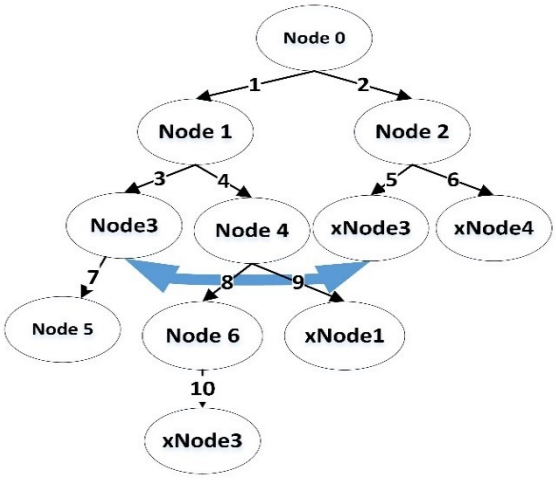
VI.2.3 Example

To illustrate how the algorithm works, Figure 15 (page 48) shows a directed graph G_1 that has seven nodes and ten edges. Three of the seven nodes are MIEN nodes. Node 1, Node 3 and Node 4 are non-leaf MIEN nodes that affect the number of possible spanning trees of G.

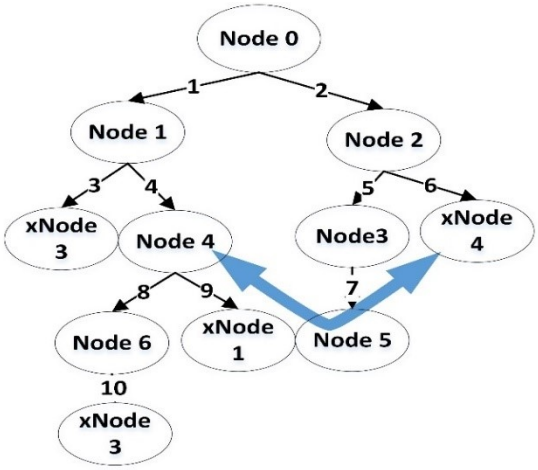
If we assume that Figure 19.a is the initial spanning tree produced by the algorithm in Figure 16, and Node 3 is the first MIEN node to be processed. Swapping the subtree rooted at Node 3 and its copy xNode3 (prefix x is added since it is MIEN node as explained in VI.2.1.1) that is at the same level results in the subtree rooted at Node 3 including Node

5 to be descendent of Node 1 rather than Node 2 as shown in the second spanning tree of G in Figure 19.b.

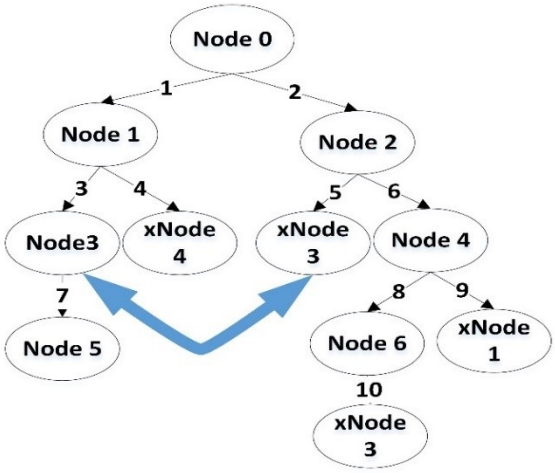
There are no other copies of Node 3 to be switched with the current position of Node 3 to produce a new tree since the third copy of Node 3 is at a further distance from the root. Therefore, in a breadth first traversal algorithm, the third copy of Node 3 can never be branched at that position. Therefore, the algorithm moves to the second MIEN node, which is in this case Node 4 that is a child of Node 1. By swapping Node 4 and xNode 4



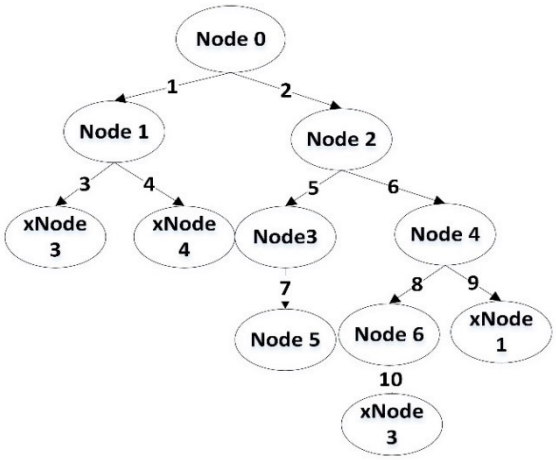
a) BFS Tree #1 of G



b) BFS Tree #2 of G



c) BFS Tree #3 of G



d) BFS Tree #4 of G

Figure 19 BFS Spanning Trees of Graph G_1 in Figure 15 (page 48).

that are at the same level (in Figure 19.a), a new tree gets produced as shown in Figure 19.c.

The next step is providing another copy of the tree in Figure 19.a in order to have the combination where both Node 3 and Node 4 are both descendants of Node 2. This step is performed by the tree updating loop of the algorithm (Figure 18: line 11). The algorithm creates a copy of the second tree to update (Figure 19.b) and swaps Node 4 and xNode 4 to produce the fourth spanning tree shown in Figure 19.d.

Although Node 1 is a MIEN node, no swapping is possible since the two possible positions of Node 1 are not on the same level in the tree and hence any breadth first traversal will reach Node 1 as a root first before the copy xNode1.

VI.3 Depth First Traversal

Depth First Traversal is a tree/graph search algorithm which extends the current path being traversed as far as possible before backtracking to the last choice point and trying the next alternative path [17]. A depth first traversal algorithm should handle the cycles in a graph by avoiding branching a node more than once. All the resulting depth first traversal trees should include all vertices and edges of the original graph.

In the depth first traversal, the algorithm tries to move as deep as possible in one direction before backtracking to branch in another direction. Therefore, the nodes that are traversed immediately after one another have the same traversal path starting from the root node until the point where the depth first traversal has no solution but to backtrack and branch to reach a new node. In other words, when the algorithm reaches one node in a DFS traversal, it tries to reach the other nodes in a way that guarantees they have the longest

common paths prefixes from the root with the nodes that have already been added to the tree.

The DFS traversal algorithm is also based on the MIEN node hypothesis discussed in section VI.1. Similar to the BFS algorithm, an initial tree is generated through DFS traversal. However, the rest of the trees are generated by switching the locations where the MIEN nodes are branched. Swapping the MIEN nodes for the DFS algorithms requires ensuring that the resulting tree is a DFS tree. To find all depth first traversal trees, one needs to calculate all combinations of paths leading to MIEN nodes that are non-leaf nodes. To limit the resulting trees to DFS trees only, each combination should consist of the paths with the longest common prefixes (i.e., depth) among the paths leading to those MIEN nodes. The common prefix is simply the longest common paths (sequence of edges) between the paths from the root to MIEN nodes.

VI.3.1 The Algorithm

The algorithm starts by generating one depth first tree using the standard DFS algorithm (Figure 20: line 1). This tree is the base tree for the rest of the traversal trees. All other DFS trees are simply variations of this initial tree.

Algorithm Create Depth Colony
Input: A directed graph G with a root node $N_{initial}$
Output: All possible depth first trees of the input graph **TreeList**

```
1   T0 ← DFS (G, Ninitial)
2   Clist ← CreateDepthPathsCombinations (G)
3   i ← 0
4   for each C ϵ Clist
5     Ti ← Clone(T0)
6     QuickSort(C)
7     for each Path p in C
8       for each Edge e in p
9         N1 ← GeMIENPosition(e)
10        N2 ← GetMIENPositionInInitaialTree(N1)
11        SwitchSubtrees(N1, N2)
12      endfor
13    endfor
14    TreeList = TreeList + Ti
```

Figure 20 Creating All Depth Trees.

Assuming the input graph G has l number of MIENs ($N_0, N_1, \ldots, N_i, \ldots, N_l$). For each MIEN node N_i, there exists a list of all possible m paths ($N_i P_0, N_i P_1, \ldots, N_i P_i, \ldots N_i P_m$) to reach N_i from the root node. To generate all possible depth first traversal trees, all combinations of paths are created (line 2) where each combination consists of the paths with the longest common prefix (deepest common path). A combination C consists of l

Algorithm CreateDepthPathsCombinations (Recursive)

Input: A directed graph **G**, each ***MIEN*** node $N_i \in$ **G** stores all paths $N_i\mathbf{P_0}\ldots N_i\mathbf{P_x}$ that leads to node N_i, where $N_i\mathbf{P_j}$ is a sequence of edges $\mathbf{e_0,..,e_k}$ that connects the initial node N_0 to Ni
a MIEN node N_d : Initially the first MIEN node in the list of MIEN nodes.
a combination of paths C: Initially empty.
Output: A list **Clist** of all possible combinations $\mathbf{C_0}\ldots\mathbf{C_z}$ of paths that does not violate the depth first traversal criteria.

```
1    PList ← GetLeadingPaths(Nd)
2    for each path P in PList
3     pathRejected ← false
4     for each path Pi in C
5       L0 ←GetCommonPrefixLength(P, Pi)
6       Ni ← GetMIENNode(i)
7       for each path Pj in GetLeadingPaths(Ni)
8         L1 ←GetCommonPrefixLength(P, Pj)
9         If (L1 > L0)
10          PathRejected ←true
11        endif
12      endfor
13      if (PathRejected = true)
14        for each path Pl in GetLeadingPaths(Nd)
15          L2 ←GetCommonPrefixLength(Pl,Pi)
16          if L2 > L0
17            PathRejected = true
18          Else
19            PathRejected = false
20          endif
21        endfor
22      endif
23      endfor
24      if (pathRejected = false)
25        C = C + P
26        CreateDepthPathsCombinations( G, C,Nd+1)
27        if ( size( C ) = number of MIEN nodes)
28          Add ( ListOfDepthCombinations, C)
29        endif
30      endif
31      endfor
```

Figure 21 Creating Depth Paths Combinations

number of paths, where l is equal to the number of MIEN nodes, as there is a path for each MIEN node in order to produce a new spanning tree. $C = \{N_0 P_x, \ldots, N_Z P_{y, \ldots,} N_l P_k\}$, where $0 \leq x, y, k < n$ and $0 \leq z < l$, while n is a variable that changes for each node to be equal to the number of paths leading to each node. To ensure all paths added to one combination do not violate the depth first criteria, i.e., the paths have the longest possible prefix, the following steps are followed:

1. The algorithm takes the path $N_0 P_0$ from the list of paths leading to MIEN node N_0, and adds that path to the current combination of paths C (initially empty).
2. The next MIEN node N_1 is considered. The first path leading to N_1, which is N_1P_0, is examined to make sure that if added to C, the depth first traversal will be maintained. To do this, the algorithm:
    a. Calculates the intersection between N_1P_0 and N_0P_0: $I_0 = N_1P_0 \cap N_0P_0$ (line 5)
    b. Then, loops on all paths leading to node N_0 other than N_0P_0 (i.e., N_0P_1, $N_0P_2, \ldots, N_0P_n$) and calculates the common prefixes between each of those paths and the path N_1P_0 (line 8). For example, $I_1 = N_0P_1 \cap N_1P_0$. If $I_1 > I_0$, this means that node N_0 can be reached through a path that is deeper in the subtree containing N_1P_0. To clarify, if the path N_1P_0 is the one first traversed in a tree, it is not possible to choose N_0P_0 as the path leading to N_0 in that tree (line 10). However, if the opposite is true, i.e., N_0P_1 is the first path traversed, the two paths/locations may exist in the same tree. The reason for this is that if N_0P_0 is traversed first, the algorithm can reach the leaf of that path and start backtracking without visiting MIEN node N_1, and can start

another branch which contains N_1P_0 (after backtracking). Therefore, the algorithm needs to perform another check to decide if N_0P_1 and N_1P_1 may coexist in the same tree or not. To do this, the algorithm loops on all paths leading to N_1 other than N_1P_0 to check if there is another location possible for N_1 that is closer in the tree to the location where N_0 is branched (N_0P_0) than the location defined by N_1P_0 (line 14). If such path exists, then the two paths N_0P_1 and N_1P_1 cannot be included in the same combination, and therefore N_1P_1 is rejected (line 17) and the algorithm moves to the following path N_1P_2 to examine it.

To summarize this step mathematically,

$I_0 = N_1P_0 \cap N_0P_0$

$I_1 = N_1P_0 \cap N_0P_x$

$l_2 = N_0P_0 \cap N_1P_y$

If there exists a path N_0P_x leading to N_0, where $l_1 > l_0$ and there exists a path N_1P_y leading to N_1, where $l_2 > l_1$, then the path N_1P_0 gets rejected and the algorithm does not add it to the current combination.

This is repeated until each node N_x has a path N_xP_i that gets added to the combination C (line 25).

Now that each MIEN node N_x has a defined location, the combination C represents one depth first traversal tree and it can be added to the list of depths first combinations (line 28).

The algorithm now moves to the trees construction step, which proceeds as follows:

1. To make sure the tree construction works correctly, the parents of each MIEN node are placed in their designated position as calculated by the current combination *C* before attempting to move the descendants in the desired locations. Otherwise, when moving the subtrees rooted at the parent, the locations of the descendants will get reshuffled. Since the path to reach a parent node from the root of the graph must be shorter than any of the paths leading to one of the descendants of that parent, the algorithm uses quick sort (Figure 20: line 6) to sort each combination *C* from the shortest path to the longest path.
2. As mentioned earlier, one depth first initial tree is generated at the beginning. The algorithm then uses this initial tree as the base tree. The initial tree is cloned once for each combination *C* (line 5), then all the MIEN nodes are relocated according to the paths stored in *C* that were generated in the previous steps (lines 8 to 12). This step is repeated for all combinations *C* to get all possible depth first traversal trees (loop at line 7).

VI.3.2 Complexity Analysis

In this section, we analyze the time complexity and the space complexity of the DFS algorithm.

VI.3.2.1 Time Complexity

The complexity of a DFS to get the initial tree is $O(N + E)$ where E is the total number of edges in the graph and N is the total number of nodes. Given that each node is the destination for one edge except for the root node, the complexity of the DFS (initial tree generation) is $O(N+E)=O(E+E) =O(E)$.

The complexity of the quicksort for each combination is $O(N.log(N))$ where N is the number of items to be sorted [58]. In our case, the quicksort algorithm complexity is $O(P.log(P))$ where P is the average number of paths leading to MIEN nodes. The complexity of getting all the possible depth first combinations is $O(P^2)$ where P is the average number of paths leading to a MIEN node. The complexity of switching MIEN nodes, which includes cloning the tree, is $O(T \times (N + N_{mien}))$ where T is the resulting number of trees and N_{mien} is the number of MIEN nodes.

The total Complexity of the algorithm is therefore $O(E) + N_{mien} \times O(P^2) + O(PlogP) + O(T \times (N + N_{mien}))$ …(1)

For the worst-case scenario when all possible trees are depth first $T = P^2$ (all combinations of paths). Therefore, $N_{\text{mien}} \times O(P^2) = O(T \times N_{mien})$. Also, in the worst-case scenario all nodes are MIEN nodes ($N = N_{mien}$). Substituting this in (1), the complexity

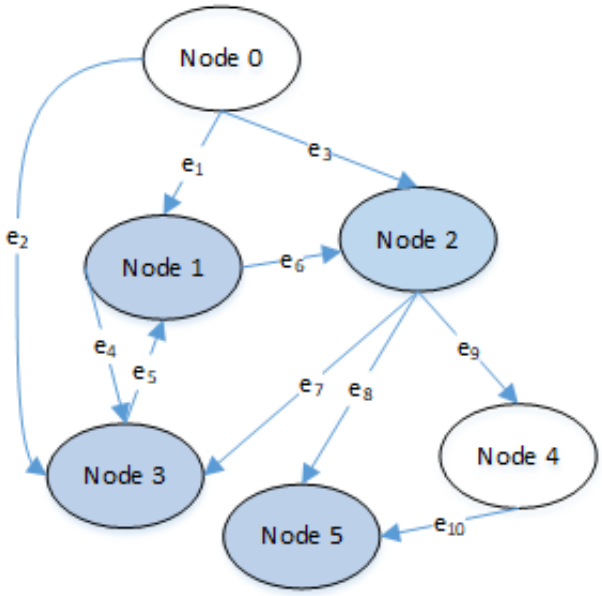


Figure 22 Directed Graph G_2

becomes $O\ (E) + 2 \times N \times O\ (P^2) + O(PlogP) = O\ (E) + O(N \times P^2) + O(PlogP)$. Therefore, the complexity of the finding all DFS trees is $O\ (E) + O\ (N \times P^2)$

VI.3.2.2 Space Complexity

The maximum number of trees generated for the DFS algorithm is all possible spanning trees of the original directed graph. Therefore, what is explained in VI.2.2.2 about the BFS trees space complexity, applies also to the DFS trees.

VI.3.3 Example

Figure 22 shows a directed graph with three non-leaf MIEN nodes (having more than one incoming edge): Node 1, Node 2, and Node 3. Node 5 is a MIEN node, but also a leaf. Table 5 lists all the possible paths leading to each MIEN node in the graph of Figure 22.

Based on the discussion at the beginning of this chapter, the possible number of all kinds of traversal trees is all possible combinations of paths leading to MIEN nodes. This number is the product of the number of paths leading to Node 1, Node 2, and Node 3 which is 3×4×3 = 36. However, not all those trees are depth first traversal trees. The depth algorithm described in section VI.3.1 works for this example as follows:

- Generate one depth first tree using the depth first search algorithm.
- Adds $N_1P_0 = e_1$ as the chosen path leading to Node 1 to the first combination of paths. Combination = $\{N_1P_0\}$.
- Then, it starts examining the paths leading to the next MIEN node, one path at a time. Let us assume it is Node 3. For example, e_2 is examined as follows: $N_3P_0 \cap N_1P_0 = \{e_2\} \cap \{e_1\} = \{\emptyset\}$.

The intersections between e_2 and the other paths leading to Node 1 are then calculated.

$N_3P_0 \cap N_1P_1 = \{e_2\} \cap \{e_2, e_5\} = \{e_2\}$

⇨ $N_3P_0 \cap N_1P_1 > N_3P_0 \cap N_1P_0$, but N_1P_0 is already added to the list of combinations. Therefore, N_3P_0 gets rejected and it does not get added to the current combination.

- Then the following possible path leading to Node 3 is examined: $N_3P_1 = \{e_1, e_4\}$. Similarly, to the previous step, since N_3P_1 does not intersect with any path leading to Node 1 other than N_1P_0, N_3P_1 gets added to the combination. $C = (N_1P_0, N_3P_1)$
- The algorithm now moves to Node 2 and examines the first path N_2P_0 against N_1P_0 and N_3P_1.

  Although the first condition is violated since $N_2P_0 \cap N_3P_1 < N_2P_0 \cap N_3P_3$, it is still possible to add the path because of the second condition: There is no intersection between N_3P_1 and any of the other paths leading to Node 2 that is longer than $N_2P_0 \cap N_3P_1$. Thus, in a DFS tree N_2 cannot be reached through the subtree rooted at the location of Node 3 which is represented by N_3P_1. The traversal can reach this location

Table 5 All Possible Leading Paths to MIEN nodes

MIEN Nodes	Leading paths
Node 1	$N_1P_0 = \{e_1\}$ $N_1P_1 = \{e_2, e_5\}$ $N_1P_2 = \{e_3, e_7, e_5\}$
Node 2	$N_2P_0 = \{e_1, e_6\}$ $N_2P_1 = \{e_2, e_5, e_6\}$ $N_2P_2 = \{e_3\}$
Node 3	$N_3P_0 = \{e_2\}$ $N_3P_1 = \{e_1, e_4\}$ $N_3P_2 = \{e_3, e_7\}$ $N_3P_3 = \{e_1, e_6, e_7\}$

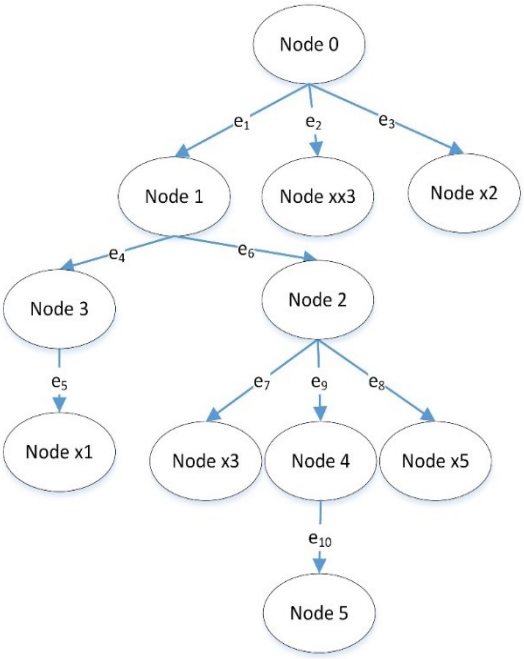


Figure 23 Tree for paths combination *{N1P0, N3P1, N2P0}*

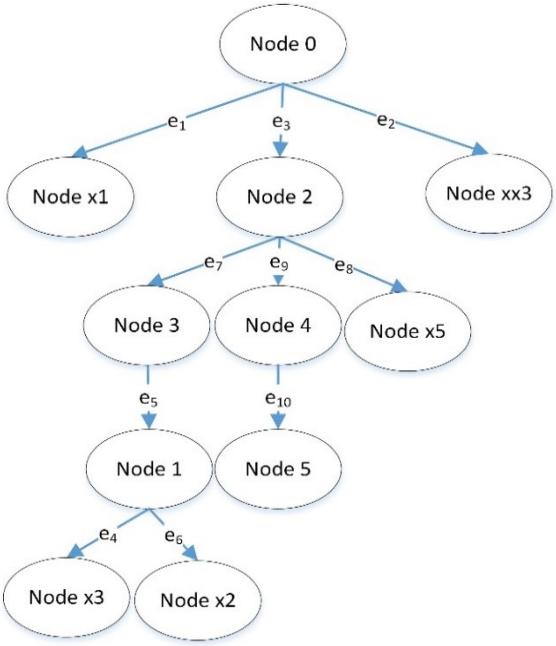


Figure 24 Tree for *{N_1P_2, N_3P_2, N_0P_0}*

of Node 3, backtrack to Node 1, and then branch to reach Node 2 through the path N_2P_0. Therefore, N_2P_0 gets added as a valid path to the current combination. $C = \{N_1P_0, N_3P_1, N_2P_0\}$.

- Now that the size of Combination is equal to the number of MIEN nodes, the current combination is added to the list of possible depth combinations that can later define a DFS tree.
- To create the corresponding tree, the algorithm just switches the locations of the MIEN nodes existing in the depth first tree generated at the first step of the algorithm. There is no need to touch the other nodes belonging to the graph that are not MIEN nodes since nothing changes about those nodes from one tree to the other except if they exist in one of the subtrees rooted at a MIEN node. In this case, such nodes get automatically

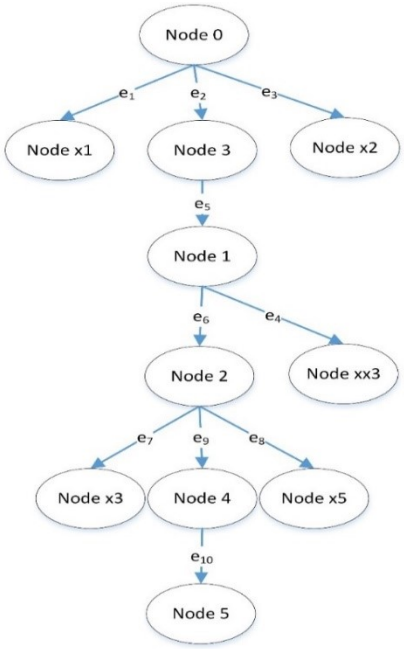


Figure 25 Traversal Tree of Graph G2 for Path *{N_2P_1, N_3P_0, N_4P_1}*

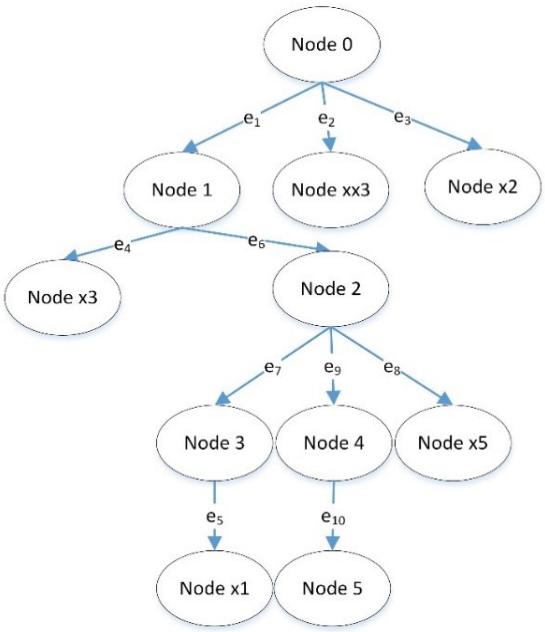


Figure 26 Tree Corresponding to Combination *{N_1P_0, N_3P_3, N_4P_2}*

relocated when shuffling MIEN nodes. The tree resulting from this example is shown in Figure 25.

The algorithm generates four possible depth trees, the first of which is shown in Figure 23 while the rest of the generated trees are shown in Figure 24, Figure 25 and Figure 26.

VI.4 Random Traversal Algorithm

An algorithm is called randomized when its behavior is not only dependent on the inputs provided to it, but also on a random number generator that is independent of the inputs [17]. Since random algorithms are proven to be both simple and efficient [59], a random algorithm for creating benchmark test suites against which the efficiency of the test suites generated by the other three algorithms can be compared.

The main idea of the random graph traversal is that instead of systematically choosing a node to expand from the tree based on a certain criterion, such as the distance from the initial node for instance in the case of a BFS traversal, the algorithm randomly picks any node in the graph to process. The proposed algorithm guarantees that all edges and all nodes are covered in the produced tree. The rationale behind this is to keep all the variables other than the structure of the tree fixed. This way the variance in results is only dependent on the difference in the ways the trees are generated using BFS, DFS, and random traversal and not on the content of the tree itself.

In the experiment, ten unique random trees are generated for each experimental object to get a representative sample of the possibly generated trees. Using only one random tree is not sufficient for statistical analysis. In addition, having one tree might result in a specific BFS or DFS tree which would be much better (or worse) than others and which can mislead the comparison process. On the other hand, we choose not to increase the number of random trees to avoid the possibility of duplicate trees. We tested generating random trees for all our experimental objects and the algorithm produced unique trees.

We validated that the use of ten random trees is reasonable during the results analysis phase, as the results achieved by the random test suites is uniformly distributed which shows that the random sample of test suites is representative of the population of all generated test suites (Chapter VIII). It may be advisable in future work to increase the size of the random trees sample while ensuring the uniqueness of the generated trees for experimental objects that have a high number of possible traversal trees such as the VCR.

```
Algorithm RandomTreeGenerator (G)
Input: A directed graph G.
Output: A spanning tree T of the input graph G.

1   n_g ← GET_INITIAL_NODE(G)
2   ADD (t, n_g)
3   e_out← GET_OUTGOING_EDGES(n_g)
4   ADD (e_toprocess, e_out)
5   do
6      s ← SIZE(e_toprocess)
7      i ← GET_RANDOM(0,s)
8      e ← GET_EDGE(e_toprocess)
9      REMOVE(e_toprocess,e)
10     n_g← GET_DESTINATION_NODE(e_toprocess)
11     if not LEAF(n_g) and not ADDED (t, n_g) then
12       e_out← GET_OUTGOING_EDGES(n_g)
13       ADD (e_toprocess, e_out)
14     Endif
15     if not ADDED (t, n_g)
16       ADJUST_MIEN_NODE_PREFIX(n_g)
17       ADD (T, e, n_g)
18     Endif
19  While not size(e_toprocess) 0
```

Figure 27 Random Algorithm Pseudocode

VI.4.1 The Algorithm

The random algorithm has two main decisions to make. The first one is deciding which edges of the initial graph are possible candidates to be added to the resulting tree, while the second one is randomly selecting one of the candidate edges and adding it to the graph. The random algorithm may result in a tree as deep as one of the DFS trees and may also result in trees with more shorter paths similarly to the BFS trees.

The random algorithm starts by creating a list of unique edges $e_{toprocess}$ that can possibly be added to the resulting tree. Upon triggering the algorithm, a copy of the initial node of the graph is created (Figure 27: line 1) and added to the tree as the tree root node (line 2). Now that the tree has only one node, there is only one way to expand the tree which is by expanding the root. The outgoing edges of the root node are added to the list

of unique edges that are eligible for processing $e_{toprocess}$. Each time a node is added to the tree, the outgoing edges of that node are added to $e_{toprocess}$, which is a FIFO data structure.

One of those edges, e, is randomly chosen with equal probability (lines 7 and 8). The destination node (now n_g) of e is retrieved from graph *G* (line 10). If the retrieved node (new n_g) is neither a leaf node nor has been added to the tree before, it gets added to the tree (line 10), e gets removed from $e_{toprocess}$ and the outgoing edges of n_g get added to $e_{toprocess}$. If the retrieved node has been added before to the tree, the prefix gets adjusted (MIEN node handling) before adding the node to the tree. This guarantees that the algorithm produces trees only and not graphs with cycles. If the current node getting added to the tree is a MIEN node that a copy of which has been added to the tree before, the outgoing edges do not get added to $e_{toprocess}$. Therefore, the tree will never have duplicate edges, and the algorithm has a stopping condition. The algorithm then keeps repeating the process (loop in lines 15 to 19) until it consumes all the edges in $e_{toprocess}$, which means that all the nodes and edges of the input graph have been added to the tree.

VI.4.2 Complexity Analysis

VI.4.2.1 Time Complexity

Theorem: Random algorithm runs in *O(E)* time.

Proof: The algorithm in Figure 27 loops once on the edges of the graph that can be added to the generated spanning trees. The size of the list of edges that can possibly be added to the spanning tree is equal to the number of graph edges at the maximum. Therefore, the time complexity of the random algorithm is *O(E)* in the worst case where *E* is the number of edges (transitions) in the original directed graph.

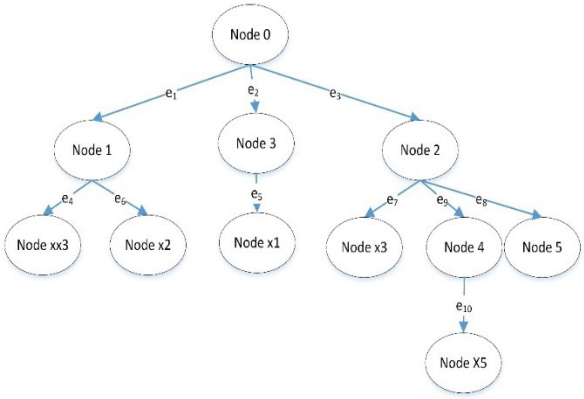


Figure 28 Random Algorithm Run #1

VI.4.2.2 Space Complexity

Theorem: The resulting trees of the random algorithm occupy *O(E)* space.

Proof: All spanning trees or full trees span all the nodes as well as all the edges of the original graph. Therefore, the occupied space for each tree is *N+E*. However, there are stored copies of the MIEN nodes. Given that edges are not repeated in any of the resulting trees, and that each edge has only one destination node, the number of nodes in the resulting trees is always *E+1*, where *E* is the number of edges and the one added for the start node that is not a destination of any of the edges. This means that the number of nodes *N* is equal to *E+1*. Therefore, each tree occupies space of $E+E+1 = 2E+1$. However, the algorithm generated 10 random trees for each graph and hence the total complexity is $20\times(E+1)$. This proves that the space complexity of the generated trees for the random algorithm is equal to *O(E)*.

VI.4.3 Example

Graph G_2 of Figure 22 is used as an illustrative example. Figure 28 and Figure 29. illustrates two different runs of the random algorithm given the directed unweighted graph in Figure 22 as the input graph.

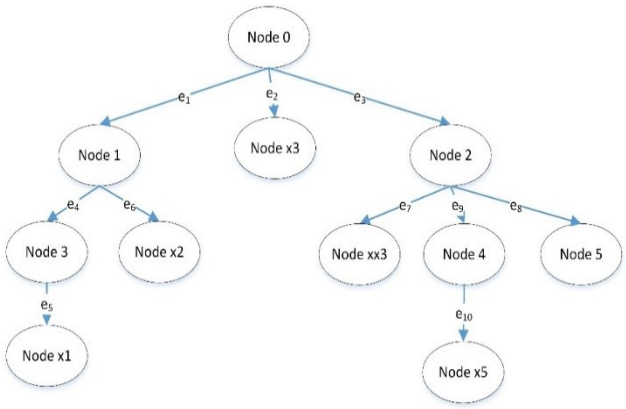


Figure 29 Random Algorithm Run #2

VI.5 Round-trip Algorithm

Unlike the three previously described algorithms, the round-trip test suite generation algorithm used in the experiments is a combination of two algorithms that have been reused from other sources as described in the following section. To generate the round-trip paths test suites, STAGE-1 first calculates all the round-trip paths available in the graph. Then, the prefixes from the root to the nodes starting the round-trip paths are calculated.

VI.5.1 Getting All Round-trip Paths

The algorithm for getting round-trip paths is available online [91]. The algorithm loops on all the nodes in the graph to find all the round trips for each node. Initially, the initial node is considered the current node. The traversal loops on the neighbors of the current node. If the start node is reached for the second time the path is returned as a round-trip path. Otherwise, for each neighbor of the node, the algorithm adds the neighbor to the current path. Then, the round-trip algorithm initiates a recursive call using each neighbor node as the current node, then removes that neighbor node from the current path after returning from the recursive call. This method assures that for the round-trip there is no

[0] Node0-e3-Node2-e7-Node3-e5-Node1-e6-Node2

[1] Node0-e1-Node1-e4-Node3-e5-Node1-e4-Node3

[2] Node0-e1-Node1-e4-Node3-e5-Node1-e6-Node2-e7-Node3

[3] Node0-e1-Node1-e4-Node3-e5-Node1

[4] Node0-e1-Node1-e6-Node2-e7-Node3-e5-Node1

Figure 30 RTP Test Suite for the Directed Graph G_1

repeating node except the first and last since once a node is hit again the algorithm realizes that a round-trip path has been formed. However, when calculating the prefix as shown in the following section minimum node repetition may occur.

VI.5.2 Creating the Prefix

The prefix is the path from the initial node of the graph to the node starting and ending the round-trip path. It is possible to have more than one prefix for each node in one graph, as nodes could be reachable from the initial node through more than one path. The analysis phase of the results is concerned with the contribution of each category of paths to the coverage or the mutation score. Therefore, it has been chosen to use the shortest prefix to minimize the contribution of that prefix to the mutation score as a mean to get more accurate results.

The shortest prefix algorithm has been adapted from an algorithm to find the shortest path between two nodes in a directed unweighted graph [92].

Note that unlike other traversal algorithms we exercise, the RTP algorithm does not exercise all transitions, which is different from what Binder suggests [8]. However, the question that my thesis raises and tries to answer is whether the RTP can find faults that are not detectable by the other traversal algorithms. Therefore, the design of the round-trip

algorithm serves this purpose by focusing only on the faults revealed by exercising the round-trip paths.

VI.5.3 Example

Providing the directed graph of Figure 22 to the RTP algorithm explained above, produces the test suite of Figure 30.

Chapter VII Results

The numbers of unique test suites generated by STAGE-1 and the average number of transitions executed per test suite for each traversal algorithm are shown in Table 6. We discuss next each experimental object separately. For each experimental object, we report the data collected from the experiments. Although we thoroughly analyze the statistics of the collected data in Chapter VIII, in this chapter we report for each of the four main test suite generation strategies (BFS, DFS, random, RTP) some descriptive statistics such as the mean mutation score, standard deviation among test suites, the ratio of mutation score over mutation coverage, mutation coverage as well as average execution time.

VII.1 Cruise Control

As shown in Table 6, STAGE-1 produces three BFS test suites, three DFS test suites, ten random test suites, and one RTP test suite for the cruise control object. Due to the structure of the FSM, the BFS and DFS test suites are identical.

149 (44.35%) mutants are covered by each BFS/DFS test suites. Two of these test suites kill 58 mutants, while one kills 59 mutants (17.3%): Figure 31. The different BFS/DFS and random traversals do not affect mutation score, neither do they affect mutation coverage. Nothing in the structure of the trees can explain the different execution times; we conjecture this is due to caching. The RTP test suite kills 59 mutants out of 149 covered mutants. This gives a mutation score of 17.5% and a coverage ratio of 44.3%. Only one of these 59 mutants are not killed by any other test suite. There is no clear difference

Table 6 Output Matrix of STAGE-1

Experimental object Name		Cruise Control	ATM	Ord. Set	VCR
BF Trees	*No. of Trees Generated (Test Suites)*	3	9	4	24
	No. of Test Cases/ Tree	25	14	27	49
	Avg. No. of Transitions/ Suite	89	53	61	174
	Generation Time (ms)	<1	<1	1	6
DF Trees	*No. of Trees Generated (Test Suites)*	3	27	59	101,947
	No. of Test Cases/ Tree	25	14	27	49
	Avg. No. of Transitions/ Suite	89	61	81.9	293.42
	Generation Time (ms)	4	20	5	156,475 (2mn36s)
Round-Trip	*No. of Test Suites*	1	1	1	1
	No. of Test Cases	47	16	27	1407
	No. of Transitions	225	84	73	9768
	Generation Time (ms)	15	<1	16	798
Random Tree	*No. of Trees Generated (Test Suites)*	10	10	10	10
	No. of Test Cases/ Tree	25	14	27	49
	Avg. No. of Transitions/ Suite	89	58.6	67	201.6
	Generation Time (ms)	<1	<1	<1	31

between BFS, DFS, and RTP, though RTP is roughly twice as expensive as BFS and DFS (Table 6).

Note that when we refer to the number of mutants killed by the RTP, this number includes the faults killed by the prefix as well. However, as mentioned in VI.5.2, the algorithm uses the shortest prefix to minimize the contribution of the prefix to the number of faults detected.

One general observation we can make is the low mutation score of all test suites: 17%. This can be explained by the fact that many of the seeded faults are related to the timing aspect of the code (e.g., speed increases when the driver pushes on the gas pedal), which is not modeled in the FSM: the FSM is an abstraction of a behavior that includes functional properties and timing properties; it abstracts away from timing properties. In

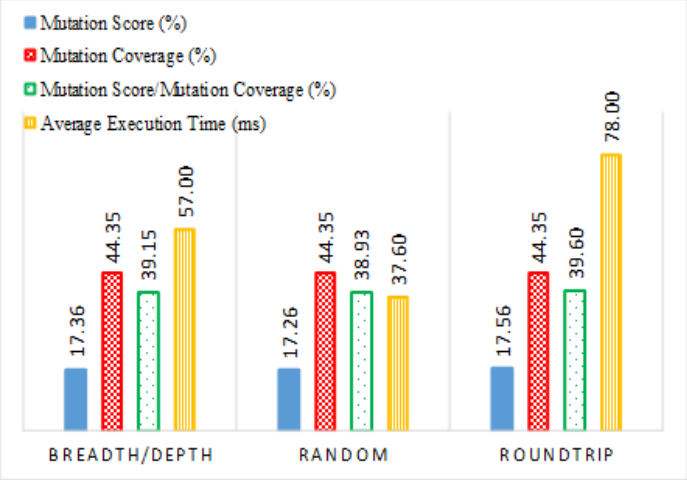


Figure 31 Cruise Control Results

other words, the FSM model is adequate for the parts of the system it simulates and hence it can detect faults related to such modeled aspects. Other aspects such as the speed and timing can be modeled using more relevant techniques (e.g., a Simulink model).

VII.2 ATM

STAGE-1 generates nine BFS test suites, 27 DFS test suites, ten random test suites, and one RTP test suite (Table 6). From Figure 33, we observe that 106 (32.8%) of the

[0] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-10-SelectSAction-13-xxSelectAccount
[1] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-10-SelectSAction-14(15)-ProcessS-21-xSelectSAction
[2] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-10-SelectSAction-15(10)-xProcessS
[3] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-10-SelectSAction-16-xxProcessS
[4] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-11-xQuit
[5] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-12-SelectCAction-17-xxxSelectAccount
[6] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-12-SelectCAction-18(10)-ProcessC-22-xSelectCAction
[7] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-12-SelectCAction-19(5)-xProcessC
[8] Start0-1(PIN 20 30)-Validate1-2(PIN)-SelectAccount-12-SelectCAction-20-xxProcessC
[9] Start0-1(PIN 20 30)-Validate1-3(Wrong1)-Validate2-4(Wrong2)-Validate3-6(Wrong3)-xxxQuit
[10] Start0-1(PIN 20 30)-Validate1-3(Wrong1)-Validate2-4(Wrong2)-Validate3-7(PIN)-xxxxSelectAccount
[11] Start0-1(PIN 20 30)-Validate1-3(Wrong1)-Validate2-5(PIN)-xSelectAccount
[12] Start0-1(PIN 20 30)-Validate1-3(Wrong1)-Validate2-9-xxQuit
[13] Start0-1(PIN 20 30)-Validate1-8-Quit

Figure 32 ATM Round-trip Paths Test Suite

seeded faults are detected by all BFS, DFS, and random test suites. The RTP test suite detects only 83 mutants (25.7%). For coverage, all test suites except for RTP cover 267 (82.6%) mutants, while the RTP test suite covers 230 (71.2%) of the seeded mutants.

The reason for the lower mutation score of the RTP is that the round-trip paths do not exercise all the transitions. For example, the transition between the state Validat2 and SelectAccount state of the ATM FSM (appendix A.2) is not exercised in the RTP test paths shown in Figure 32. Binder's solution to this issue is to recommend the execution of simple paths [8]. Since we are primarily interested in RTPs, we did not include simple paths in our test suites. However, the ratio of mutation score to coverage still shows lower measure for the RTP test suite than other test suites: some mutants are not killed even if they are covered; specific paths in the FSM need to be triggered in order to kill the mutants and those paths are not exercised by the RTP (even with prefixes). We also note that all the mutants killed by the RTP test suite are killed by the other test suites. This is a different observation than the case in cruise control object, which can be explained by looking at the FSMs. There is only one possible shortest prefix to each round-trip path in the cruise control FSM.

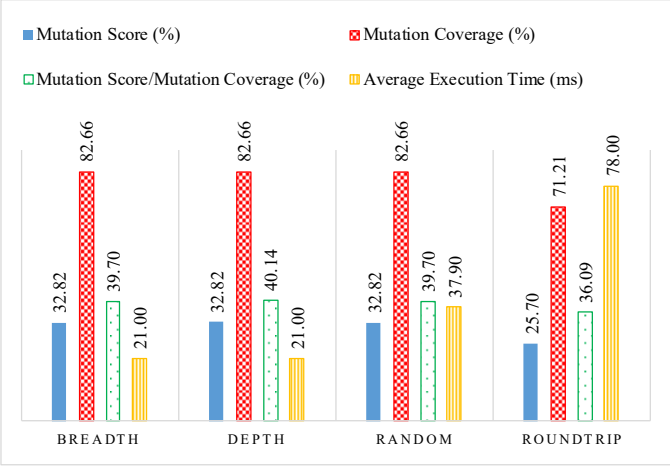


Figure 33 ATM Results

This means that the RTP test suite in the cruise control is not missing any transition by choosing the shortest prefix since there is only one prefix unlike the case with the ATM FSM. In other words, all states except for the initial state are members of at least one round-trip paths. This leads to expecting higher mutation score from the RTP test suite of the cruise control object unlike the case with the ATM. Figure 33 is a summary of the results for the ATM object. As shown in Figure 33, the RTP test suite takes the longest to execute.

VII.3 Ordered-Set

STAGE-1 generates four BFS test suites, 59 DFS test suites, ten random test suites, and one RTP test suite for the ordered set object (Table 6). The mean values of the mutation score, mutation coverage, ratio between mutation score and mutation coverage, and the execution time are shown in percentages in Figure 35. The mutation score for the BFS test suites ranges from 263 to 266 mutants out of 727, while coverage ranges from 443 to 450 mutants. For the DFS test suites, mutation score ranges from 260 to 271 mutants, while coverage is 442 mutants for all test suites. Random test suites mutation scores range from 260 to 271 mutants, while covered mutants are at a minimum of 442 and a maximum of 450 mutants. The mutation score for the RTP test suite is 262 mutants, while the coverage is 442 mutants.

The highest mean mutation score is achieved by the DFS test suites, while the highest coverage is achieved by a random as well as one of the BFS test suites. We note that the test suite that achieves the highest mutation score is not the one that achieves the highest coverage. Since the ordered-set FSM results are more varied than the two previous experimental objects systems, more analysis of the results is needed.

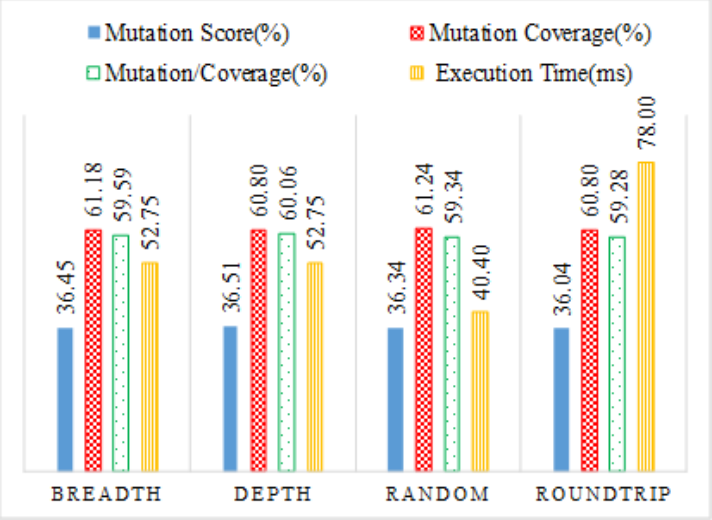


Figure 35 Ordered Set Results

Although the BFS test suites have higher mutation coverage on average than the DFS test suites, Figure 35, the DFS test suites have better mutation score than the BFS test suites. The RTP algorithm has the highest average execution time. The higher execution time of the round -trip is due to more and longer test cases in many cases. The random test suites also have the highest standard deviation in mutation score which is expected as the random nature of the algorithm results in trees that vary in structure (Table 7). The mutation score and standard deviation of DFS test suites are slightly higher than that of BFS test suites. Figure 36 shows how the mutation scores vary among different test suites belonging

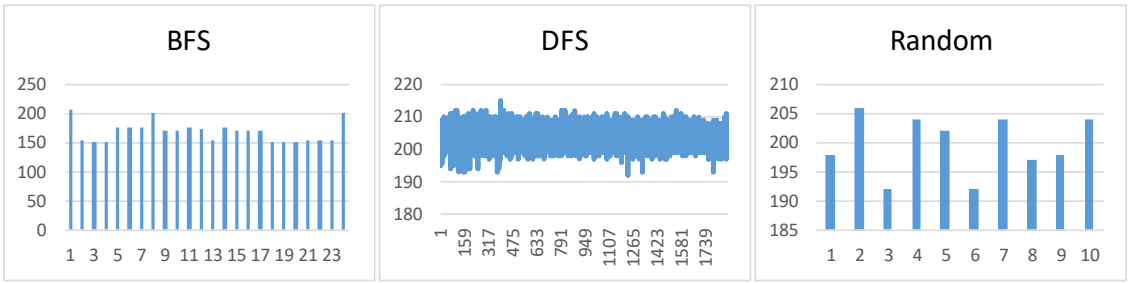


Figure 34 VCR Killed Mutants Comparison Within Each Traversal Method (X-Axis: Test Suite No., Y-Axis: No. of Killed Mutants.

to the same traversal strategy of the ordered set FSM. For example, only two out of the four BFS test suites have the same mutation score.

VII.4 VCR

As listed in Table 6, STAGE-1 generates 24 BFS test suites, 101,947 DFS test suites, 10 random test suites, and one round-trip paths test suite for the VCR object. Given the vast number of DFS test suites produced, the experiment in hand focuses the analysis on 1,887 DFS test suites only. The 1,887 test suites span the complete set of DFS test suites; they are not clustered on the first few thousand for instance and are rather distributed among the whole set of test suites. Approximately, one test suite is picked for each 50 test suites. We tried to maintain a large difference between the structure of the chosen test suites. This guarantees that this sample is representative since the way the DFS algorithm is designed results in the fact that any two highly similar trees differ by only one transition.

Out of the 1649 seeded mutants, the BFS test suites catches between 150 and 206 mutants with an average of 168 mutants. The number of mutants revealed by the DFS test

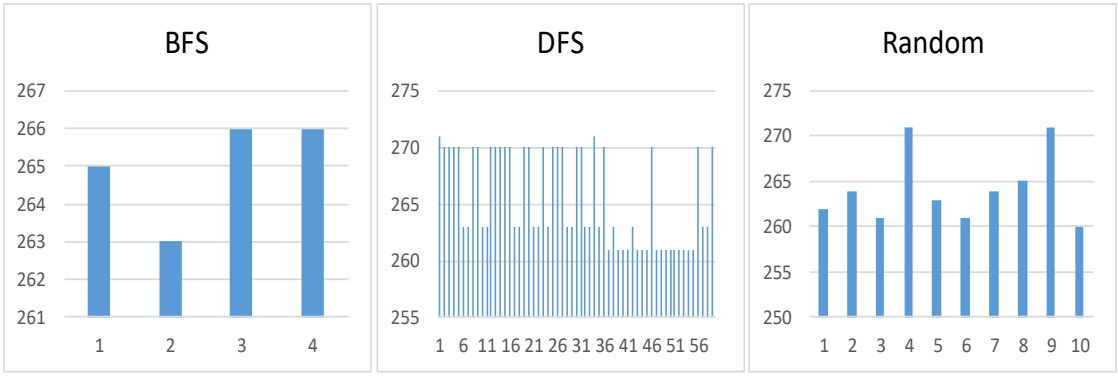


Figure 36 Ordered Set Killed Mutants Comparison within Each Traversal Method (X-Axis: Test Suite No., Y-Axis: No. of Killed Mutants.

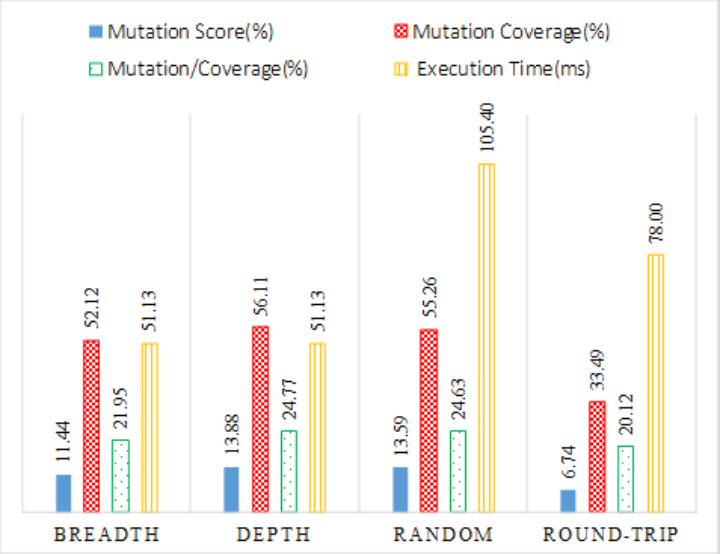


Figure 37 VCR Result

suites is between 192 and 215 mutants, while the random test suites are at a minimum of 192 and a maximum of 206 mutants killed. Finally, the RTP test suite kills 99 mutants.

Figure 37 compares the mean values of the mutation score, coverage, mutation to coverage ratio, and execution time for all the algorithms. Test suites generated using the DFS algorithm have the highest mean mutation score (13.8%), directly followed by the random algorithm (13.5%), then the BFS (11.4) and finally the lowest mutation score is for the RTP test suite (6.7%). The coverage follows the same order as the mutation score as shown in Figure 37. The random test suites are on average the slowest followed by the RTP, and then the BFS and DFS test suites are the fastest: Figure 37.

Figure 34 visualizes the comparison between the different test suites in terms of mutation score. It illustrates how the mutation score varies between the different test suites, even if those test suites are generated using the same traversal algorithm (i.e., BFS, DFS, or Random). Although the DFS chart of Figure 34 is crowded due to the number of

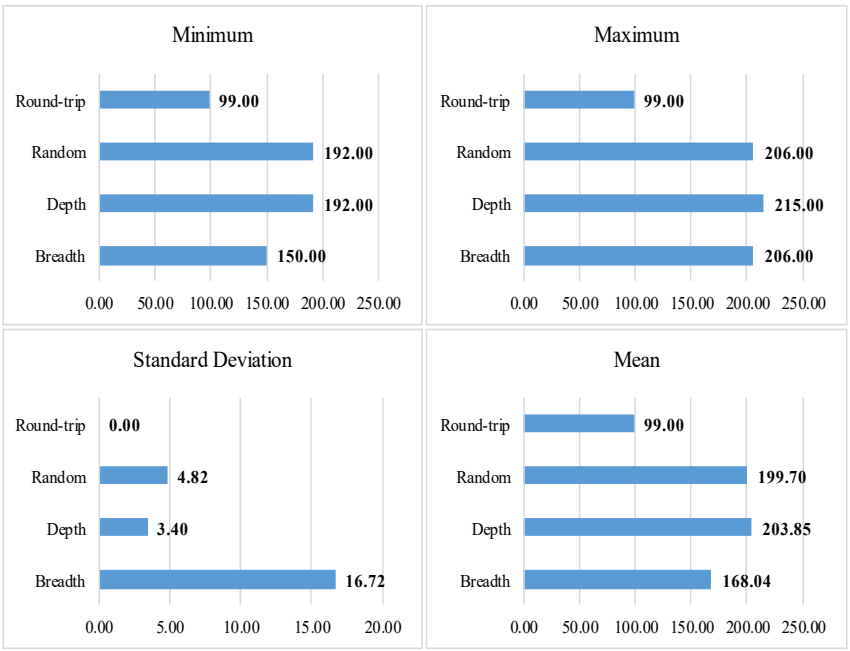


Figure 38 VCR No. of Killed Mutants Statistics

exercised test suites (1887), it is clear from the height of the bar that the mutation score varies between 119 and 215 killed mutants (see also Figure 38).

It is observed from the experiment that 27 mutants are killed only by the RTP test suite and the DFS test suites, but not killed by any other test suite. In addition, 18 mutants are killed only by the RTP test suite.

Chapter VIII Results Analysis

After collecting data from the operation phase, interpreting the data is paramount to be able to draw conclusions. In the previous chapter (Chapter VII) we provided results for each experimental object separately and included some descriptive analysis of the results. In this chapter, we provide a quantitative analysis of the results by wrapping up the descriptive analysis provided earlier and use inferential statistics of the achieved results (VIII.1). We then focus on qualitative analysis by studying the different categories of mutants, and the relationship between the different mutant operators and the achieved results (section VIII.2.2). Last, we provide a structured discussion to answer the research questions (section VIII.3).

VIII.1 Statistical Analysis

In this section, we provide a quantitative analysis of the collected data in the form of both descriptive and inferential statistics. While the descriptive analysis describes the relationship between variables in a sample population (section VIII.1.1), inferential statistics tries to use a sample of the population to make inference about the whole population (section VIII.3) [84].

VIII.1.1 Descriptive Statistics

The first step of analyzing the collected data deals with the descriptive analysis of the data set in order to better understand the nature of the data [86]. By providing a set of single number descriptive statistics of the data (mean, median, and standard deviation), one can form an impression of the nature of the data sample [84]. Table 7 collects all the

Table 7 Mutation Score Results Summary for All Experimental objects

Measurement \ Algorithm		BFS	DFS	Random	Round-trip
Mean Mutation Score	Cruise Control	58.3	58.3	58	59.0
	ATM	106.0	106.0	106.0	83.0
	Ordered Set	265.0	268.7	264.2	262.0
	VCR	168.0	203.9	199.7	99.0
Min. Mutation Score	Cruise Control	58	58	58	59
	ATM	106	106	106	83
	Ordered Set	263	261	260	262
	VCR	150	192	192	99
Max. Mutation Score	Cruise Control	59	59	58	59
	ATM	106	106	106	83
	Ordered Set	266	271	271	262
	VCR	206	215	206	99
Mutation Score Standard Deviation	Cruise Control	0.1	0.1	0.0	0.0
	ATM	0.0	0.0	0.0	0.0
	Ordered Set	1.2	1.3	3.7	0.0
	VCR	16.7	3.4	4.8	0.0

mutation score statistical data of the four experimental objects. Maximum values among all algorithms for mutation score, mean, minimum, and maximum scores are underlined for each experimental object.

Although DFS test suites produce lower minimum mutation scores in the case of the ordered set experimental object, there is more than one reason to consider this piece of data an outlier that does not contradict the previous general observation we made. First, the difference is only two mutants out of 263 and it is in only one DFS tree. Second, the mutation score does not vary considerably for DFS test suites. This means that in most cases the average mutation score is a valid measure for the mutation score of the DFS test suites. This is especially evident in the two most significant experimental objects: the ordered set and the VCR.

We use box plots as a useful way to show the relative performance of different algorithms when dealing with many artifacts [4]. Figure 39 contains four box plots that visualize the data provided in Table 7 for each experimental object. The figure shows that the data is not uniformly distributed since the four quadrants for the different testing methods are not equal, that there are no outliers in the data and that there is some skewness in parts of the collected data. For the ordered set BFS test suites, the median mutation score achieved is higher than the mean number of killed mutants for all BFS test suites. To the contrary, the median number of killed mutants for the DFS test suites is lower than the mean number of killed mutants. While the data shows that BFS test suites are more consistent in detecting faults (shown by the small height of the box), the average mutation score for the DFS is higher.

The larger variation of fault detection scores for the ordered set, as opposed to the other objects, is likely due to the fact that the ordered set methods accept parameters and

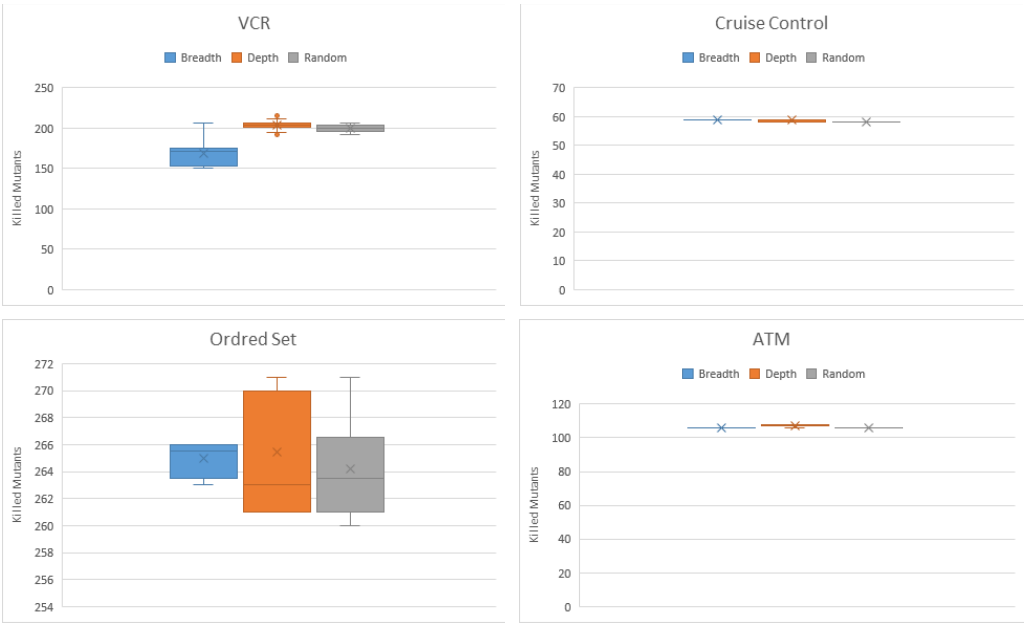


Figure 39 Box Plots of Mutation Score Vs. Testing Methods

the choice of parameter values affects which parts of the code are exercised (e.g., inserting an integer at the beginning, middle or at the end of the ordered data structure triggers different control flows). On the other hand, the other objects are purely event-driven. In our choice of parameters for the different test suites, we tried to be consistent to minimize the effect of the chosen parameters on the effectiveness and to avoid variation on factors affecting the mutation score other than the type of the testing method used.

We also note, 25% of the DFS test suites killed more than 270 mutants (upper quartile of the DFS box in Figure 39). The maximum killed mutants achieved by the DFS is higher than that of the BFS test suites and the median mutation score achieved by the BFS is almost equal to the average mutation score of the DFS test suites and higher than the DFS test suites median. DFS test suites that achieve higher mutation scores than the median have more variations among them in terms of mutation score (more box area above the median line). The random test suites for the ordered set have a distribution closer to the uniform distribution which makes sense since they are composed of different types of test suites.

For the VCR, there are two outliers in the DFS data. However, the outliers are mild outliers with values very close to the maximum and minimum values achieved by the rest of the data [85]. Figure 39 shows that most DFS test suites are either as effective or more effective than the top 25% of BFS test suites (in terms of mutation score). The VCR data is consistent, to some extent, with the data of the ordered set. For example, the BFS test suites mutation score is not uniformly distributed and the distribution of the data among the second, third and fourth quartiles is the same as well as the relevant position of the median and the average values. However, for the first quartile there is more variety of data.

Also, as observed for the ordered set random collection of test suites, the distribution of their mutation scores is closer to the normal distribution than both the BFS and DFS.

To sum up, the ATM and the Cruise control test suites achieve very similar results, while the VCR and the ordered set have more test suites with more variety in mutation scores which makes the data collected from these two experimental objects interesting to analyze further. From the descriptive data provided in Table 7 and Figure 39, we find that although in the ordered set the BFS test suites are more consistent, in all cases the test suites that achieve the highest mutation score are a DFS test suite and in general there is a bigger percentage of DFS test suites that reach high mutation score.

VIII.1.2 Inferential Statistics

In this section, we perform different types of inferential statistics of the collected data. There is a general null hypothesis that we formulate for all four case studies:

H_0: The different methods (BFS, DFS Random) used to generate the test suites does not affect the number of killed mutants.

An ANOVA (Analysis of Variance) test can be employed to test a null hypothesis to test the variation among and between more than two groups. The often-used t-test, on the other hand, compares only two sample means [86]. However, ANOVA, as well as the t-test, are parametric tests that assume the normal distribution of data [4]. Most of our data is not uniformly distributed, but one can argue that with the number of data points that we have and the robustness of the ANOVA test [35], we can assume the amenability of our data to the use of such tests. There are alternative non-parametric tests, but it is known that parametric tests are more powerful [4]. To accommodate for this problem, we choose to run both types of tests on our data and then interpret the results. In this section, we use the

parametric t-test and ANOVA as well as the non-parametric alternatives Mann-Whitney (U-test) [52] and Kruskal-Wallis tests [50].

ANOVA test measures whether the variance between test suites belonging to one group is more significant than the difference between the groups (BFS, DFS, and Random in our case). We perform an ANOVA test on the results obtained from each experimental object to evaluate whether the method chosen (DFS, BFS, or Random) to generate the test suites is predicted to affect the effectiveness of the test suites for systems other than the experimental objects used in our study. Figure 40 shows the results of the one-way ANOVA tests for the four objects. For all objects except the ordered set, the ρ value is less

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Breadth	4	1060	265	2
Depth	59	15662	265.4576	16.59731
Random	10	2642	264.2	15.28889

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	13.81073	2	6.905363	0.436952	0.647752	3.127676
Within Groups	1106.244	70	15.80349			
Total	1120.055	72				

Ordered Set

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Breadth	9	954	106	0
Depth	27	2894	107.1852	0.541311
Random	10	1060	106	0

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	15.66506	2	7.832528	23.93043	1.03388E-07	3.21448
Within Groups	14.07407	43	0.327304			
Total	29.73913	45				

ATM

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Breadth	24	4033	168.0417	291.7808
Depth	1887	384672	203.8537	11.58199
Random	10	1997	199.7	25.78889

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	30530.02	2	15265.01	1017.077	7.7E-302	3.000416
Within Groups	28786.69	1918	15.0087			
Total	59316.71	1920				

VCR

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Breadth	3	177	59	0
Depth	3	175	58.33333	1.333333
Random	10	580	58	0

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	2.333333	2	1.166667	5.6875	0.016807	3.805565
Within Groups	2.666667	13	0.205128			
Total	5	15				

Cruise Control

Figure 40 One-way ANOVA Killed Mutants Vs. Methods.

Ordered Set						VCR					
	BFS	DFS	Random		Total		BFS	DFS	Random		Total
Rank Sum R	131	2172	398			Rank Sum R	43227.5	1788868	13986		
group size	4	59	10		73	group size	24	1887	10		1921
Rsquare/n	4290.25	79959.05	15840.4		100089.7	Rsquare/n	77859032	1.7E+09	19560820		1.79E+09
H					6.33921	H					68.31907
df					2	df					2
p					0.04202	p					0
α					0.05	α					0.05
sig					YES	sig					YES

Figure 41 Kruskal-Wallis Test for Ordered Set and VCR

than 0.05 which is the traditional threshold α used to determine if the null hypothesis is rejected [4]. The ρ value denotes the probability of rejecting the null hypothesis while it is true. However, we have to note here that the underlying data does not meet the uniform distribution condition, and this may cause the higher possibility of a misleading ρ value of the ordered set. The rest of the results of the ANOVA tests suggests rejecting the null hypothesis H_0 in favor of the alternative hypothesis that suggests the different ways of producing test suites actually affect the number of killed mutants.

As mentioned earlier, the Kruskal-Wallis test is a nonparametric alternative to the ANOVA test. We had calculated the Kruskal-Wallis test for the ordered set and the VCR following the procedure described by Zaiontz [87]. In Figure 41, the ρ value is less than 0.05 which means that null hypothesis H_0 is rejected and that the different testing strategies make a difference in the number of killed mutants. Thus, both the parametric and the non-

t-Test: Two-Sample Assuming Unequal Variances								
	Breadth	*Depth*		*Breadth*	*Random*		*Depth*	*Random*
Mean	265	265.4576	Mean	265	264.2	Mean	265.4576	264.2
Variance	2	16.59731	Variance	2	15.28889	Variance	16.59731	15.28889
Observations	4	59	Observations	4	10	Observations	59	10
Hypothesized Mean Difference	0		Hypothesized Mean Difference	0		Hypothesized Mean Difference	0	
df	7		df	12		df	13	
t Stat	-0.51773		t Stat	0.561644		t Stat	0.934735	
P(T<=t) one-tail	0.310302		P(T<=t) one-tail	0.292348		P(T<=t) one-tail	0.183482	
t Critical one-tail	1.894579		t Critical one-tail	1.782288		t Critical one-tail	1.770933	
P(T<=t) two-tail	0.620604		P(T<=t) two-tail	0.584696		P(T<=t) two-tail	0.366963	
t Critical two-tail	2.364624		t Critical two-tail	2.178813		t Critical two-tail	2.160369	

Figure 42 Ordered Set Pairwise t-test

t-Test: Two-Sample Assuming Unequal Variances									
	Breadth	*Depth*		*Breadth*	*Random*		*Depth*	*Random*
Mean	168.0417	203.8537	Mean	168.0417	199.7	Mean	203.8537	199.7
Variance	291.7808	11.58199	Variance	291.7808	25.78889	Variance	11.58199	25.78889
Observations	24	1887	Observations	24	10	Observations	1887	10
Hypothesized Mean Difference	0		Hypothesized Mean Difference	0		Hypothesized Mean Difference	0	
df	23		df	30		df	9	
t Stat	-10.2683		t Stat	-8.24692		t Stat	2.583488	
P(T<=t) one-tail	2.31E-10		P(T<=t) one-tail	1.66E-09		P(T<=t) one-tail	0.014764	
t Critical one-tail	1.713872		t Critical one-tail	1.697261		t Critical one-tail	1.833113	
P(T<=t) two-tail	4.63E-10		P(T<=t) two-tail	3.32E-09		P(T<=t) two-tail	0.029527	
t Critical two-tail	2.068658		t Critical two-tail	2.042272		t Critical two-tail	2.262157	

Figure 43 VCR Pairwise t-test

parametric tests suggest that the method used to generate the transition trees test suites affect the number of killed mutants.

Now that we identified there is a significant difference between the three testing methods (BFS, DFS, and Random) for three of the four objects using the one-way ANOVA (Figure 40) and by the two objects analyzed using Kruskal-Wallis (Figure 41), we need to run three pairwise tests between every pair of testing methods to measure the effect size in each case.

Table 8 Categories of Mutants

Object	Cruise Control				ATM				Ord. Set				VCR			
Seeded mutants	336				323				727				1469			
Total covered mutants	159				267				450				987			
Covered by method	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random
	149	149	149	149	267	267	230	267	450	442	442	450	890	894	492	890
Covered by method but alive	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random
	90	90	90	91	161	159	161	148	182	173	180	174	684	4	391	682
Killed by method	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random
	59	59	59	58	106	108	83	106	268	269	262	276	206	890	99	208
Test equivalent by method	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random	BFS	DFS	RTP	Random
	277	277	277	278	217	215	240	217	459	458	465	451	1443	758	1548	967
Covered but alive	88				159				169				81			
Considered equivalent	265				215				446				743			

We ran the parametric t-test as well as the non-parametric t-test for the ordered set and the VCR. The pairwise t-test for the ordered set (Figure 42) does not reject the null hypothesis, while the VCR pairwise t-test (Figure 43) rejects the null hypothesis. The same results are obtained from the U-test (Figure 44 and Figure 45). However, VCR has more statistical power as an experimental object due to the very large number of test suites compared to the other experimental objects.

VIII.2 Qualitative Analysis:

In this section of the thesis, we examine the categories of mutants (seeded, covered, killed and equivalent) and how mutants of each category differ in relation to the experimental objects on the one hand and the testing strategy used (BFS, DFS, random, or RTP) on the other hand (sectionVIII.2.1). We also perform a qualitative analysis by studying the relationship between the different mutation operators and the testing strategy as well as the used experimental objects (VIII.2.2).

	Breadth	Depth			Depth	Random			Breadth	Random
count	4	59		count	59	10		count	4	10
median	265.5	263		median	263	263.5		median	265.5	263.5
rank sum	137	1879		rank sum	2085	330		rank sum	38	67
U	109	127		U	275	315		U	12	28
	one tail	two tail			one tail	two tail			one tail	two tail
alpha	0.05			alpha	0.05			alpha	0.05	
U	109			U	275			U	12	
mean	118			mean	295			mean	20	
std dev	33.80058	ties		std dev	56.40143	ties		std dev	7.024291	ties
z-score	0.266268			z-score	0.354601			z-score	1.138905	
effect r	0.033547			effect r	0.042689			effect r	0.304385	
U-crit	62.40299	51.75208		U-crit	202.2279	184.4552		U-crit	8.44607	6.232643
p-value	0.395017	0.790033		p-value	0.361444	0.722889		p-value	0.127371	0.254743
sig (norm)	no	no		sig (norm)	no	no		sig (norm)	no	no

Figure 44 Ordered Set Pairwise U-test

	Breadth	Depth
count	24	1887
median	171	204
rank sum	2880	1824036
U	42708	2580
	one tail	two tail
alpha	0.05	
U	2580	
mean	22644	
std dev	2675.092	ties
z-score	7.500304	
effect r	0.171573	
U-crit	18243.87	17400.92
p-value	3.19E-14	6.37E-14
sig (norm)	yes	yes

	Depth	Random
count	1887	10
median	204	200
rank sum	1795239	5014.5
U	4959.5	13910.5
	one tail	two tail
alpha	0.05	
U	4959.5	
mean	9435	
std dev	1720.211	ties
z-score	2.601716	
effect r	0.059735	
U-crit	6605.505	6063.449
p-value	0.004638	0.009276
sig (norm)	yes	yes

	Breadth	Random
count	24	10
median	171	200
rank sum	320.5	274.5
U	219.5	20.5
	one tail	two tail
alpha	0.05	
U	20.5	
mean	120	
std dev	26.29329	ties
z-score	3.784236	
effect r	0.648991	
U-crit	76.75139	68.46611
p-value	7.71E-05	0.000154
sig (norm)	yes	yes

Figure 45 VCR Pairwise U-test

VIII.2.1 Mutants Categories

In this section results for different categories of mutants are listed and the statistics related to them are discussed. Table 8 shows the different categories of mutants for each experimental object:

1. Seeded mutants: Mutants that are seeded in each SUT by Major.
2. Total covered mutants: This number is the union of all the mutants covered by all test suites. Recall that a covered mutant is a mutant which statement of insertion is executed. Note that covered mutants are not necessarily killed.
3. Covered by method: This row breaks down the number of covered mutants mentioned in item 2 above by test suite generation method. For example, BFS trees as a group in the cruise control system cover in total 149 mutants (Table 8). This does not necessarily mean that each BFS tree/test suite

covers all the 149 mutants, but each of the 149 mutants has been covered by at least one BFS test suite.

4. Covered by method but alive: This number shows how many mutants are covered by the group of test suites belonging to one test method (e.g. BFS, DFS, Random, or RTP), but not killed by any of the test suites belonging to this method. Similar to item 3 above, covered but alive mutants are not necessarily covered by each test suite in each method.

5. Killed by method: These are mutants that have been covered and killed by at least one test suite belonging to one method.

6. Test equivalent for method: This is the number of covered mutants that are not killed by any of the test suites belonging to one method. These mutants are considered equivalent with respect to the method they are listed under. However, there are not necessarily equivalent mutants for all test suites (see section V.2)

7. Covered but alive: This is the union of the mutants mentioned in item 4 above. These mutants are considered equivalent mutants since they are not killed by any test suite.

8. Considered equivalent: These are mutants that are not covered as well as mutants that are covered but alive. These mutants are not useful for comparing different types of test suites. None of the test suites managed to kill those mutants, or in other words, those mutants are test-equivalent for all test suites. Therefore, we exclude them from the comparison. Covered

and alive mutants in addition to not covered mutants are considered equivalent mutants (section V.2).

Although the total number of killed mutants by the entire set of DFS test suites is remarkably higher than the other three methods (BFS, RTP, and random) for the VCR, this does not indicate that the DFS is stronger, since not all test suites kill all the mentioned mutants. Some mutants are killed by a small subset of the DFS test suites. Only 204 mutants out of the 890 mutants killed by the set of DFS test suites (underlined in Table 8) are killed by at least 1362 test suites; one mutant is killed by 760 DFS test suites only; three mutants are killed by 759 DFS test suites, and the rest of the mutants are killed by twelve or less DFS test suites. Figure 46 is a histogram that clarifies the distribution of the frequency by which a mutant is killed (Y-axis). On the X-axis we have bins of size 20 that shows the number of mutants. We see that more than 650 mutants fall in the first bin. In other words, more than 650 mutants are killed by less than twenty test suites.

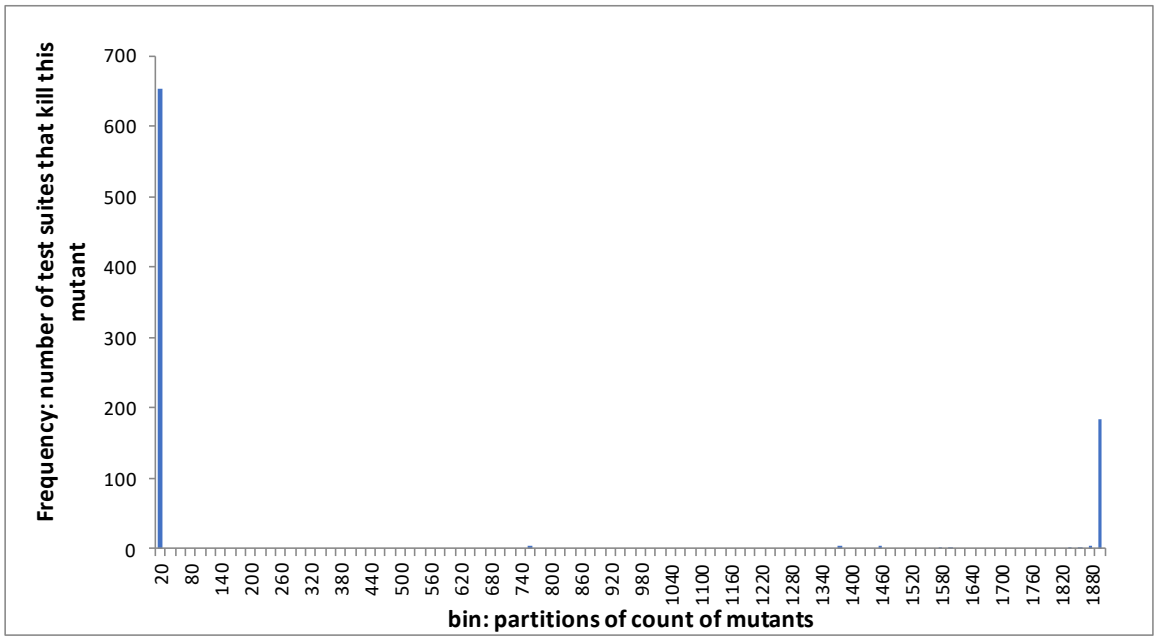


Figure 46 VCR Killed Mutants Histogram

Therefore, the high number of mutants killed by the union of all the DFS test suites does not reflect a strength of the DFS trees as a testing method. However, this observation means that the large number of test suites generated must have some variety between them that causes different test suites to detect mutants that are not detectable by the rest of the test suites in the same group. We conclude that measuring the average mutation score overall test suites belonging to one category is a more precise measurement than the total number of killed mutants. It is also important to mention that we looked closely at the mutants that have been killed by twelve twenty or fewer test suites (first bin on the X-axis of Figure 46). We can call this category of mutants the hard to kill mutants. We consolidated the data and found out that there is no single test suite that performed considerably better than the rest at killing the hard to kill mutants. For example, any single VCR DFS test suite killed a maximum of eleven hard to kill mutants.

VIII.2.2 Mutation Operators

Table 9 categorizes the number of mutants and the percentage of killed mutants by operator for each type of test suites (BFS, DFS, Random, and RTP). The percentage of killed mutants by each type of test suites is the collection of mutants killed by at least one test suite belonging to this type. Some operators listed in Table 4 (page 41) were not applicable for the code and therefore were not inserted by Major: Table 9 shows the operators Major was able to use given the specifics of the code of the subjects.

We observe from the table that the different types of test suites do not perform considerably differently for any operator except for the VCR DFS; similarly to the discussion of section VIII.2.1, we attribute this to the much larger number of DFS test suites for VCR.

We also observe from Table 9 that for all objects except cruise control, all test suites performed best on EVR mutants. In the cruise control, the COR mutants ranked top in terms of being detected. On the one hand, Major inserted 4 COR mutants and all of them are in a conditional statement that evaluates the state of the SUT. However, two of them did not affect the destination state of the object and hence were not detected by any of the algorithms. On the other hand, the EVR mutants that replace statements not affecting the state of the system were not killed. For example, if the cruise control is in the state cruising and the EVR mutant changes the speed invariant (Table 2, page 32), but does not set it to zero, then the state of the system does not change. Therefore, the mutant does not get killed

Table 9 Mutation Scores for Each Mutant Operator

Experimental Object	Mutant		AOR	COR	ROR	STD	LVR	EVR
Cruise Control	Inserted		68	4	86	70	97	11
	Total Killed		1.47%	50.00%	18.60%	27.14%	18.56%	27.27%
	Killed by	BFS	0.00%	50.00%	18.60%	27.14%	19.59%	27.27%
		DFS	1.47%	50.00%	18.60%	27.14%	18.56%	27.27%
		Random	0.00%	50.00%	18.60%	27.14%	18.56%	27.27%
		RTP	0.00%	50.00%	18.60%	27.14%	19.59%	27.27%
ATM	Inserted		16	12	17	205	42	27
	Total Killed		50.00%	58.33%	35.29%	26.83%	52.38%	70.37%
	Killed by	BFS	50.00%	58.33%	35.29%	26.83%	26.19%	70.37%
		DFS	50.00%	58.33%	35.29%	26.83%	26.19%	70.37%
		Random	50.00%	58.33%	35.29%	26.83%	26.19%	70.37%
		RTP	50.00%	33.33%	11.76%	22.44%	19.05%	55.56%
Ordered Set	Inserted		180	94	171	99	166	17
	Total Killed		42.78%	40.43%	35.09%	30.30%	40.36%	52.94%
	Killed by	BFS	40.56%	38.30%	33.33%	29.29%	38.55%	52.94%
		DFS	40.00%	40.43%	33.33%	29.29%	38.55%	52.94%
		Random	42.22%	38.30%	34.50%	30.30%	39.76%	52.94%
		RTP	38.89%	37.23%	33.33%	28.28%	37.95%	52.94%
VCR	Inserted		264	132	242	498	192	141
	Total Killed		56.82%	50.00%	67.77%	65.26%	50.00%	63.83%
	Killed by	BFS	0.76%	12.12%	15.29%	20.08%	3.65%	31.21%
		DFS	56.82%	48.48%	67.36%	60.44%	48.96%	60.99%
		Random	1.14%	12.12%	15.29%	20.08%	4.17%	31.21%
		RTP	0.76%	9.09%	7.02%	8.43%	3.13%	14.18%

since it does not represent a state fault. On the other hand, the EVR mutants that affect the state of the system are killed by all test suites. Due to the nature of the cruise control system, most of the EVR mutants do not affect the state of the object, which is not the case for other experimental objects. For the ATM, for example, the EVR mutants affect the state of the object by changing the PIN or changing the number of attempts to enter a valid PIN (both variables are part of the state invariants as shown in Table 2, page 32). Therefore, all test suites were more successful at finding EVR faults in the ATM.

We also notice a very low mutation score for the AOR faults in the cruise control, compared to other objects. This is due to the fact that almost all the arithmetic operations in the cruise control were related to the timing aspect of the system that was not reflected in the FSM model (the FSM is an abstraction).

Figure 47 is a plot of the percentage of killed mutants for each operator by each algorithm divided by the total number of killed mutants for this operator. We chose to

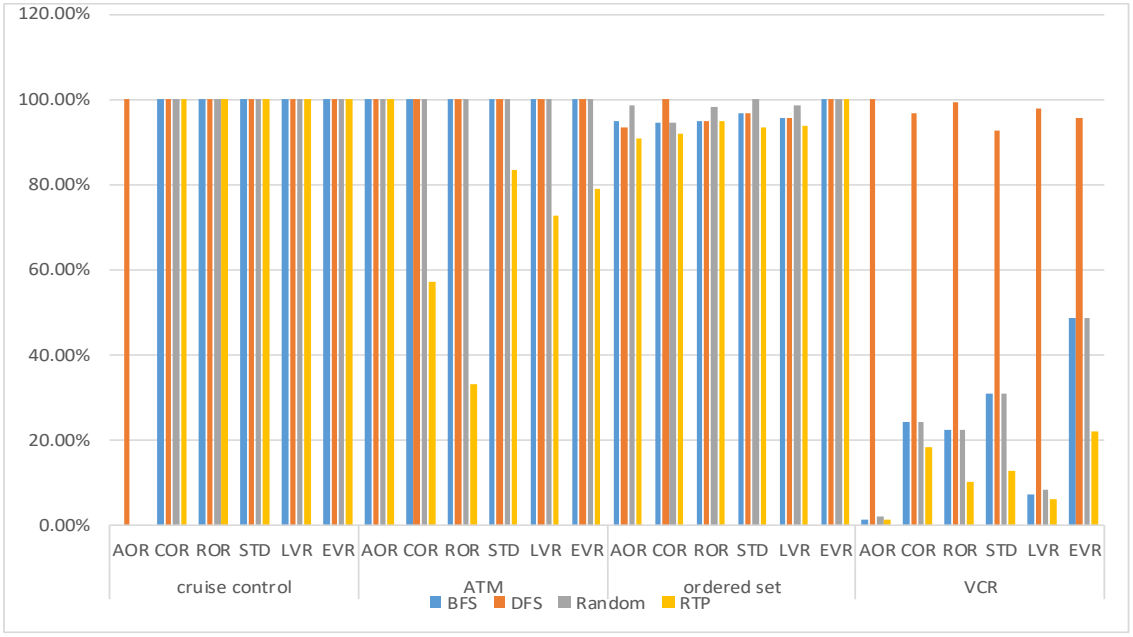


Figure 47 Algorithms Performance for Each Mutant Operator

divide by the total number of killed mutants instead of the mutation score because we want to compare the performance of the algorithms relative to the operators rather than measuring the effectiveness of the test suites, and this measure gives a better visualization of this piece of data. Figure 47 shows some interesting results:

1. All algorithms performed similarly for all operators for the cruise control and the ATM, except for two cases.
    A. First, the ATM RTP test suites kill fewer mutants than other algorithms for all operators except the AOR. This is explained by the way we construct the RTP test suites and the structure of the ATM FSM (appendix A.2). RTP test suites include all complete RTPs in addition to the shortest prefixes leading to the RTPs. As mentioned earlier (section VII.2), in our study we are interested in examining the complete RTPs capability of finding faults that are not detectable by other algorithms (Chapter III: RQ1). Therefore, although Binder in his work adds simple paths in constructing an RTP test suite, to ensure that the criterion subsumes All-transitions by construction, we do not include them to be able to answer our research questions. Note also that as mentioned earlier, Binder's algorithm does not necessarily cover all complete RTP paths in the FSM of the SUT in a complete manner (Section II.1). Due to the fact that the RTP test suite covers only part of the ATM FSM (appendix A.2), due to the structure of the ATM FSM, the RTPs together with their prefixes for the ATM test suite has all the arithmetic operators related to calculating balance, amount to be withdrawn, etc. To the contrary, most relational operators are mutated in the part of the FSM

that is missed by the RTP test suite: For example, the path [start, validate 1, validate 2, validate 3, Quit] is not included in the RTP test suite as it does not include any RTP. Any mutant that is seeded in the part of this path [validate 2, validate 3, Quit] will not be detected by the RTP test suite as it is neither part of a complete RTP, nor part of the shortest path leading to a complete RTP. This path, for instance, uses the equal operator to compare the PIN entered to the actual PIN several times.

B. Second, the DFS of the cruise control is the best performer for the AOR. The 100% killed mutants achieved by the DFS algorithm for the cruise control cannot be used to deduce or generalize results since there is only one mutant killed by the DFS and this is the only AOR mutant killed by all test suites.

2. Although the number of DFS test suites is higher for the ordered set, both groups in most cases perform the same. However, the collection of random algorithms performs better. This is possibly due to the fact the collection of random test suites includes test cases from both BFS as well as DFS test suites.
3. Although as explained above, RTP test suites do not cover all transitions, for the VCR it performs relatively well for all operators when compared to the other algorithms (VCR DFS excluded for reasons explained in section VIII.2.1). This is due to the high connectivity of the VCR FSM. This comment also applies to the cruise control where the RTP achieves the same results as other algorithms.

To conclude, the effectiveness of test suites is not related to the operator used in an absolute manner. However, depending on the nature of the SUT, there might be some

higher probability of detecting a specific type of mutant. In addition, since Major does not report a failure of a test case unless the state of the SUT has been affected, and this is visible when checking a state invariant, mutant operators that do not directly cause a failure by changing the state of the SUT, and resulting in a violated state invariant, are less likely to be killed.

VIII.3 Results Conclusion

This section provides a summary of the experimental results for all the experimental objects to answer the research questions proposed in Chapter III.

VIII.3.1 RQ1: Are all the faults revealed by a round-trip paths test suite also revealed by transition tree test suites?

The answer to this question is simply no. In two of our experimental objects, the cruise control and the VCR, the complete round-trip paths test suite reveals faults that the other test suites covering round-trip paths in pieces do not reveal. In the case of cruise control, the complete RTP test suite reveals one fault that is not revealed by any other test suite. For the VCR, the results are more dramatic: The RTP test suite reveals 46 faults that neither the BFS test suites nor the random test suites manage to reveal. This is equal to approximately 17% of the highest mutation score achieved by all algorithms. However, only 18 of those 46 mutants are revealed by the DFS test suites.

VIII.3.2 RQ2: Does the algorithm used (BFS, DFS, or Random) to generate the transition tree affect fault detection?

From the data presented before and summarized in VIII.1, the DFS test suites achieve the highest mutation score mean for the ordered set and the VCR experimental

objects, i.e, the more complex FSMs, while they share the highest score with the BFS traversal trees for the cruise control and the ATM experimental objects.

The cruise control has identical DFS and BFS test suites as explained in section VII.1. This neither refutes that argument in favor of using DFS test suites, neither supports it statistically.

For the ATM experimental object, when studying the different trees produced from STAGE-1, one observes that few partial paths (parts of longer paths) are common to all trees. The length of those paths contributes to almost half of the total paths length. Thus, the resulting trees spread horizontally while having relatively small depth. The consequence of this is that when using either the BFS or the DFS algorithm, the differences between the trees are minimal.

More precisely, in all cases, the difference between one tree and another is a replacement of one edge and its destination node. It may be concluded, on the one hand, that this is the reason why the same mutation score is achieved by BFS, DFS, and random algorithms as shown in Figure 33. On the other hand, this same observation is the reason the RTP test suite achieves quite a low mutation score. Recall each test in an RTP test suite is made of the shortest path prefix to a complete round-trip path. Hence such test suite misses all transitions that are neither part of the round-trip paths nor the prefixes added to these paths. In the case of the ATM unlike VCR and ordered set, a high number of transitions does not belong to any RTP path.

In brief, the analysis of the experimental results suggests that the answer to RQ2 is yes: the algorithm used for tree traversal does affect mutation score and DFS seems to be the best alternative (section VIII.1).

VIII.3.3 RQ3: Do the distinct trees generated using one algorithm differ in their effectiveness at finding faults (a.k.a. their mutation score)?

The test suites sometimes vary significantly in terms of mutation score, even within one group of test suites, i.e., using the same traversal algorithm.

The most evident example is the BFS test suites of the VCR experimental object. While the minimum mutation score achieved by BFS test suites is 150 mutants, the maximum is 206 mutants. This is also clear by the high standard deviation of the mutation score of the BFS test suites of this experimental object as indicated in the last row of Table 7. However, the overall trend is a relatively low standard deviation for test suites falling in the same group.

Figure 36 and Figure 34 represent the variation in mutation score of the ordered set test suites and the VCR test suites, respectively. The charts illustrate how the mutation score of test suites generated using the same strategy (BFS, DFS, or Random) can vary in terms of mutation score.

VIII.3.4 RQ4: Is it possible to derive a common trend in the trees generated that helps achieve higher fault detection?

Observing that the test suites generated using the DFS traversal algorithm perform better on average, we tend to believe that there is a positive correlation between the length of test cases (a DFS tree has longer test cases than a BFS tree) and the mutation score.

During the experimentation, trees with high mutation score are compared with lower mutation score trees across the spectrum of the same traversal algorithm. It is observed that in many cases the tree (test suite) with a better mutation score has lengthier paths (test cases). However, no statistical evidence can be deduced from the data. The following chapter attempts to answer this question empirically, by using the results to validate the suggested hypotheses.

Chapter IX Hypotheses

In this section, in light of the results presented earlier, we put forward some hypotheses, which the presented experiments attempt to validate with the collected data.

IX.1 Hypothesis #1: Test Paths Length Affects Mutation Score

To prove the validity of this hypothesis, the mean paths length (MPL) has to be calculated for each test suite.

$$MPL=\left(\frac{sum\ of\ all\ paths\ lengths\ for\ all\ test\ cases\ in\ the\ test\ suite}{number\ of\ test\ cases\ in\ the\ test\ suite}\right)$$

Given the considerable number of trees produced in the experiments, an automation tool has been developed as part of the experimentation to perform the calculations. For each experimental object the mean paths lengths are compared across all traversal algorithms. The mutation scores are plotted against the average lengths for each group of test suites derived using the same traversal algorithm. Also, one chart visualizing the relationship between the average paths lengths and the mutation score for all generated test suites regardless of the traversal algorithm is plotted.

The round-trip paths are excluded from the comparison as it is already concluded from the results that it achieves lower mutation score than the rest of the algorithms due to lower mutation coverage.

IX.1.1 Cruise Control

On the one hand, all test suites derived for the cruise control experimental object achieve the same mutation score, except for one test suite that misses only one of the faults revealed by the other test suites. On the other hand, all the test suites have the same mean

paths length. The results of the calculations for the cruise control experimental object are in support of the suggested hypothesis if the minor difference in mutation score is neglected. However, one cannot use this result to generalize the hypothesis to other SUTs without further experimentation and results analysis.

IX.1.2 ATM

The ATM experimental object test suites achieve the same mutation score across all groups belonging to different traversal algorithms. Contrary to what is hypothesized, the variation in the mean lengths of the test suites paths does not affect the mutation score. The average paths length ranges from 8.57 to 10.86 for the test suites of the ATM experimental object (i.e., around two extra states per paths difference between the test suite with shorter average paths length and the one with longer average paths length). This small difference does not affect the mutation score.

IX.1.3 Ordered Set

For the ordered set experimental object, there are some interesting observations. The average length is the same for all BFS test suites, while the mutation scores vary slightly between 263 to 266 mutants with these test suites. For the randomly generated test suites, the general trend is that the mutation score increases in proportion to MPL as shown in Figure 48. For the DFS test suites, the mutation score varies between the different test suites as the maximum mutation score is 271 while the minimum is 260 mutants. However, the paths lengths have more variations as shown in Figure 48. The general trend line does not support the hypothesis. When comparing the mutation score across all the test suites, it

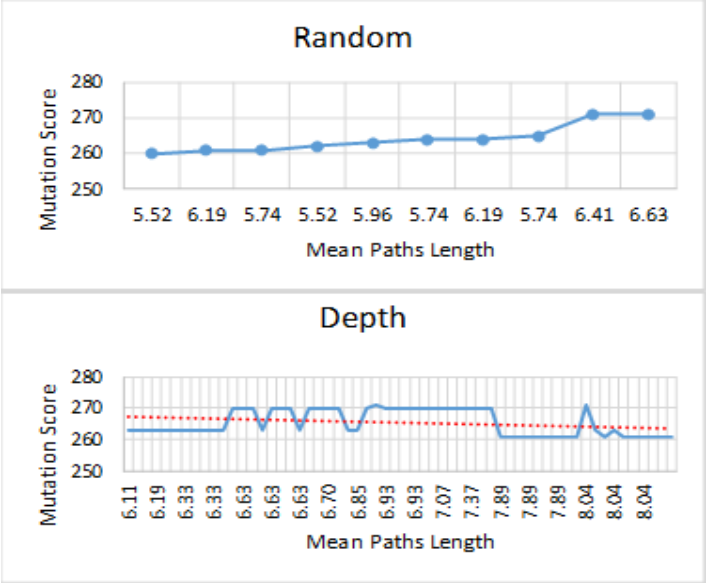


Figure 48 Ordered Set: Killed Mutants Vs. Mean Paths Length for Random and DFS Traversals.

is observed that the general trend is a very slight decrease in mutation score when the test paths length increases. However, the mutation score is at a peak around the median of the test paths mean lengths as shown in Figure 49.

IX.1.4 VCR

The VCR is the largest experimental object, and therefore more interesting results can be expected. The MPL across all VCR BFS test suites is identical, while the mutation

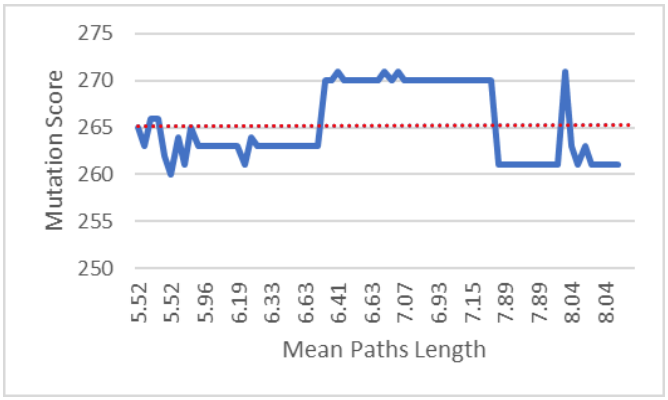


Figure 49 Ordered Set Overall Trend Killed Mutants Vs. Mean Paths Length.

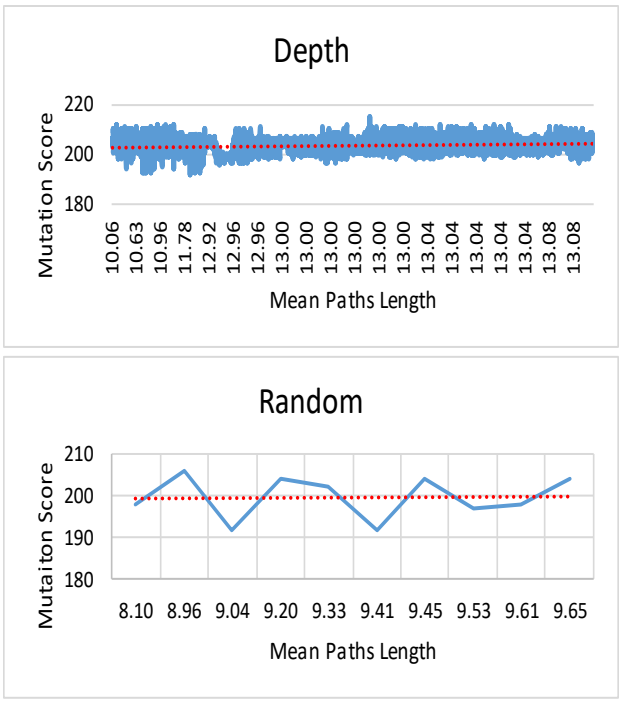


Figure 50 VCR: Killed Mutants Vs. Mean Paths Length for Random and DFS Traversals.

score varies from 150 and 206 killed mutants. This refutes the suggested hypothesis strongly as the mutation score varies significantly by more than 27%. For the random algorithm, the mutation score varies from 192 to 206 killed mutants. The relationship between the mutation score and MPL is shown in Figure 50. The trend line for this case is increasing.

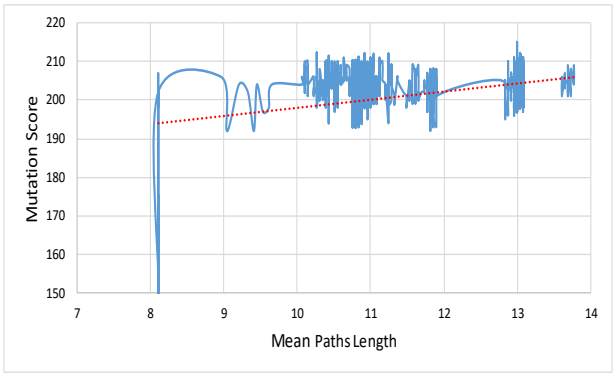


Figure 51 VCR: Overall Trend for No. of Killed Mutants Vs. Mean Paths Length

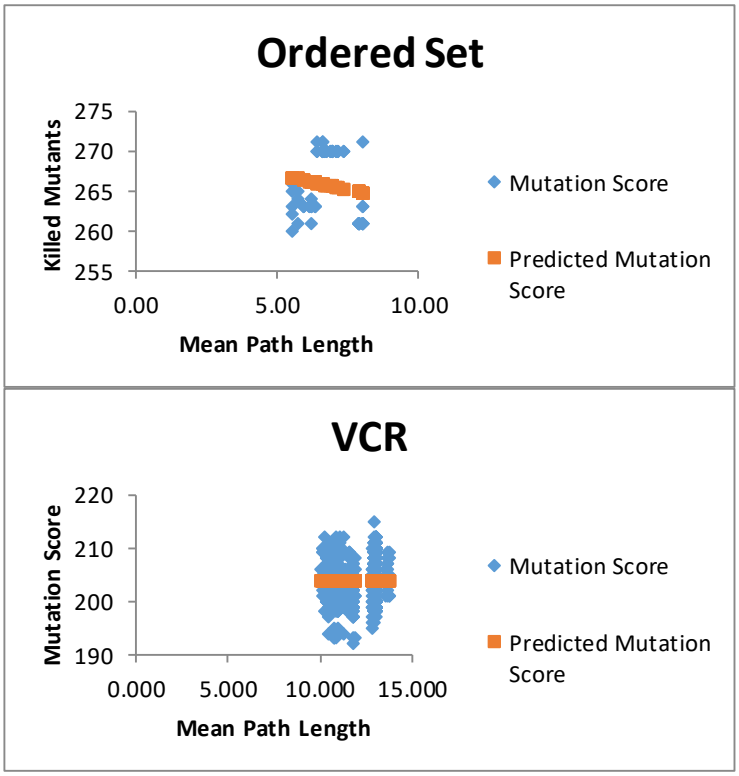


Figure 52 Killed Mutants and Mean Path Length Fit Plot

However, for the DFS test suites, the general trend is a very slow increase in mutation score relative to the MPL Again, the peak mutation scores occur around the median of the MPL. Figure 51 shows the general trend across all test suites produced using the three different traversal algorithms. The mutation score increases in proportion to the test paths mean.

We use excel regression analysis to find the relationship between the path length and the mutation score for both the ordered set and the VCR. The correlation coeefficients for the mutation score in relation to the mean paths length is approximately -0.16 for the ordered set, and -0.004 for the VCR. The results do not allow us to draw a conclusion about a linear relationship between the number of killed mutants and the mean paths length of the used test suite. This is also evident from the plot shown in Figure 52.

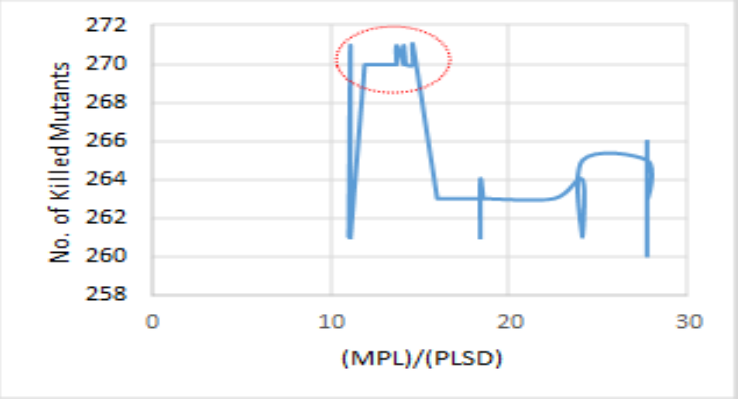


Figure 54 Ordered Set: No. of Killed Mutants Vs. (MPL/PLSD) Ratio.

IX.2 Hypothesis #2: Balancing the Tree Affects Mutation Score

Based on the observation pointed out in IX.1 that high mutation score test suites have MPL close to the median paths length, we suggest the hypothesis that balanced trees (test suites) achieve better mutation score. In other words, test suites that include test cases with similar lengths are a better choice if one wishes to increase the mutation score. However, it has been already shown that the length affects the mutation score positively in many cases. Otherwise, balanced BFS test suites would have been the most effective. Consequently, having relatively balanced trees with lengthiest paths can achieve best results. To translate this into a measurable criterion, the following steps are followed:

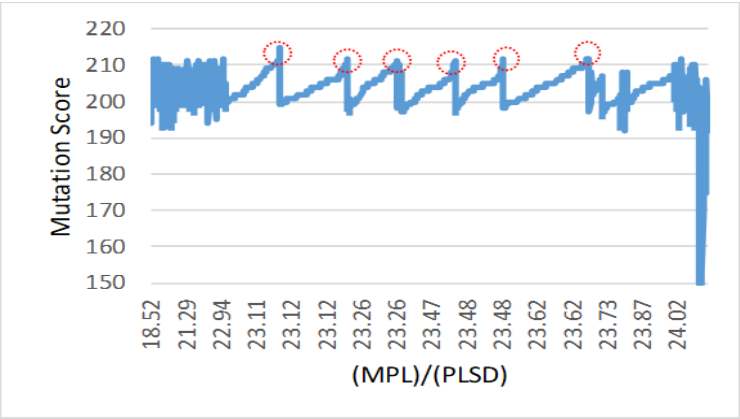


Figure 53 VCR: No. of Killed Mutants Vs. (MPL/PLSD) Ratio.

1. Calculating the mean length of the paths in each test suite, which is already done in IX.1.
2. Calculating the standard deviation of the paths lengths in each test suite. A tree with a low standard deviation is more balanced.
3. Calculate the ratio $\frac{MPL}{Paths\ Lengths\ Standard\ Deviation\ (PLSD)}$ and look for the test suites that maximize this ratio. This test suite maps to the most balanced tree with lengthiest paths.

The calculated ratio for the ordered set and the VCR is plotted against the mutation score as for smaller experimental objects the mutation scores has minor variations if any.

Figure 54 and Figure 53 show the result for ordered set and VCR, respectively. Although the results do not show that the mutation score is proportional to the ratio as suggested above, there is some interesting consistent pattern of high mutation scores (as indicated with the red dots in Figure 54 and Figure 53) that is especially evident in the VCR experimental object with the larger number of test suites. The pattern shows a tangent wave around the average mutation score value horizontal line. In other words, the mutations score constantly increases with the increase in the MPL to PLSD ratio, and then suddenly drops again. Then the mutation score goes through the same cycle repeatedly as the MPL to PLSD ratio increases. Whether the ordered set pattern appearing in Figure 54 is a snapshot of the same behavior if the number of test suites is increased or not, is a question that needs further investigation.

Figure 55 is the line fit plot for the number of killed mutants to the ratio MPL/PLSD. While the correlation coefficient of the ordered set is negative (-0.016825) when running regression analysis to understand the relationship between these two

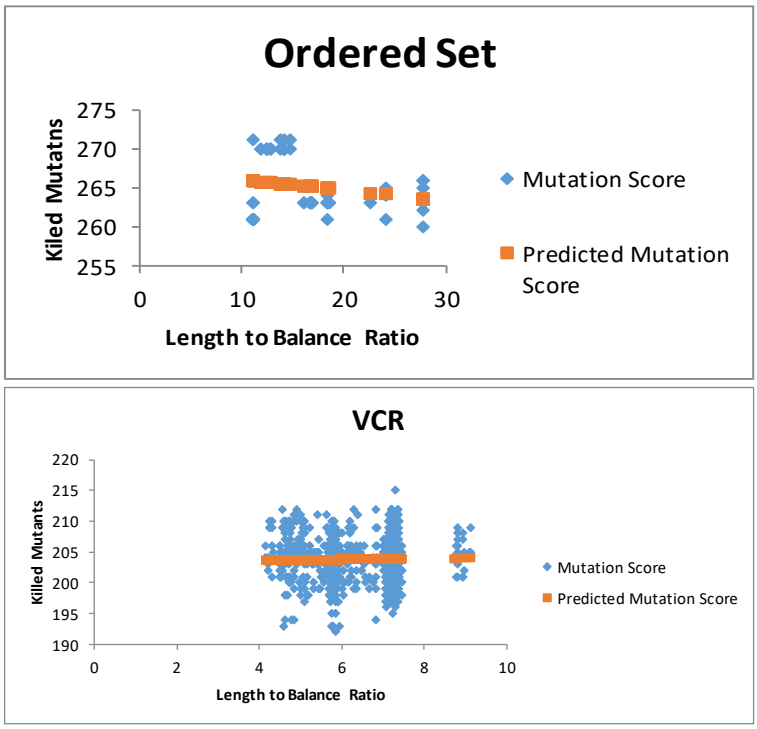


Figure 55 Killed Mutants to MPL/PLSD Fit Plot

variables, the correlation coefficient for the VCR is positive (0.016818). This indicates that the calculated ratio affects the number of killed mutants for the VCR test suites. However, the value of the correlation coefficient is not high enough to indicate a significant relationship between the mutation score and the MPL/PLSD ratio.

IX.3 Hypothesis #3: There is a relation between RTP performance and the FSM connectivity.

We discuss in section V.1.1 that unlike Binder's test suites that include simple paths in addition to RTPs in piecewise manner[8], our RTP test suites are composed of complete RTPs as well as the shortest prefixes that lead to those RTPs. By doing this, we focus on the complete RTPs performance relative to covering RTPs in pieces. In section VIII.2.2, we mention an observation that suggests better RTP performance relative to other test

suites for systems modeled using more connected FSMs. We find the mentioned observation worthy of more analysis. Therefore, we formulate the following hypothesis: The percentage of mutants that the RTP kills to the total killed mutants increases as the connectivity of the FSM of the SUT increases.

In Table 10, we list the no. of states, number of transitions, number of possible transitions, and number of missing transitions for all four experimental objects. The number of possible transitions is the number of transitions if the FSM was fully connected. In other words, we deal with the FSMs here as a directed graph. A complete directed graph in graph theory is a graph that has exactly one edge from every vertex to every other vertex [62]. Translating this in FSM terms, a complete FSM is an FSM with exactly one transition from every state to every other state. To calculate the total number of possible transitions in a complete FSM, we use the following formula:

All possible no. of transitions = (no. of states) × (no. of states)

Equation 3

Note that we exclude here transitions from a state to itself as well as transitions that share the same initial state and the same destination state. The option of measuring the connectivity by the number of transitions present in the FSM is only a surrogate since it

Table 10 Connectivity of Experimental Objects.

SUT	*No. of states*	*No. of transitions*	*No. of possible transitions*	*No. of missing transitions*	*RTP killed mutant/total killed mutants*
Cruise Control	5	29	25	0	100.00%
ATM	10	22	100	78	97.51%
Ordered Set	9	35	81	46	78.30%
VCR	17	65	289	225	48.55%

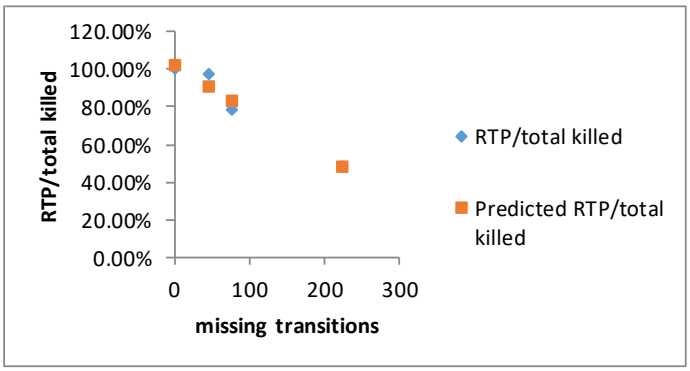


Figure 56 Regression analysis of the RTP killed mutant to the missing transitions

excludes the number of states from the calculation. For example, an FSM with three states and six transitions is a connected FSM, while an FSM with four states and the same number of transitions (six) is not. Therefore, the number of states should be part of the equation used to measure the connectivity of the state machine.

We use Equation 3 to calculate the fourth column of Table 10, and we subtract the number of transitions (column 2) from this number to get the number of missing transitions. The number of missing transitions is the number of transitions that are needed to make the FSM/digraph complete.

Figure 56 shows a linear relationship between the killed mutants/total number of killed mutants for the RTP test suite and the number of missing transitions (Table 10). As apparent from the figure, the proportion of mutants killed by the RTP test suite to the total killed mutants is inversely proportional to the number of missing transitions. In other words, the less connected the FSM of the SUT the worse the performance of the RTP test suite. This proves Hypothesis #3.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.978536899
R Square	0.957534463
Adjusted R Square	0.936301694
Standard Error	0.059964843
Observations	4

ANOVA

	df	*SS*	*MS*	*F*	*gnificance F*
Regression	1	0.162159	0.162159	45.09701	0.021463
Residual	2	0.007192	0.003596		
Total	3	0.169351			

	Coefficients	*andard Err*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*ower 95.0%*	*pper 95.0%*
Intercept	1.019482754	0.04317	23.61556	0.001788	0.833737	1.205228	0.833737	1.205228
missing transitions	-0.002390587	0.000356	-6.71543	0.021463	-0.00392	-0.00086	-0.00392	-0.00086

Figure 57 Regression Analysis of the Relationship between FSM Connectivity and RTP Performance.

Figure 57 shows the results of the regression analysis performed by excel. The results show a low ρ value of <0.05 as highlighted in yellow, which means that the null hypothesis is rejected [4, 78]. The null hypothesis here is that the mutation score of the RTP test suite is not related to the number of transitions that are if available would make the FSM complete. The results show that the correlation coefficient is very close to one (highlighted in red in Figure 57), which means that the relationship between both variables is almost linear as clear from Figure 56.

IX.4 Hypothesis #4: Mutation Score is Affected by Number of Complete RTPs

Based on the conclusion we reached that RTP test suite kills mutants that are not killed by other test suites (section VIII.3.1), in this section, we are investigating whether

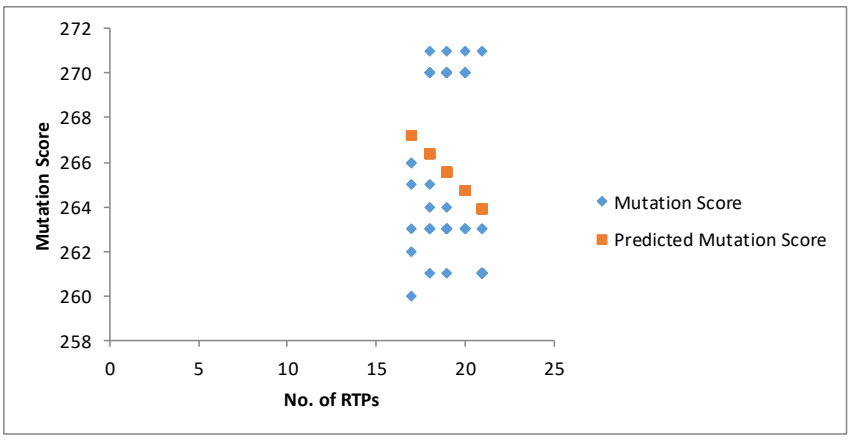


Figure 58 Regression Analysis of Mutation Score Vs. No. of Complete RTPs for Ordered set.

the number of complete RTPs covered in other test suits (BFS, DFS, or random) affect their mutation score. Since the cruise control and the ATM test suites have very similar mutation scores, we focus our analysis here on the ordered set and the VCR test suites. To do this, we calculate the mutation scores and the number of complete RTPs in each BFS, DFS, and Random test suite in the ordered set and the VCR. We perform regression analysis on the results obtained using both experimental objects.

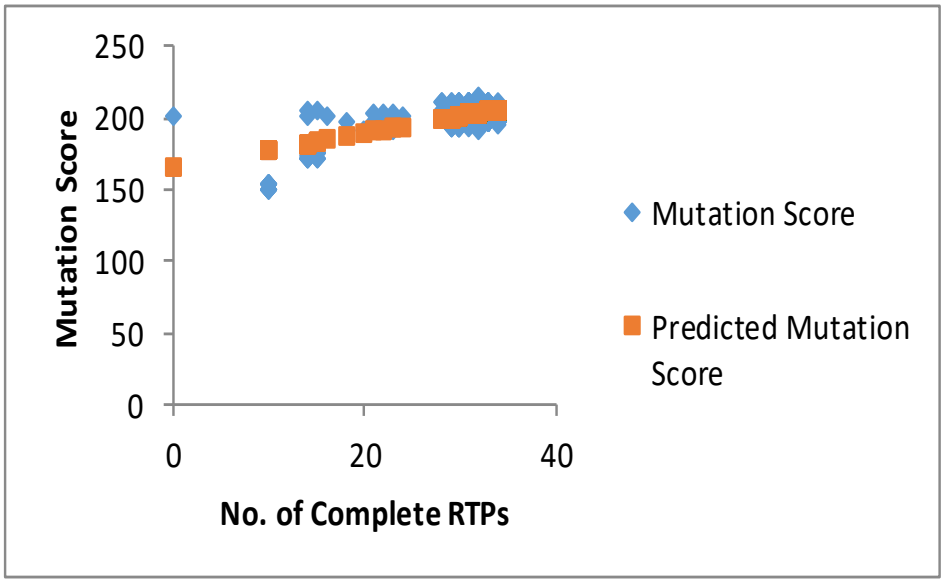


Figure 59 Regression Analysis of Mutation Score Vs. No. of Complete RTPs for VCR Test Suites.

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.572868
R Square	0.328178
Adjusted R Square	0.327828
Standard Error	4.556992
Observations	1921

ANOVA

	df	*SS*	*MS*	*F*	*gnificance F*
Regression	1	19466.43	19466.43	937.4104	5.7E-168
Residual	1919	39850.28	20.76617		
Total	1920	59316.71			

	Coefficient	*tandard Err*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*ower 95.0%*	*Upper 95.0%*
Intercept	165.8583	1.230067	134.8368	0	163.4459	168.2707	163.4459	168.270713
No. of RTPs	1.179666	0.03853	30.61716	5.7E-168	1.104102	1.25523	1.104102	1.255230038

VCR

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.264351
R Square	0.069882
Adjusted R Square	0.056781
Standard Error	3.830536
Observations	73

ANOVA

	df	*SS*	*MS*	*F*	*gnificance F*
Regression	1	78.27113	78.27113	5.334360849	0.023821
Residual	71	1041.784	14.67301		
Total	72	1120.055			

	Coefficient	*tandard Err*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*ow*
Intercept	281.2471	6.936351	40.54684	7.70165E-51	267.4164	295.0778	26
No. of RTPS	-0.82593	0.357604	-2.30962	0.023821256	-1.53897	-0.11289	-1

Ordered Set

Figure 60 Mutation Score and RTP Count Regression Analysis

For the ordered set, with a sample of 73 observations (test suites) the correlation coefficient ρ between the mutation score and the number of RTPs covered is equal to -0.264. Since ρ is negative, we conclude that the value of the mutation score decreases as the number of covered RTPs increases. This might be counterintuitive, but the small value of ρ shows that there is no linear correlation between the mutation score and the number of RTPs in a test suite [86]. This is also clear from Figure 58 where the values of the mutation score are scattered away from the line representing the predicted mutation score. For the VCR test suites, using 1921 observations, the correlation coefficient is 0.573. Figure 59 shows a more linear relationship between the mutation score of the VCR test suites and the number of complete RTPs they include.

The VCR results lead us to reject the null hypothesis which, in our case, is that there is no relationship between the two variables—mutation score and number of complete RTPs—with much higher confidence than for the ordered set. This is shown in Figure 60 where the p-value (highlighted) is less than 0.05 in the VCR, which means that the null

hypothesis is rejected [4]. This leads us to suggest that the more connected the FSM that models the SUT, the more advisable it is to include complete RTPs in the test suites used for this system.

Chapter X Threats to validity

Wohlin et al. [86] classify validity threats in software engineering into conclusion, internal, construct and external threats. The conclusion threat focuses on the statistical significance of the results. Despite a large number of test suites, we experiment on, we still recommend applying the same experiments conducted on the study on more experimental objects of size closer to the VCR and the ordered set presented. In our experimentation, internal threats to validity are handled by avoiding variation in factors other than the factor

Table 11 Example of Mutants and Corresponding State-based Faults

Inserted Mutant	Equivalent state fault	Explanation

The original version of the SUT checks if the data structure is full, it resizes the array: The mutant replaces +1 by -1
`if (_last+1 >= _set_size)`
`this.resizeArray();`
LVR: 1 |==> -1
	Missing or wrong transition that results in an incorrect target state.	In this case, the array gets resized before it is full. This means that instead of reaching a filled state, the transition puts the system in a partially filled state (appendix A.3).

The original version of the SUT checks if the data structure is not overflowing, it adds the new element to the data structure. The mutant replaces the if statement so that the data structure is always assumed to be overflowing. **COR: overflow |==> true**	Missing or incorrect event that results in a valid event being ignored,	If the program is trying to add an element to the data structure, it wouldn't be able to since it is always in the overflow state where no elements could be added.
The SUT is trying to add an element to the ordered set. The mutant totally deletes this statement. **STD: set[i] = n |==> <NO-OP>**	A missing or incorrect action that results in the wrong behavior,	An element that was intended to get added to the data structure is not being added.
The SUT is checking if the data structure is Empty. The mutant returns always false even if the data structure is not empty. **ROR: _last < 0 |==> false**	An extra, missing or corrupt state,	The isEmpty function returns false when the data structure is empty. This results in always missing the Empty state (appendix A.3).
The original SUT removes an element from the SUT if the data structure size is greater than or equal to zero. The mutant replaces the zero by a negative one. **LVR: 0 |==> -1**	An illegal event that gets accepted when it should not	If the array size is 0, the system still attempts to remove an element from the array. Removing an element from an empty array gets accepted while it should not.

under test. For instance, we verified that all test suites having common paths use the same input values to the same functions. This is especially important with the ordered set since events require integer input values. For construct validity that focuses on relating the theory behind the experiments to the observations [25], Major mutation seeding tool is chosen with this concern in mind as explained in section V.2. Assuming that a mutant that is covered and alive with all our test suites is an equivalent mutant, is a minor construct validity threat. However, exercising and comparing large numbers of test suites make this assumption reasonable.

Whether the seeded mutants are representative of state faults is a legitimate question. Nevertheless, the test oracle that we enclose in STAGE-2 is designed to detect state faults only as mentioned in section V.1.2. The oracle throws an exception in the event that the actual state of the object is different from the state expected according to the FSM model and only then a mutant is considered killed. The details of the oracle strategy are discussed in section V.1.1. Hence, all mutants that do not correspond to state faults are automatically not caught for all test suites and therefore does not affect the result of comparing the effectiveness of the test suites included in our experiments. Table 11 shows some examples of mutants inserted by Major and their corresponding state-based faults. All the examples shown are for the ordered set experimental object. For example, the second row of the table is a COR mutant that sets the overflow variable to true. This replacement causes the system to be always in the overflow state. This means that when an event to add an element, for instance is triggered, the system ignores this event since it assumes that the current state is an overflow state and no additional elements can be added. This mutant is killed by all test suites.

The fourth threat to validity is the external validity threat that is concerned with whether the results can be generalized outside the scope of this study. We choose experimental objects that are of varied sizes and characteristics to make sure they are representative of a large number of FSMs that are used for testing purposes. As explained in Chapter IV, we investigated industrial case studies and concluded that our experimental objects are of similar complexity and size. This is to overcome the external validity threat. However, one must be cautious in generalizing the results without experimenting with more experimental objects. In our results analysis, we are specific about which experimental objects contribute to results generalization.

Chapter XI Conclusion

FSMs are widely used to model software artifacts [83]. Consequently, the area of FSM testing has received remarkable attention from both researchers and practitioners. This thesis provides thorough background information on finite state machines, state-based testing, and mutation analysis that we use to measure the effectiveness of our test suites. Following that, we focus our research works on two testing criteria that have proven to be good compromises in terms of effectiveness and cost for testing systems based on FSMs [60]. Those criteria are complete RTPs and transition trees that cover RTPs in pieces. We discuss the differences between the two criteria and based on the discussion we formulate four research questions.

The experiments are performed on four experimental objects that are carefully selected to be different in structure and size and to resemble real-life systems. Those are a cruise control system, an ATM, an ordered set data structure and a VCR. Then, we explain in detail the experimental setup we use to conduct our experiments to answer the different research questions.

The first research question to answer is whether exercising the round-trip paths in a complete manner reveals more faults than exercising those round-trip paths in a piecewise manner using transition trees. Results of the experiments show that complete RTPs find faults that are not detectable by transition trees that cover RTPs in pieces. The second question is whether using a certain tree traversal technique (BFS, DFS, and random) affects the mutation score of the resulting test suites. The answer to this question is yes as we prove using different statistical techniques that the different groups of test suites vary in

their ability to kill mutants. The third research question is whether the mutation score differs among the test suites generated using the same traversal algorithm. We concluded that it differs. The fourth research question asks whether it is possible to find common criteria for test suites that detect more faults. The answer to this question is that test suites generated from more balanced trees with lengthiest possible paths are more effective but with low significance. In addition, systems that are based on more connected FSMs benefit the most from complete RTP testing and hence we recommend including complete RTPs in test suites designed for those systems.

XI.1 Contributions

This thesis contributes to the state-based testing literature in many ways. After reviewing the literature and providing sufficient background, we empirically prove that transition trees do not always cover RTPs in a complete manner. Then we define the research questions that need to be answered. To answer the addressed questions, we design and develop a robust reusable experimental setup that includes documented steps to repeat experiments similar to those we conduct in this dissertation. The setup embraces one of the main contributions of our work: designing and implementing a toolchain. The first of our two tools generates all possible transition trees using BFS or DFS traversal algorithms of the input FSM. To the best of our knowledge, the algorithms we designed are the first in the literature to generate all possible BFS/DFS trees instead of just one tree from the input graph. We provide a literature review of the tree traversal algorithms available in the literature and we show the uniqueness of our algorithms [45]. The tool also automates the generation of complete RTP test suites as well as random trees of the input graph depending

on the choice of the user [43]. The generated test suites (a total of 102,113 test suites in our case) can be run using the second of our tools. The tool can also be used to generate JUnit files to run those test suites.

Our empirical evaluation that is conducted using the toolchain and Major as a mutation analysis tool is the focal contribution of our work. Previous work that administers similar experiments uses twelve test suites, while we orchestrate our experiment using 2060 test suites [44]. As a result of our experiment, we collect a significant amount of data that we analyze using descriptive and inferential statistics employing several tests (ANOVA, t-test, Kruskal-Wallis, and U-test).

The thesis reaches several significant conclusions: complete RTPs detect faults that are not detectable by other test suites, the method used to generate the transition tree (BFS, DFS, or random) affects the number of killed mutants, the effectiveness of different transition trees generated using one method may differ, and the more connected the state machine the more we recommend using complete RTPs to test the system that is modeled using this FSM. We attempted to find a relationship between the average length of the test cases in transition trees test suites and effectiveness, but there was no statistical evidence that the average length of the paths affects the mutation score. However, the trees with a minimum difference between the length of the path have a better mutation score for the VCR which is the experimental object that has more test suites and hence more statistical significance. We also prove that depending on the structure of the FSM and the nature of the SUT, mutants belonging to different mutation operators may be detected at different rates. However, in general, the different testing methods do not affect the mutants detected for specific mutation operators.

Last, the thesis results in the following publications:

1. H. Khalil and Y. Labiche, “State-Based Tests Suites Automatic Generation Tool (STAGE-1),” in *Proceedings - International Computer Software and Applications Conference*, 2017, vol. 1, pp. 357–363.
2. H. Khalil and Y. Labiche, “On FSM-Based Testing: An Empirical Study: Complete Round-Trip Versus Transition Trees,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 305–315.
3. H. Khalil and Y. Labiche, “Finding All Breadth First Full Spanning Trees in a Directed Graph,” in *Proceedings - International Computer Software and Applications Conference*, 2017, vol. 2, pp. 372–377.
4. H. Khalil, “Finite State Machine Testing Complete Round-Trip Versus Transition Trees: On the Road of Finding the Most Effective Criterion,” in *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2017, pp. 108–111.
5. H. Khalil and Y. Labiche, “Experimental Study on Transition Trees and Complete Round Trip Paths Testing Criteria”, Journal paper in progress.

XI.2 Future Research Opportunities

There is space for more research that complements or builds on our thesis work. Now that a reusable experimental setup is available and can be used for conducting similar empirical studies, one can address the threats to validity that we mentioned (Chapter X) by conducting more experiments using systems of a size that resembles the VCR.

In our research, we mainly focus on the effectiveness of the test suites in terms of fault detection. We touch on the cost of the different testing methods by calculating the execution time. However, more research can be done on the relationship between the cost of running the test suites and the method used to generate those test suites. Then, more work can be done to find compromises between the cost and effectiveness.

In generating the transition trees, we cover each transition once and only once. This might not necessarily be the optimum case. Researchers might study the option of finding criteria by which they decide to cover some transitions twice or not to cover them at all.

By design, all the faults detected in our experiments are state-based faults. Nevertheless, the mutation tool we used is not specially designed to insert state faults. We believe that seeding state mutants while guaranteeing that all types of state faults are included in the process is worth more investigation.

All the work done in the dissertation focus on FSMs. Another research area that we believe is interesting is expanding our work to handle Extended Finite State Machines (EFSM). One simple possibility is to flatten the EFSM and convert it to an FSM, which might result in an explosion of the number of states [83]. However, given that the experimental setup we have established in this thesis is mostly automated, this should not be an obstacle to generating test suites and measuring their effectiveness. However, providing input data, if needed, to the test cases might be a time-consuming task. Another possibility is handling guard conditions automatically, but in such a case it will be the responsibility of the test engineers to design test data that exercise all possible scenarios. This is an open and interesting area of research that can benefit from our work.

Future work can also study relations (with our experimental objects, test suites, and mutants) between various measures of diversity among test cases and mutation score, similar to what others have done [32].

Bibliography

[1] P. Amman and J. Offutt, *Introduction to Software Testing*. 2008.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," *Proceedings. 27th Int. Conf. Softw. Eng. 2005. ICSE 2005.*, pp. 402–411, 2005.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[4] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[5] B. Beizer, *Software Testing Techniques*, Second., vol. 2, no. 10. New York, 2003.

[6] B. Beizer, *Black-box testing: techniques for functional testing of software and systems*. 1995.

[7] E. Berard, *Testing Object-oriented Software (Abstract)*, vol. 4, no. 2. 1992.

[8] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. 1999.

[9] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid, "Automatic Test Generation for EFSM-based Systems," *2012 Int. Work. Inf. Electron. Eng.*, vol. 29, pp. 1–60, 2012.

[10] L. C. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," *Proceedings. 26th Int. Conf. Softw. Eng.*, pp. 86–95, 2004.

[11] M. Broy, *Model-based testing of reactive systems: advanced lectures*. 2005.

[12] A. D. Brucker and J. Julliand, “Editorial: Editorial for the special issue of STVR on tests and proofs volume 2: Tests and proofs for improving the generation time and quality of test data suites,” *Softw. Test. Verif. Reliab.*, vol. 24, no. 8, pp. 591–592, 2014.

[13] M. Chakraborty, R. Mehera, and R. K. Pal, “Divide-and-Conquer: An Approach to Generate All Spanning Trees of a Connected and Undirected Simple Graph,” *Adv. Intell. Syst. Comput.*, vol. 3, pp. 65–84, 2014.

[14] J. Char, “Generation of Trees, Two-Trees, and Storage of Master Forests,” *IEEE Trans. Circuit Theory*, vol. 15, no. 3, pp. 228–238, 1968.

[15] G. Chartrand, *Introductory Graph Theory*. 2012.

[16] T. S. Chow, “Testing Software Design Modeled by Finite-State Machines,” *IEEE Trans. Softw. Eng.*, vol. SE-4, no. 3, pp. 178–187, 1978.

[17] T. Cormen, R. Rivest, and C. Leiserson, *Introduction to Algorithms*, Third. Cambridge, Mass., United States: McGraw-Hill, 2009.

[18] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,” *Empir. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[19] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, “Experimental evaluation of FSM-based testing methods,” *Proc. - 3rd IEEE Int. Conf. Softw. Eng. Form. Methods, SEFM 2005*, pp. 23–32, 2005.

[20] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, “FSM-based conformance testing methods: A survey annotated with experimental evaluation,” *Inf. Softw. Technol.*, vol. 52, no. 12, pp. 1286–1297, 2010.

[21] W. Duplications, “Generation of Trees,” no. July, pp. 7–8, 1969.

[22] H. El-Gendy and M. Amer, “Comprehensive Study of the Testing Coverage of Testing Methods for Communications Protocols and Software,” *J. Am. Sci. J Am Sci*, vol. 99, no. 11, pp. 79–84, 2013.

[23] A. T. Endo and A. Simao, “Experimental comparison of test case generation methods for finite state machines,” *Proc. - IEEE 5th Int. Conf. Softw. Testing, Verif. Validation, ICST 2012*, pp. 549–558, 2012.

[24] A. T. Endo and A. Simao, “Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods,” *Inf. Softw. Technol.*, vol. 55, no. 6, pp. 1045–1062, 2013.

[25] R. Feldt and A. Magazinius, “Validity Threats in Empirical Software Engineering Research - An Initial Survey.,” *Proc. Int’l Conf. Softw. Eng. Knowl. Eng.*, no. August, pp. 374–379, 2010.

[26] M. Fischer, R. Tönjes, R. Lasch, and A. Sciences, “This paper was presented as part of the SOCNE workshop held in conjunction with IEEE ETFA â€™ 2011 A New Approach for Automatic Generation of Tests for Next Generation Network Communication Services,” 2011.

[27] P. Frankl and S. Weiss, “An experimental comparison of the effectiveness of branch testing and data flow testing,” *Software Engineering, IEEE …*. 1993.

[28] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, “Test selection based on finite state models,” *Softw. Eng. IEEE Trans.*, vol. 17, no. 6, pp. 591–603, 1991.

[29] H. N. Gabow and E. W. Myers, “Finding All Spanning Trees of Directed and

Undirected Graphs," *SIAM J. Comput.*, vol. 7, no. 3, pp. 280–287, 1978.

[30] H. Gomaa, *Designing Concurrent, Distributed, and Real time applications with uml*. Boston: Addison Wesley, 2000.

[31] S. L. Hakimi, "On trees of a graph and their generation," *J. Franklin Inst.*, vol. 272, no. 5, pp. 347–359, 1961.

[32] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 1, pp. 1–42, 2013.

[33] N. E. Holt, L. C. Briand, and R. Torkar, "Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study," *Inf. Softw. Technol.*, vol. 56, no. 8, pp. 890–910, 2014.

[34] N. E. Holt, R. Torkar, L. Briand, and K. Hansen, "State-based testing: Industrial evaluation of the cost-effectiveness of round-trip path and sneak-path strategies," *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, pp. 321–330, 2012.

[35] G. R. Iversen and H. Norpoth, *Analysis of variance*, vol. 1. 1987.

[36] R. Jayakumar, K. Thulasiraman, and M. N. Swamy, "Complexity of Computation of a Spanning Tree Enumeration Algorithm," *IEEE Trans. Circuits Syst.*, vol. 31, no. 10, pp. 853–860, 1984.

[37] D. Jungnickel, *Graphs, Networks and Algorithms*, vol. 53, no. 9. 2008.

[38] N. Juristo, A. M. Moreno, and S. Vegas, "Reviewing 25 Years of Testing Technique Experiments," *Empir. Softw. Eng.*, vol. 9, no. 1/2, pp. 7–44, 2004.

[39] R. Just, "The major mutation framework: efficient and scalable mutation analysis for Java," *Proc. 2014 Int. Symp. Softw. Test. Anal. - ISSTA 2014*, pp. 433–436, 2014.

[40] R. Just, M. D. Ernst, and G. Fraser, “Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States,” *Proc. 2014 Int. Symp. Softw. Test. Anal.*, pp. 315–326, 2014.

[41] R. Just, F. Schweiggert, and G. M. Kapfhammer, “MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler,” *2011 26th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2011, Proc.*, pp. 612–615, 2011.

[42] S. Kapoor and H. Ramesh, “Algorithms for enumerating all spanning-trees of undirected and weighted graphs,” *Proc. 2nd Work. Algorithms {&} Data Struct.*, no. 519, pp. 461–472, 1991.

[43] H. Khalil and Y. Labiche, “State-Based Tests Suites Automatic Generation Tool (STAGE-1),” in *Proceedings - International Computer Software and Applications Conference*, 2017, vol. 1, pp. 357–363.

[44] H. Khalil and Y. Labiche, “On FSM-Based Testing: An Empirical Study: Complete Round-Trip Versus Transition Trees,” in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, 2017, pp. 305–315.

[45] H. Khalil and Y. Labiche, “Finding All Breadth First Full Spanning Trees in a Directed Graph,” in *Proceedings - International Computer Software and Applications Conference*, 2017, vol. 2, pp. 372–377.

[46] H. Khalil and Y. Labiche, “SQUALL Repository,” 2017. [Online]. Available: https://github.com/yvanlabiche/STAGE.

[47] M. Khalil and Y. Labiche, “On the round trip path testing strategy,” *Proc. - Int. Symp. Softw. Reliab. Eng. ISSRE*, pp. 388–397, 2010.

[48] D. E. Knuth 1938, *The art of computer programming*. 2005.

[49] D. C. Kozen, *The Design and Analysis of Algorithms*, vol. 1. 1992.

[50] W. H. Kruskal and W. A. Wallis, “Use of Ranks in One-Criterion Variance Analysis,” *J. Am. Stat. Assoc.*, vol. 47, no. 260, pp. 583–621, 1952.

[51] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines - A survey,” *Proc. IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[52] N. L. Leech and A. J. Onwuegbuzie, “A call for greater use of nonparametric statistics,” *Annu. Meet. MidSouth Educ. Res. Assoc.*, p. Report: ED471346. 25p, 2002.

[53] J. Lilius, I. P. Paltor, F.- Turku, J. Lilius, and I. Porres, “The Production Cell : An Exercise in the Formal Verification of a UML Model,” vol. 0, no. c, pp. 1–10, 2000.

[54] Q. Lin, “Omproving State-based Coverage Criteria Using Data.pdf.” p. 202, 2004.

[55] T. Matsui, “An algorithm for finding all the spanning trees in undirected graphs,” no. METR93-08, 1993.

[56] W. Mayeda and S. Seshu, “Generation of Trees Without Duplications,” *IEEE Trans. Circuit Theory*, vol. 12, no. 2, pp. 181–185, 1965.

[57] A. M. Memon, M. E. Pollack, and M. Lou Soffa, *Automated test oracles for GUIs*, vol. 25, no. 6. 2000.

[58] T. Mizoi and S. Osaki, “Probabilistic Analysis of the Time Complexity of Quicksort,” vol. 79, no. 3, 1996.

[59] R. Motwani and P. Raghavan, “Randomized algorithms,” *Algorithms Theory Comput. Handb.*, pp. 12–1--12–24, 2010.

[60] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta, “Assessing, comparing, and combining state machine-based testing and structural testing: A series of

experiments," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 161–187, 2011.

[61] K. Naik and P. Tripathy, *Software testing and Quality assurance*, vol. 1. 2008.

[62] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science*. 2017.

[63] J. von Neumann, "Automata Studies." pp. 43–98, 1956.

[64] J. Offutt, "MuJava: an automated class mutation system."

[65] A. Petrenko, "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method," *IEEE Trans. Softw. Eng.*, vol. 20, no. 2, pp. 149–161, 1994.

[66] A. Petrenko and G. v. Bochmann, "Selecting Test Sequences for Partially-specified Nondeterministic Finite State Machines," *7th IFIP WG 6.1 Int. Work. Protoc. Test Syst.*, pp. 95–110, 1995.

[67] A. Petrenko, A. Simao, and J. C. Maldonado, "Model-based testing of software and systems: Recent advances and challenges," *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 4, pp. 383–386, 2012.

[68] A. Petrenko and N. Yevtushenko, "Testing from partial deterministic FSM specifications," *IEEE Trans. Comput.*, vol. 54, no. 9, pp. 1154–1165, 2005.

[69] S. C. Pinto Ferraz Fabbri *et al.*, "Mutation analysis testing for finite state machines," *Softw. Reliab. Eng. 1994. Proceedings., 5th Int. Symp.*, pp. 220–229, 1994.

[70] Q. Requirements, "International Standard Iso / Iec," vol. 25021, 2012.

[71] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.

[72] K. Sabnani and A. Dahbura, “A protocol test generation procedure,” *Comput. Networks ISDN Syst.*, vol. 15, no. 4, pp. 285–297, 1988.

[73] a. Simão, a. Petrenko, and J. C. Maldonado, “Comparing finite state machine test coverage criteria,” *IET Softw.*, vol. 3, no. 2, p. 91, 2009.

[74] A. Simão and A. Petrenko, “Fault coverage-driven incremental test generation,” *Comput. J.*, vol. 53, no. 9, pp. 1508–1522, 2010.

[75] A. Simão, A. Petrenko, and N. Yevtushenko, “Generating reduced tests for FSMs with extra states,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5826 LNCS, pp. 129–145, 2009.

[76] K. Sörensen and G. K. Janssens, “An algorithm to generate all spanning trees of a graph in order of increasing cost,” *Pesqui. Operacional*, vol. 25, no. 2, pp. 219–229, 2005.

[77] I. Specification, “CPRI Specification V1.4 (2006-03-31),” vol. 4, pp. 1–64, 2006.

[78] E. Suez, “State Machine Compiler.” [Online]. Available: http://smc.sourceforge.net/.

[79] H. Ural and B. Yang, “A Test Sequence Selection Method for Protocol Testing,” *IEEE Trans. Commun.*, vol. 39, no. 4, pp. 514–523, 1991.

[80] M. Utting and B. Legeard, *Practical Model-Based Testing*, 1st ed. Morgan Kaufmann, 2006.

[81] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” no. April 2011, pp. 297–312, 2012.

[82] M. P. Vasilevskii, “Failure diagnosis of automata,” *Cybernetics*, vol. 9, no. 4, pp. 653–665, 1973.

[83] F. Wagner, T. Wagner, P. Wolstenholme, and F. Group, *Modeling Software with Finite State Machines A Practical Approach*. 2006.

[84] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists*, vol. 3rd. 2012.

[85] F. and Wilson, *Statistical Methods*. 2003.

[86] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, vol. 9783642290. 2012.

[87] Zaiontz Charles, "Real Statistics Using Excel," 2014. [Online]. Available: http://www.real-statistics.com/reliability/cronbachs-alpha/. [Accessed: 20-Aug-2005].

[88] A. Hurson, Ed., *Advances in Computers*, 1st ed., vol. 98. Academic Press, 2015.

[89] "Computer Science Finite State Machines," 2003. [Online]. Available: http://www.multiwingspan.co.uk/as1.php?page=fsm.

[90] "Graphviz." [Online]. Available: http://www.graphviz.org/.

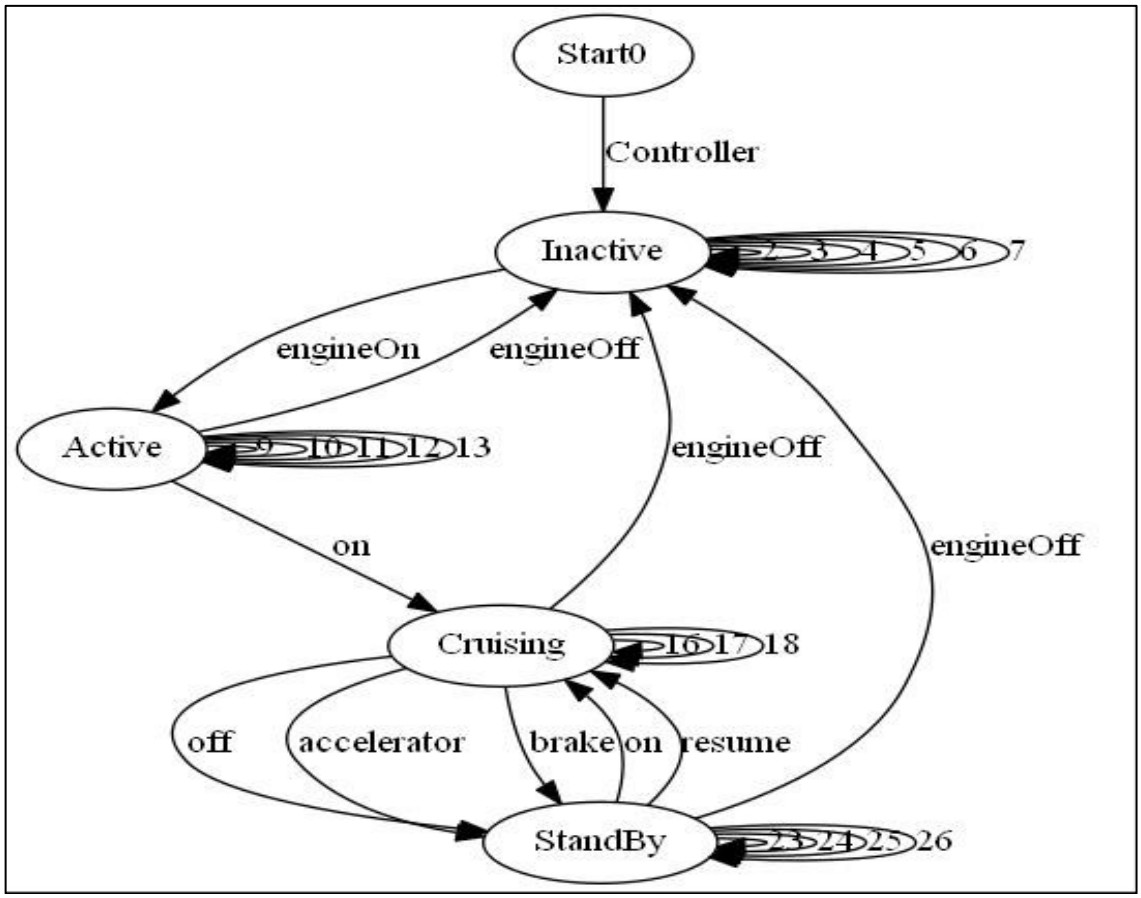
[91] "Traverse a graph, one vertex, all possible round trips," 2014. [Online]. Available: http://stackoverflow.com/questions/23547172/traverse-a-graph-one-vertex-all-possible-round-trips. [Accessed: 21-Apr-2016].

[92] "Shortest path (fewest nodes) for unweighted graph." [Online]. Available: http://stackoverflow.com/questions/1579399/shortest-path-fewest-nodes-for-unweighted-graph. [Accessed: 21-Apr-2016].

Appendix A. Experimental Object FSMs

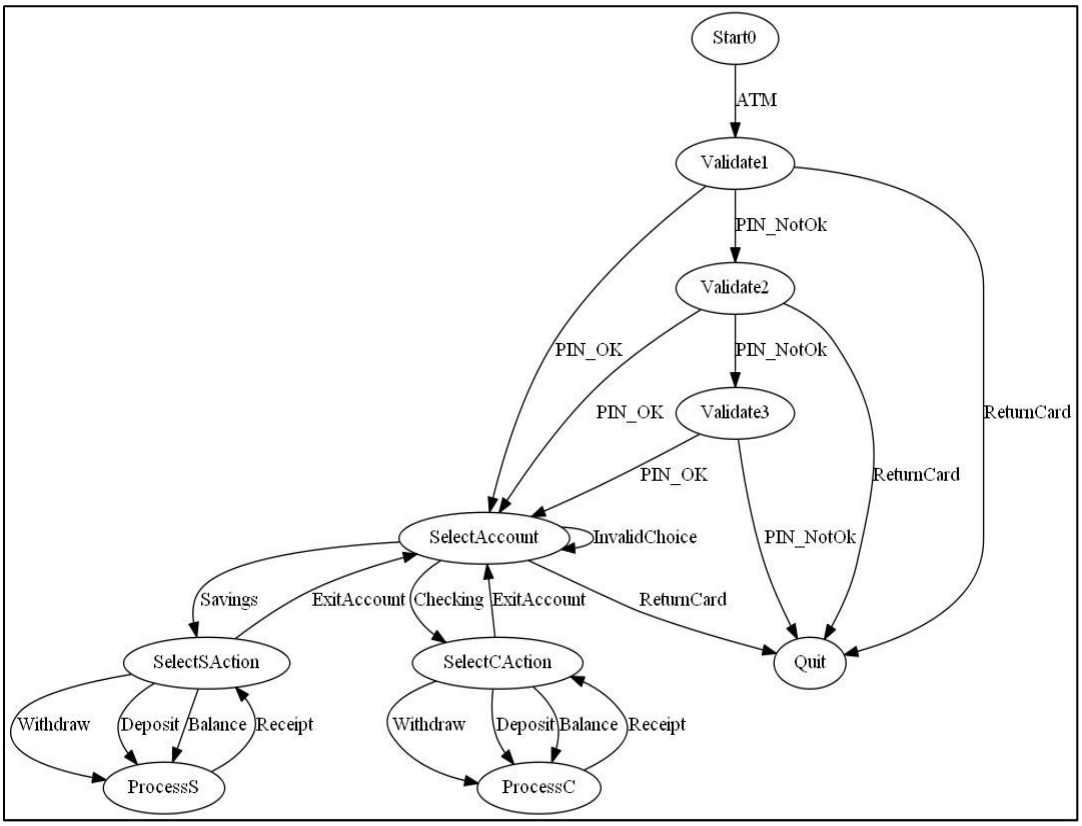
This appendix the finite state machines diagrams for the experimental objects that we used in our empirical study.

A.1 Cruise Control

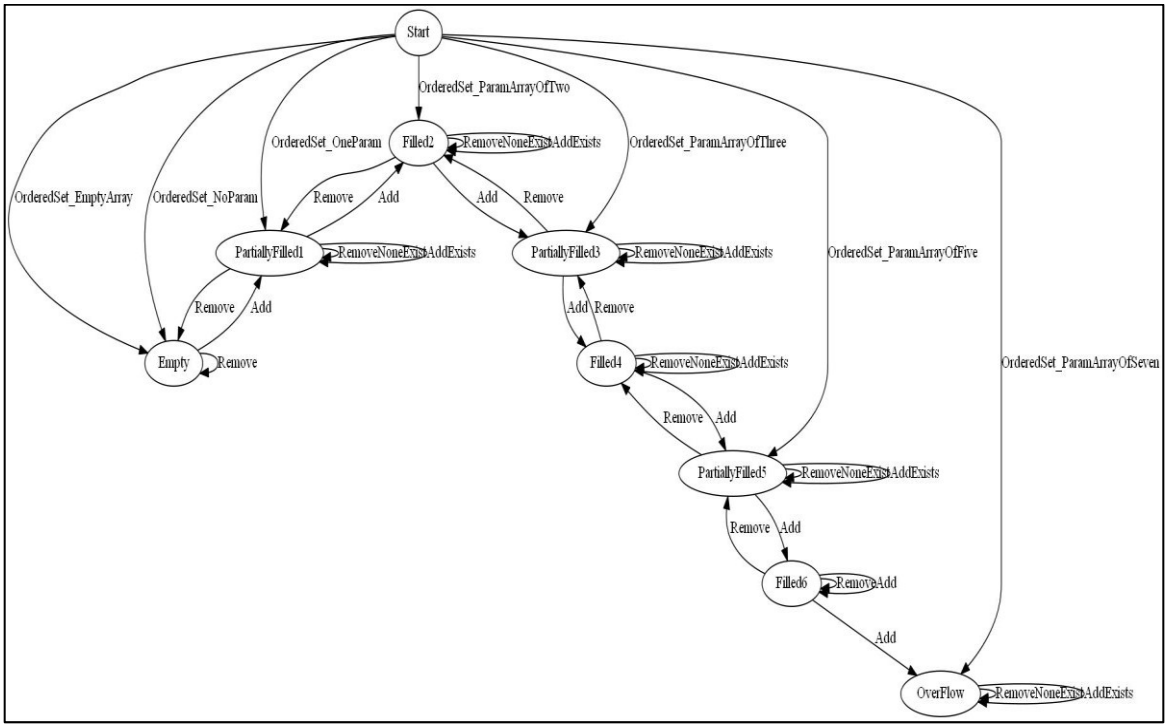


2→accelerator
3→brake
4→off
5→resume
6→engineOff
7→on
9→resume
10→brake
11→accelerator
12→off
13→engineOn
16→resume
17→on
18→engineOn
23→engineOn
24→accelerator
25→brake
26→off

A.2 ATM

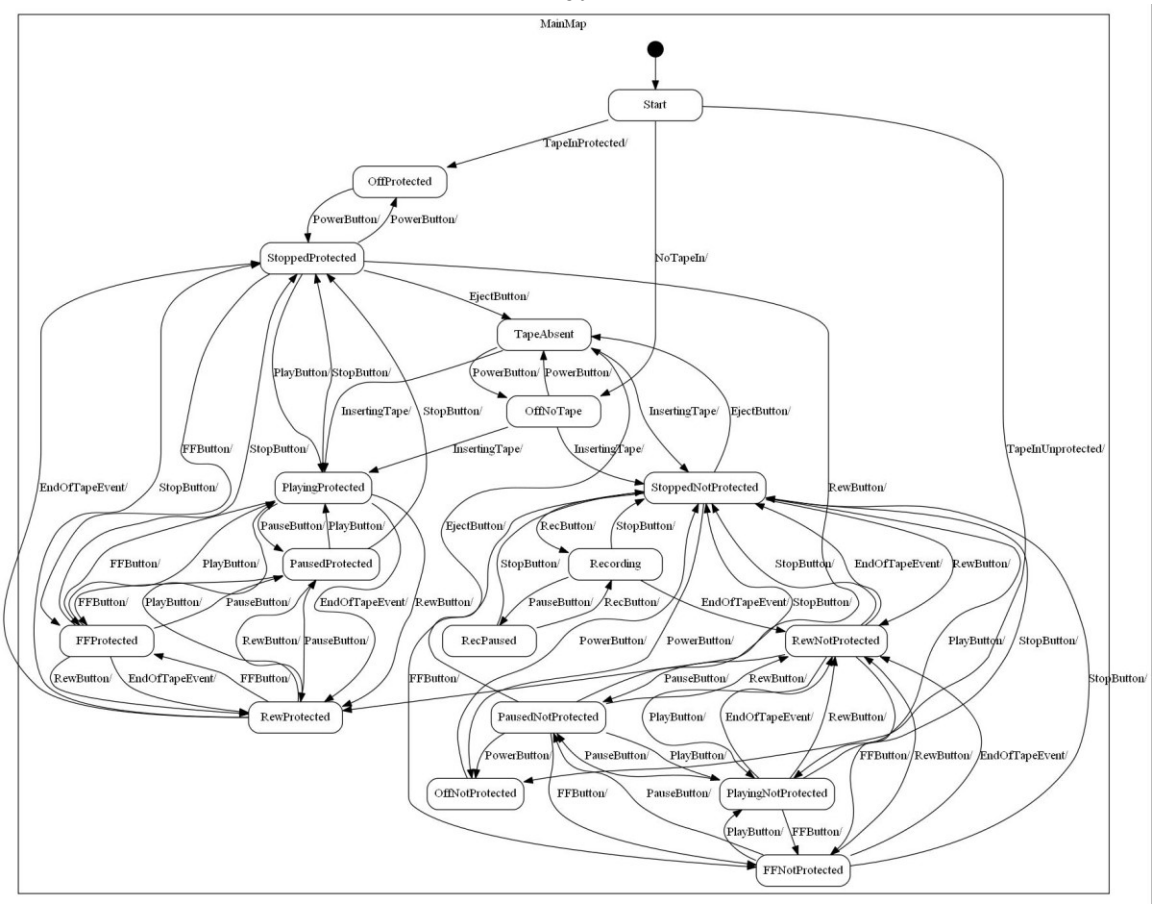

Start0
ATM
Validate1
PIN_NotOk
Validate2
PIN_OK
PIN_NotOk
PIN_OK
Validate3
ReturnCard
PIN_OK
ReturnCard
SelectAccount
InvalidChoice
PIN_NotOk
Savings
ExitAccount
Checking
ExitAccount
ReturnCard
SelectSAction
SelectCAction
Quit
Withdraw
Deposit
Balance
Receipt
Withdraw
Deposit
Balance
Receipt
ProcessS
ProcessC

A.3 Ordered Set



A.4 VCR

80


MainMap
Start
TapeInProtected/
NoTapeIn/
OffProtected
PowerButton/
PowerButton/
StoppedProtected
EjectButton/
TapeAbsent
PowerButton/
PowerButton/
OffNoTape
InsertingTape/
EjectButton/
InsertingTape/
InsertingTape/
TapeInUnprotected/
PlayingProtected
StoppedNotProtected
RewButton/
PauseButton/
PlayButton/
PausedProtected
EjectButton/
RecButton/
StopButton/
Recording
StopButton/
EndOfTapeEvent/
RewButton/
FFButton/
PlayButton/
PauseButton/
EndOfTapeEvent/
RewButton/
PauseButton/
RecButton/
EndOfTapeEvent
StopButton/
FFProtected
RecPaused
PowerButton/
PowerButton/
RewNotProtected
PlayButton/
StopButton/
RewButton/
EndOfTapeEvent/
FFButton/
FFButton/
PauseButton/
RewButton/
StopButton/
RewProtected
PausedNotProtected
PlayButton/
EndOfTapeEvent/
RewButton/
PowerButton
PauseButton/
PlayButton/
FFButton/
RewButton/
EndOfTapeEvent/
OffNotProtected
FFButton/
PauseButton/
PlayingNotProtected
PlayButton/
FFButton/
FFNotProtected

Appendix B. State Invariants

Cruise Control				
state\variable	**ignition**	**speed control**	**pedal**	**speed**
Inactive	off	disabled	0	0
Active	on	enabled	-	0
Cruising	on	enabled	decremented	>0
Standby	on	disabled	incremented	0

ATM		
state\variable	**PIN**	**Number of Attempts**
Validate1	-	0
Validate2	invalid	1
Validate3	invalid	2
SelectAccount	valid	<4
SelectSAccount	valid	<4
SelectCAccount	valid	<4
ProcessS	valid	<4
ProcessC	valid	<4
Quit	valid	<4

Ordered Set					
state\variable	**resized times**	**Actual size**	**Last**	**Overflow**	**Set size**
Filled2	0	2	1	false	2
PartiallyFilled1	0	<set_size	0	false	>0
Filled4	1	4	3	false	4
PartiallyFilled3	1	<set_size	0	false	>0
Filled6	2	6	5	false	6
PartiallyFilled5	2	<set_size	0	false	>0
Empty	0	0	-1	false	0
Overflow	<=*max_accepted_resizes*	*max_set_size*	setsize-1	true	*max_set_size*

VC R					
State\variable	**Tape**	**TapePulledToDrum**	**TapeWriteProtected**	**tapePosition**	**TapePresent**
OffProtected	not null	-	true	-	true
StoppedProtected	-	true	true	-	true
TapeAbsent	null	false	-	-	false
PlayingProtected	not null	true	true	-	true
PausedProtected	not null	true	true	-	true
FFProtected	not null	true	true	<*maxtapepos*	true
RewProtected	not null	true	true	>*mintapepos*	true
OffNoTape	-	-	-	-	false
StoppedNotProtected	-	true	false	-	true
Recording	not null	true	false	<*maxtapepos*	true
RecPaused	not null	true	false	<*maxtapepos*	true
PausedNotProtected	not null	true	false	<*maxtapepos*	true
OffNotProtected	not null	-	false	-	true
RewNotProtected	not null	true	false	>*mintapepos*	true
PlayingNotProtected	not null	true	false	<*maxtapepos*	true
FFNotProtected	not null	true	false	<*maxtapepos*	true