Comparing Transition Trees Test Suites Effectiveness for Different Mutation Operators

Hoda Khalil
Systems and Computer Engineering
Carleton University
Ottawa Ontario Canada
hodakhalil@cmail.carleton.ca

Yvan Labiche
Systems and Computer Engineering
Carleton University
Ottawa Ontario Canada
hodakhalil@cmail.carleton.ca

ABSTRACT

Research demonstrated that faults seeded mutation using operators can be representative of faults in real systems. In this paper, we study the relationship between the different operators used to insert mutants in the fault domain of the system under test and the effectiveness of different state machine test suites at killing those mutants. We are particularly interested in the effectiveness of two interrelated state machine testing strategies at finding different types of faults. Those are the round-trip paths strategy and the transition tree strategy. Using empirical evaluation, we compare the effectiveness of more than two thousand unique test suites at killing mutants seeded using eight different mutation operators. We perform experiments on four experimental objects and provide qualitative analysis of the results. We conclude that neither of the two studied strategies is more effective than the other at killing a certain type of mutants. However, the structure of the finite state machine and the nature of the system under test affect the type of faults detected by the different testing strategies.

CCS CONCEPTS

• Software and its engineering → Software creation and management → Software verification and validation → Empirical software validation • Software and its engineering → Software creation and management → Software verification and validation → Formal software verification

KEYWORDS

finite state machine, round-trip paths, transition trees, mutation operator, mutation testing.

ACM Reference Format:

Hoda Khalil and Yvan Labiche. 2020. Comparing Transition Trees Test Suites Effectiveness for Different Mutation Operators. In *Proceedings of the 11th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '20), November 8–9, 2020, Virtual, USA. ACM, New York, NY, USA, 7 pages.* https://doi.org/10.1145/3412452.3423571

A-TEST '20, November 8-9, 2020, Virtual, USA.

ACM ISBN 978-1-4503-8101-7/20/11
https://doi.org/10.1145/3412452.3423571

1 INTRODUCTION

Faults seeded using mutation operators have been used in ample of research work to represent the real faults of the software system [1, 2, 3]. In a previous study [15], we used mutation-based evaluations to empirically compare the effectiveness of two finite state machine (FSM) testing strategies; the test suites generated using complete round-trip path (RTP) testing and transition tree testing [5, 6]. We used in our empirical evaluation four experimental objects and 2,057 unique test suites [12]. We concluded that completely covering RTPs was not equivalent to the piece-wise coverage achieved through transition trees in terms of fault (mutant) detection. Also, we compared different types of transition trees using three generation algorithms: breadth first search (BFS), depth first search (DFS), and random traversal of the original FSM. We compared the test suites produced using the different algorithms to one another and the complete RTP test suite. The results obtained showed that DFS test suites could sometimes achieve better fault detection than other test suites. The experiments also showed that in some cases the complete RTP test suite detected faults that were not detected by transition trees test suites that covered RTPs in pieces. Although we used mutants seeding as a way to measure the effectiveness of the test suites, we did not inspect whether any of the examined strategies is more effective than the others at finding specific types of faults (seeded mutants) [14]. The large amount of data collected, in the thousands of test suites [15], was a valuable data set that is worthy of more analysis. Furthermore, the conclusion that RTP test suites detected faults that were not detected by other test suites was a finding that deserved further research. In this paper, we are interested to answer the research question of whether any of the four types of test suites (BFS, DFS, Random, and complete RTP) we generate are more effective at killing mutants that are seeded using specific mutation operators. We reuse the results of executing the test suites and perform further analysis of the obtained results to answer this research question. Such results can be useful for specifying which testing strategy can be used for certain categories of software systems.

We start the discussion by summarizing some background concepts (section 2). In sections 3, we review the literature. In

section 4, we introduce the experimental setup that we used in previous work and this study. Then, in sections 5, we present the results and provide qualitative analysis to inspect the capability of the different test suites at killing each type of mutant separately. Finally, we discuss the threats to validity (section 6) and the conclusions we achieved by analyzing the results (section 7).

2 BACKGROUND

This section defines the main concepts that we use in this research. This includes the type of FSMs we use, testing strategies of concern, and mutation-based evaluation that is used as a surrogate measure of fault detection.

2.1 FSM Testing

FSM modeling is a formal, rigorous, and simple technique that has been used extensively to model software artifacts [4]. We focus on FSMs that are deterministic, connected, and completely specified. For our experimental objects, we augment the original FSM specifications for completeness when this is necessary. For an unspecified input in the original FSM, we assume that the machine produces a null output and remains in the current state, which is a typical assumption for FSM modeling [17].

Chow [6] and Binder [5] are important contributors to the FSM testing literature. Chow introduces the W-method where he generates a test suite from an FSM using a BFS traversal of the FSM graph. Each path in the tree from the initial/root node to a terminal node, a.k.a. transition tree path is considered a test sequence, while the collection of paths composes a single test suite [6]. Binder, on the other hand, introduces RTP testing. To exercise RTPs, Binder also suggests deriving a transition tree from the FSM graph. The resulting tree covers each RTP either in its entirety (i.e., the RTP appears as a proper sub-path of a transition tree path) or in pieces [5]. Binder's method implies that covering RTPs in their entirety, which we refer to as complete RTPs, is equivalent from a testing point of view to covering RTPs, possibly in pieces, through a transition tree. In our previous publications, we demonstrated that the two are not equivalent at finding faults. In this research work, we examine the performance of the two strategies for each mutation operator separately.

2.2 Test Strategies

Many testing strategies can be applied to state-based testing. A state-based testing strategy that proved to provide a balance between effectiveness at finding faults and cost is the RTP criterion [9]. This criterion is satisfied when all RTPs (i.e., loops) are exercised [5]. An RTP is a simple path (no repeated node) that starts and ends with the same node [5]. A closely related criterion is the transition tree criterion [6]. For transition trees, FSMs are treated as directed graphs (digraph) where states are nodes and transitions are edges. A transition tree is a spanning tree of the FSM graph. Since, in our work, we aim to cover all possible operations that move the system under test (SUT) from one state to another (transition), or maybe to the same state, we deal with a special kind of spanning trees. Our spanning/transition trees must include all edges/transitions that are present in the FSM/digraph. Therefore, sometimes the states are repeated in the resulting tree. A test suite is constructed from the resulting tree: Each path in the spanning tree from the root node to a terminal node is considered a test sequence, and the collection of test sequences is a test suite. This way, the transition tree criterion subsumes all transitions criterion [1].

2.3 Mutation-Based Evaluation

To investigate the effectiveness of the different testing strategies, we use mutation operators. This technique has been used in many studies [2, 7, 8, 19]. We use Major, a compiler integrated automatic framework, to seed faults and analyze fault detection for our test suites [10]. Major supports a set of commonly applied mutation operators and excludes operators that have been shown to generate redundant mutants. Major 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 [10]. Table 1 lists the mutation operators supported by major and illustrates them by examples. Major determines mutation coverage, which is the number of mutants that are covered (i.e., the line of code where they are seeded is executed) by the executed test suite. 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.

Table 1: Types of mutants implemented by Major.

Operator	Description	Example(s)
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

3 RELATED WORK

Many studies have been dedicated to comparing the different methods for state-based testing using mutants [7, 8, 19]. We include in this paper the studies that are most related to the work we present in this paper. We refer the reader to chapter 2 of our previous work for a more detailed literature review of other studies that compare FSM testing strategies [12].

Endo and Simão [7] carry an experimental study to compare different state machine testing strategies that are mainly improvements of the W method [6]. The comparison considers the 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 testing. The authors discuss in detail the mutation operators used, but they do not compare the effectiveness of methods for each operator [7].

Also, Mouchawrab and colleagues [19] use mutation testing to compare RTP testing to structural (white box) testing. 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. Mutants are seeded in three experimental objects using MuJava [18]. However, the types of operators used are not discussed.

Simão and Petrenko compare FSM coverage criteria; mainly state coverage (SC), transition coverage (TC), initialization fault (IF), and transition fault (TF) coverage. The criteria under experimentation are also measured for finding faults introduced in the FSM specifications. The study also lacks discussing the effectiveness in relation to the type of mutation operators used [20].

In this research, we aim at studying the FSM testing strategies' effectiveness for groups of mutants seeded using each of the applicable mutation operators (see table 1). We discuss how the operator used to insert each group of mutants affects the capability of each algorithm (i.e. BFS, DFS, RTP, and random test suites) at killing the inserted mutant.

4 EXPERIMENTAL DESIGN

We use the same experimental setup used in our previous work. The exact steps and configuration required to replicate or perform similar experiments are available online [16] and documented in chapter 5 of our previous publication [12]. In this section, we briefly present the experimental objects we use, the process used to generate and execute the test suites (section 4.1), some test suites specifications (section 4.2), and how we handle equivalent mutants (section 4.3).

4.1 Experimental Objects and Process

In our study, we use four experimental objects. The FSM and consequently the code of the four systems have different structures and sizes [15]. The four experimental objects are an embedded controller of a cruise control simulator, an automated teller machine (ATM), an ordered set data structure, and an electromechanical device of a videocassette recorder VCR. Table 2 summarizes the characteristics of the different objects we use. The FSM diagrams of the objects can be found in appendix A of our previous publications [12]. The experimental process can be divided into four steps: generating test suites (BFS, DFS, Random, and RTP), producing JUnit files (one for each test suite), seeding faults in the code of the SUT, and running the JUnit tests using the faulty versions of the code. Finally, we measure the mutation score and analyze the data (section 5). We refer the reader to a previous work of the details of the steps we followed to automatically generate 102,113 test suites and sample them to produce the 2,057 unique test suites we used in this experiment [12, 13].

Table 2: Characteristics of the four experimental objects.

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

4.2 Test Suites

For each experimental object, we used ten unique randomly generated test suites in addition to all possible BFS, DFS, and complete RTP test suites. Using only one random tree is not sufficient for statistical analysis. Besides, having one random tree might result in a specific BFS or DFS tree which could be much better (or worse) than others and which can mislead the comparison process. On the other hand, we chose not to increase the number of random trees to avoid the possibility of duplicate trees. We tested generating random trees for all the experimental objects and the algorithm produced unique trees [12].

It is important to note that the RTP test suites we use are different from what Binder suggests as they do not include simple paths [5]; Binder adds simple paths to exercise transitions of the FSM that are not in RTPs, thereby ensuring that the RTP test suite exercises all transitions. As a result of our decision, unlike Binder's RTP test suites, the RTP test suites we use are not guaranteed to exercise all transitions. This ensures we can study the effectiveness of exercising the complete RTPs in isolation from the other (simple) paths that can contribute to the effectiveness of the test suites and mislead the results; we do not want to include the effectiveness of simple paths as we are only interested in RTPs. Similarly, to minimize the contribution of prefixes (from the initial state to states starting RTPs) to fault detection, the prefixes are the shortest paths. The test oracle we use checks the states reached by the actual SUT and compares the values of the state variables to the expected state after executing each transition. If the actual state of the SUT does not conform to the expected state, an exception is issued, and the mutant is considered killed.

4.3 Handling Equivalent Mutants

Equivalent mutants 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 [11]. 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 mutant. 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 whereas an equivalent mutant is necessarily test-equivalent for any test suite. Figure 1 illustrates these notions; it shows three test suites as an illustrative example. For example, *X* is a test-equivalent mutant for test suites T2 and T3, but it is not test-equivalent with respect to test suite T1. Therefore, X is not an equivalent 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 nor 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 different from the state expected by the requirements. M also is an equivalent mutant, but one that has not been covered by any test suite. In our experiments, we consider that an equivalent mutant is a mutant that is test-equivalent with respect to all 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.

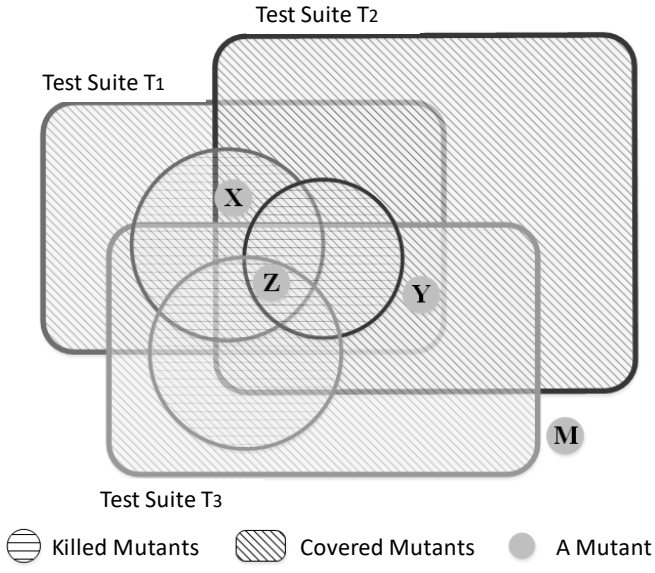

Figure 1: Covered, killed, test-equivalent, and equivalent mutants.

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

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

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 the scale of the measurement (mutants) not a matter of validity.

Given the proof above and the large to the very large number of test suites the study 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.

5 RESULTS AND ANALYSIS

Table 3 categorizes the number of seeded mutants and the percentage of killed mutants, by operator, for each type of test suite (BFS, DFS, Random, and RTP) for each object.

The percentage of killed mutants by each type of test suite is the collection of mutants killed by at least one test suite belonging to this type. Table 3 only shows the operators (from Table 1) that Major can 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 test suites. We attribute this to the much larger number of DFS test suites for VCR. We also observe from Table 3 that for all objects except cruise control, all the test suites perform best on EVR mutants. In the cruise control, the COR mutants rank top in terms of being detected. On the one hand, Major inserts only four COR mutants, and all of them are in conditional statements that evaluate the state of the SUT. Only two of them do not affect the

destination state of the object and hence are not killed by any of the traversal algorithms. On the one hand, the EVR mutants that replace statements not affecting the state of the system are not killed. For example, if the cruise control is in the state cruising and the EVR mutant changes the speed (a state variable in the cruise control object) but does not set it to zero, then the state of the system does not change. Therefore, the mutant is not killed (recall that, as discussed in section 4.1, oracles use the state variables to compare to what is expected in the original specifications of the system). 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 personal identification number (PIN) or changing the number of attempts to enter a valid PIN. 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 since almost all the arithmetic operations in the cruise control are related to the timing aspect of the system that is neither reflected in the FSM model nor in the state variables that the oracle uses (the FSM is an abstraction).

Figure 2 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 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.

All algorithms perform similarly for all operators for the cruise control and the ATM, except for two cases. 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 (see [12]). RTP test suites include all complete RTPs in addition to the shortest prefixes leading to the RTPs. As mentioned earlier, in our study we are interested in examining the capability of complete RTPs at finding faults that are not detected by other algorithms. 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 evaluate RTPs in isolation. Also, note that as mentioned earlier, Binder's algorithm does not necessarily cover all complete RTP paths in the FSM completely. Therefore, when building the RTP test suite, the structure of the FSM matters. The ATM FSM is not complete [12]; as a result, as discussed earlier, constructing the RTP test suite (with RTPs and prefix) misses some transitions. One consequence of this is that the RTPs together with their prefixes for the ATM test suite exercise all the arithmetic operators (AOR) related to calculating balance, amount to be withdrawn, etc., whereas most mutants that belong to relational operators (ROR) are alive because they are seeded in pieces of code that implement transitions that are missed by the RTP test suite.

Table 3: Mutation scores for each mutation operator.

Experimental Object	Mutant		AOR	COR	ROR	STD	LVR	EVR
Cruise Control	Inserted		68	4	86	70	97	11
	Total Killed		1.4%	50.0%	18.6%	27.1%	18.5%	27.2%
	Killed by	BFS	0.0%	50.0%	18.6%	27.1%	19.5%	27.2%
		DFS	1.4%	50.0%	18.6%	27.1%	18.5%	27.2%
		Random	0.0%	50.0%	18.6%	27.1%	18.5%	27.2%
		RTP	0.0%	50.0%	18.6%	27.1%	19.5%	27.2%
ATM	Inserted		16	12	17	205	42	27
	Total Killed		50.0%	58.3%	35.2%	26.8%	52.3%	70.3%
	Killed by	BFS	50.0%	58.3%	35.2%	26.8%	26.1%	70.3%
		DFS	50.0%	58.3%	35.2%	26.8%	26.1%	70.3%
		Random	50.0%	58.3%	35.2%	26.8%	26.1%	70.3%
		RTP	50.0%	33.3%	11.7%	22.4%	19.0%	55.5%
Ordered Set	Inserted		180	94	171	99	166	17
	Total Killed		42.7%	40.4%	35.0%	30.3%	40.3%	52.9%
	Killed by	BFS	40.5%	38.3%	33.3%	29.2%	38.5%	52.9%
		DFS	40.0%	40.4%	33.3%	29.2%	38.5%	52.9%
		Random	42.2%	38.3%	34.5%	30.3%	39.7%	52.9%
		RTP	38.8%	37.2%	33.3%	28.2%	37.9%	52.9%
VCR	Inserted		264	132	242	498	192	141
	Total Killed		56.8%	50.0%	67.7%	65.2%	50.0%	63.8%
	Killed by	BFS	0.7%	12.1%	15.2%	20.0%	3.6%	31.2%
		DFS	56.8%	48.4%	67.3%	60.4%	48.9%	60.9%
		Random	1.1%	12.1%	15.2%	20.0%	4.1%	31.2%
		RTP	0.7%	9.0%	7.0%	8.4%	3.1%	14.1%

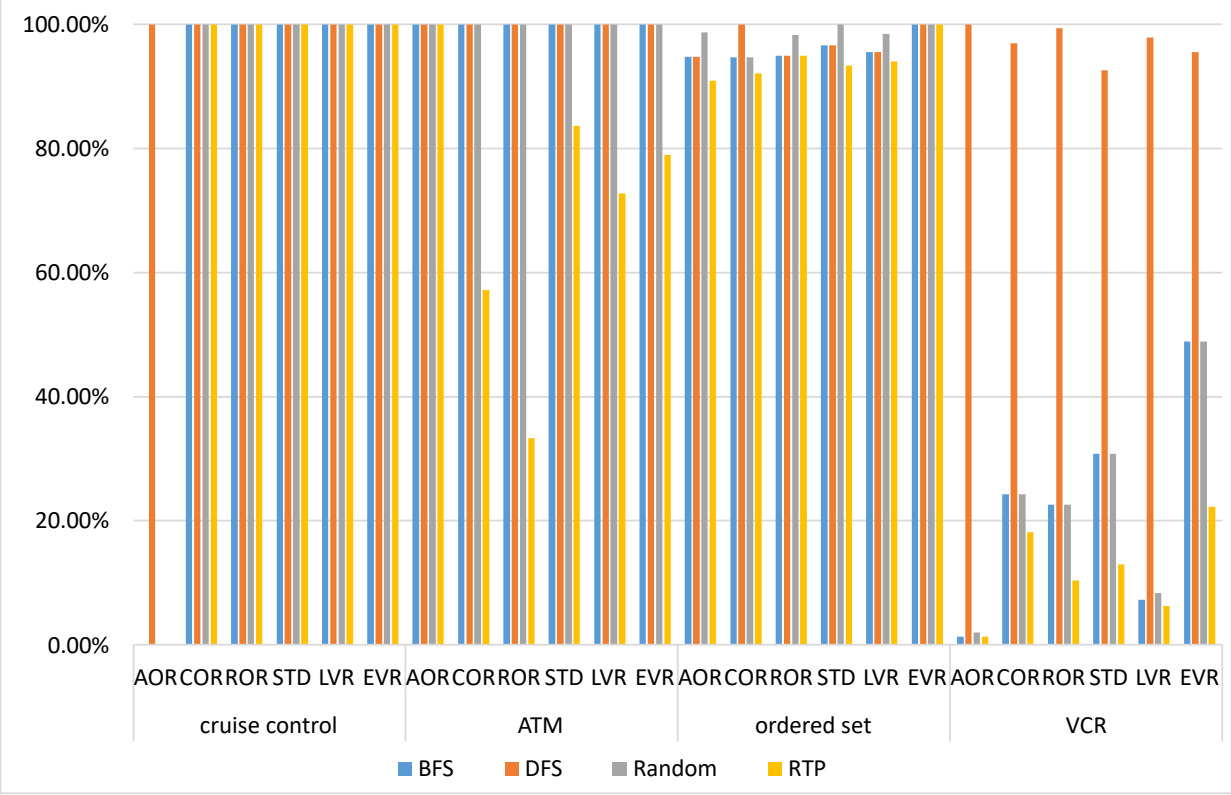

Figure 2: Algorithms performance for each mutation operator.

Second, the DFS test suites of the cruise control are the best performers for the AOR operator. 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. Although the number of DFS test suites is higher for the ordered set than the number of BFS test suites, both groups of test suites mostly perform the same. However, random test suites perform better. This is possible because the collection of random test suites includes BFS and DFS test suites.

As explained above, an RTP test suite does not necessarily cover all transitions. Nevertheless, for the VCR it performs relatively well for all operators when compared to the other test suite generation algorithms. This is due to the high connectivity of the VCR FSM.

6 VALIDITY THREATS

We acknowledge that, as in any other experimental study, there are some threats to the validity of our results. Assuming that a mutant, that is covered and alive for all test suites, is an equivalent mutant is a minor construct validity threat. Exercising large numbers of test suites make this assumption reasonable. Besides, given that we are comparing the test suites and not evaluating each test suite in isolation makes mutants that are equivalent to all test suites exercised irrelevant to the results obtained. Detailed analysis and proof that this threat is limited due to the way we set up our experiments are discussed in section 4.3.

Whether the seeded mutants are representative of state faults or not is a legitimate question. Nevertheless, the test oracle that we use is designed to detect state faults only. The oracle throws an exception if 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. 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. For example, in the ordered set object, the SUT checks if the data structure is not overflowing, then it adds the new element to the data structure. In the fault domain, where Major seeds mutants, the mutant replaces the if statement so that the data structure is always assumed to be overflowing. A COR mutant sets the overflow variable to be always true. This means that when an event to add an element is triggered, for instance, the system ignores this event since it assumes that the current state is an overflow state and no additional elements can be added. The equivalent state fault for this mutant is "Missing or incorrect event that results in a valid event being ignored". This mutant is killed by all test suites.

In brief, we confirm the presence of a few validity threats. However, we believe that they are minor and do not affect our results.

7 CONCLUSION

In the presented empirical study, we used four experimental objects to compare the effectiveness of transition trees and complete RTP FSM testing strategies at finding specific types of faults using different mutation operators. We compared how 2057 test suites of different types (BFS, DFS, random, and complete RTP) perform in terms of killing seeded mutants using eight mutation operators. We empirically compared the performance of the test suites for each operator and presented the results. From the qualitative analysis of the presented results, we conclude that the type of faults detected is not related to the testing strategy (BFS, DFS, Random, or RTP), but rather related to the SUT and the aspects of the SUT modeled by the FSM. In other words, depending on the nature of the SUT, there might be a higher probability of killing a specific type of mutant.

REFERENCES

[1] Paul Amman and Jeff Offutt. 2016. *Introduction to Software Testing* (2nd. ed.). Cambridge University Press, New York, NY. DOI: https://10.1017/9781316771273

[2] James H. Andrews, Lionel C. Briand, and Yvan Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments? In *Proceedings of the 27th International Conference of Software Engineering (ICSE)*, May 15-21, 2005, St. Louis, MO, 402–411. DOI: https://doi.org/10.1145/1062455.1062530

[3] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *IEEE Trans. Softw. Eng.* 32, 8 (Sept. 2006), 608–624. DOI: https://doi.org/10.1109/TSE.2006.83

[4] Boris Beizer. 1990. *Software Testing Techniques* (2nd. ed.). Van Nostrand Reinhold Co., New York, NY.

[5] Robert V. Binder. 1999. *Testing Object-Oriented Systems: Models, Patterns, and Tools* (1st. ed.). Addison-Wesley Professional, Boston, MA.

[6] Tsun S. Chow. 1978. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.* SE-4, 3 (May 1978), 178–187. DOI: https://doi.org/10.1109/TSE.1978.231496

[7] Andre Takeshi Endo and Adenilso Simao. 2013. Evaluating Test Suite Characteristics, Cost, and Effectiveness of FSM-Based Testing Methods. *Inf. Softw. Technol.* 55, 6 (June 2013), 1045–1062. DOI: https://doi.org/10.1016/j.infsof.2013.01.001

[8] Nina Elisabeth Holt, Lionel C. Briand, and Richard Torkar. 2014. Empirical Evaluations on the Cost-Effectiveness of State-Based testing: an Industrial Case Study. *Inf. Softw. Technol.* 56, 8 (Aug. 2014), 890–910. DOI: https://doi.org/10.1016/j.infsof.2014.02.011

[9] Nina Elisabeth Holt, Richard Torkar, Lionel Briand, and Kai Hansen. 2012. State-Based Testing: Industrial Evaluation of the Cost-Effectiveness of Round-Trip Path and Sneak-Path Strategies. In *Proceedings of the IEEE 23rd International Symposium of Software Reliability Engineering (ISSRE)*, November 27-30, 2012, Dallas, TX, 321–330. DOI: https://doi.org/10.1109/ISSRE.2012.17

[10] Rene Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 6-10, 2011, Lawrence, KS, 612–615. DOI: https://doi.org/10.1109/ASE.2011.6100138

[11] Rene. Just, Michael D. Ernst, and Gordon Fraser. 2014. Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 21-25, 2014, San Jose, CA, 315–326. DOI: https://doi.org/10.1145/2610384.2610388

[12] Hoda A. Khalil. 2018. FSM Testing Based on Transition Trees and Complete Round Trip Paths Testing Criteria. Ph.D. Dissertation. Carleton University, Ottawa, ON, Canada.

[13] Hoda Khalil and Yvan Labiche. 2017. Finding All Breadth First Full Spanning Trees in a Directed Graph. In p*roceedings of the 41st International Computer Software and Applications Conference (COMPSAC)*, July 4-8, 2017, Turin, Italy, 372–377. DOI: https://doi.org/10.1109/COMPSAC.2017.128

[14] Hoda Khalil and Yvan Labiche. 2017. State-Based Tests Suites Automatic Generation Tool (STAGE-1). In *Proceedings of the 41st International Computer Software and Applications Conference (COMPSAC)*, July 4-8, 2017, Turin, Italy, 357–363. DOI: https://doi.org/10.1109/COMPSAC.2017.221

[15] Hoda Khalil and Yvan Labiche. 2017. On FSM-Based Testing: An Empirical Study: Complete Round-Trip Versus Transition Trees. In *Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, October 23-26, 2017, Toulouse, France, 305–315. DOI: https://doi.org/10.1109/ISSRE.2017.34

[16] Hoda Khalil and Yvan Labiche. 2017. FSM-Testing-Transtiontion-Trees-and-RTP. Retrieved from https://github.com/hakhalil/FSM-Testing-Transtiontion-Trees-and-RTP.

[17] David Lee and Mihalis Yannakakis. 1996. Principles and Methods of Testing Finite State Machines - A Survey. *Proc. IEEE.* 84, 8 (August 1996), 1090–1123. DOI: https://doi.org/10.1109/5.533956

[18] Yu Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An Automated Class Mutation System. *Softw. Test. Verif. Reliab.* 15, 2 (June 2005), 97–133. DOI: https://doi.org/10.5555/1077303.1077304

[19] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. 2011. Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments. *IEEE Trans. Softw. Eng.* 37, 2 (March-April 2011), 161–187. DOI: https://doi.org/10.1109/TSE.2010.32

[20] Adenilso Simão and Alexandre Petrenko. 2010. Fault Coverage-Driven Incremental Test Generation. *Comput. J.* 53, 9 (November 2010), 1508–1522. DOI: https://doi.org/10.1093/comjnl/bxp073