Using Rule Mining for Automatic Test Oracle Generation

Alejandra Duque-Torres[a], Anastasiia Shalygina[a], Dietmar Pfahl[a] and Rudolf Ramler[b]

[a]Institute of Computer Science, University of Tartu, Tartu, Estonia
[b]Software Competence Center Hagenberg GmbH, Hagenberg, Austria

Abstract

Software testing is essential for checking the quality of software but it is also a costly and time-consuming activity. The mechanism to determine the correct output of the System Under Test (SUT) for a given input space is called test oracle. The test oracle problem is a known bottleneck in situations where tests are generated automatically and no model of the correct behaviour of the SUT exists. To overcome this bottleneck, we developed a method which generates test oracles by comparing information extracted from object state data created during the execution of two subsequent versions of the SUT. In our initial proof-of-concept, we derive the relevant information in the form of rules by using the Association Rule Mining (ARM) technique. As a proof-of-concept, we validate our method on the Stack class from a custom version of the Java Collection classes and discuss the lessons learned from our experiment. The test suite that we use in our experiment to execute the different SUT version is automatically generated using Randoop. Other approaches to generate object state data could be used instead. Our proof-of-concept demonstrates that our method is applicable and that we can detect the presence of failures that are missed by regression testing alone. Automatic analysis of the set of violated association rules provides valuable information for localizing faults in the SUT by directly pointing to the faulty method. This kind of information cannot be found in the execution traces of failing tests.

Keywords

Software testing, test oracle, association rule mining, test oracle automation, machine learning methods in software testing

1. Introduction

Software testing is an essential activity for quality assurance in software development process as it helps ensure the correct operation of the final software [1]. However, software testing has historically been recognised to be a time-consuming, tedious, and expensive activity given the size and complexity of large-scale software systems [2]. Such cost and time involved in testing can be managed through test automation. Test automation refers to the writing of special programs that are aimed to detect defects in the System Under Test (SUT) and to using these programs together with standard software solutions to control the execution of test suites. It is possible to use test automation to improve test efficiency and effectiveness.

Software testing, automated or not, has four major steps: test case generation, predicting the outcomes of the test cases, executing the SUT with the test cases to obtain the actual outcome, and comparing the expected outcome against the actual outcome to obtain a verdict (pass/fail) [3]. In these steps there are two major challenges: find effective test inputs, *i.e.*, inputs that can reveal faults in the SUT, and determine what should be the correct output after execution of the test cases. The second challenge refers to the the *test oracle problem*. A test oracle is a mechanism that determines the correct output of SUT for a given input [4]. Although substantial research has been conducted to provide test oracle automatically, apart from model-driven testing, the oracle problem is largely unsolved.

Motivated by the above, we developed a method to derive test oracles based on information contained in object state data produced during the execution of the SUT. Object state data is the set of the values of all defined attributes of an object at a certain point of time. We assume that most programs have objects with a mutable state, and the execution of methods can modify the state of the program. The idea of using the state information roots in the assumption that relations contained in the state data when testing a new version of the SUT should remain unchanged as compared to a previous version.

Our proposed method employs Association Rule Mining (ARM). In our context, the purpose of ARM is to mine interesting relations in the state data of the SUT. ARM is an unsupervised machine learning (ML) method [5]. The algorithms used in ARM attempt to find relationships or associations between categorical variables in large transactional data sets [6]. In particular, we were interested in understanding whether the information provided by the resulting model can help to verify the correct operation of new versions of the SUT to which we normally apply existing tests for regression testing. More specifically, we wonder if we can detect and locate faults in new ver-

QuASoQ'20: 8th International Workshop on Quantitative Approaches to Software Quality, December 1, 2020, Singapore
duquet@ut.ee (A. Duque-Torres);
anastasiia.shalygina@gmail.com (A. Shalygina);
dietmar.pfahl@ut.ee (D. Pfahl); rudolf.ramler@scch.at (R. Ramler)
0000-0002-1133-284X (A. Duque-Torres); 0000-0003-2400-501X (D. Pfahl); 0000-0001-9903-6107 (R. Ramler)

sions of the SUT. We tackle our goal by answering the following research questions:

RQ1: *How effective is the rule mining approach?*. This research question investigates to what extent ARM is able to detect failures.

RQ2: *What information regarding fault localisation can the method offer?* This research question explores to what extend the information contained in association rules helps developers locate faults in the code.

In our experiments, we use the *Stack Class* of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate.

2. Association Rule Mining

ARM is a rule-based unsupervised ML method that allows discovering relations between variables or items in large databases. ARM has been used in other fields, such as business analysis, medical diagnosis, and census data, to find out patterns previously unknown [6]. The ARM process consists of at least two major steps: finding all the frequent itemsets that satisfy minimum support thresholds and, generating strong association rules from the frequent derived itemsets by applying minimum confidence threshold.

A large variety of ARM algorithms exist. [7]. In our experiments, we use the Apriori algorithm from Python3 Efficient-Apriori library [8]. It is well known that the Apriori algorithm is exhaustive, that is, it finds all the rules with the specified support and confidence. In addition, ARM doesn't require labelled data and is, thus, fully unsupervised. Below we define important terminology regarding ARM:

Itemset: Let $I = \{X, \ldots, Y, Z\}$ be a set of different items in the dataset D. Itemset is a set of k different items.

Association rule: Consider a dataset D, having n number of transactions containing a set of items. An association rule exposes the relationship between the items.

Support: The support is the percentage of transaction in the dataset D that contains both itemsets X and Y. The support of an association rule $X \rightarrow Y$:

$support(X \rightarrow Y) = support(X \cup Y) = P(X \cup Y)$

Confidence: The confidence is the percentage of transactions in the database D with itemset X that also contains the itemset Y. The confidence is calculated using the conditional probability which is further expressed in terms of itemset support: $confidence(X \rightarrow Y) = P(Y|X) = support(X \cup Y)/support(X)$

Lift: Lift is used to measure frequency X and Y together if both are statistically independent of each other. The lift of rule $(X \rightarrow Y)$ is defines as $lift(X \rightarrow Y) = confidence(X \rightarrow Y)/support(Y)$.

A lift value one indicates X and Y appear as frequently together under the assumption of conditional independence.

3. Method

Figure 1 presents an overview of the method for rule-mining based tests oracles generation. Overall, the proposed method comprises two phases. *Phase I* is responsible for the ruleset generation, *i.e.*, the rule mining part. The output of this phase is the ruleset. *Phase II* is in charge of applying the ruleset to the new SUT versions. Thus, the output of this phase could be seen as a fault report for new SUT versions. Below we describe them in detail:

3.1. Phase I - Rule Set Generation

Phase I starts with the extraction of the state data. Then, feature selection and encoding are performed so that all the features become appropriate to use for the rule mining. The features should be encoded as categorical if there is a need. When all the required operations with the raw data are performed, the state data from the first version of SUT is received, and the rule mining process starts. After that, one gets a set of rules. All the process can be split into three main steps which are detailed below:

Step 1.1 - State data acquisition: This step comprises two activities:

Activity 1.1.1 (Produce test) is responsible for the generation of tests. To perform this activity, we use Randoop to generate unit tests automatically. Randoop is a popular random unit test generator for Java[1]. Randoop creates method sequences incrementally by randomly selecting a method call to apply and using arguments from previously constructed sequences. When the new sequence has been created, it is executed and then checked against contracts. Two important parameters that we use in our experiments are test limit and a random-seed. The test limit parameter helps to limit the number of tests in the test suite. The random seed parameter allows us to produce multiple different test suites since Randoop is deterministic by default. Therefore, these two parameters allow us to generate many test suites of different size containing various test cases.

Activity 1.1.2 (Execute the test suite) is responsible for state tracking and saving raw data. To track the states of the SUT while running the test suite and save it to the text file for later analysis, we built a special program that helps to track and save the information of the state of the SUT. We call this program *Test Driver*. The idea behind the Test Driver is that the methods of the SUT

[1] https://randoop.github.io/randoop/

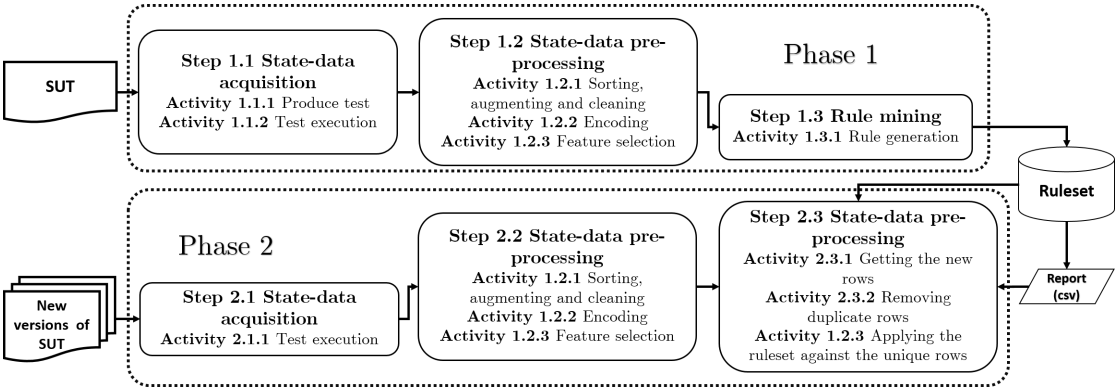

Figure 1: Overview of the method for rule-mining based test oracles generation

can be logically divided into two categories: methods that change a state of the class instance of the SUT, and methods that reflect the state. These methods are so-called state methods. The test driver tracks and stores the information returned by state methods if they are called immediately after the test case execution. The information is saved in a CSV file.

Step 1.2 - State data pre-processing: This data pre-processing step is made up of three activities:

Activity 1.2.1 (Sorting, augmenting and cleaning) is responsible for ensuring that the data is correct, consistent and useable. This activity has three main functions: *sort*, *aug*, and *clean*. The *sort* function is responsible for sorting the dataset based on the TestId and InstanceId, this is done to find interesting sequences in the data, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. *e.g.*, it is possible to add characteristics that indicate the previous state. This is made trough *aug* function. The *clean* function removes the rows that are no needed or inconsistent rows.

Activity 1.2.2 (Encoding) is in charge of preparing the data according to the requirements of the rule mining algorithm. For example, Apriori [9], which is the algorithm used in this paper, works only with categorical features. Thus, Activity 1.2.2 categorises and generalises the numerical inputs into string representations.

Activity 1.2.3 (Feature selection) is an extra activity which allows to explore the different performance of the method when different features are used. For instance, in this paper we used five different datasets which contain different numbers of features.

Step 1.3 - Rule mining: This step is responsible for generating the set of rules by using the Apriori ARM algorithm.

3.2. Phase II

Phase II is in charge of applying the ruleset to the new versions. Like Phase I, Phase II comprises three steps

Step 2.1 - State data acquisition: Unlike Step 1.1, this step has only one activity, the test execution activity. In phase one, we assume that the first version of the SUT is correct; then, we build test suites using Randoop and the test driver. In Step 2.1, the same tests generated in Activity 1.1.1 are used to test the new versions of the SUT.

Step 2.2 - State data pre-processing: This step performs the same activities as Step 1.2 to prepare the state dataset of the new SUT version.

Step 2.3 - Apply the ruleset to the new SUT version: This step comprises three activities. *Activity 2.3.1* is in charge of comparing and selecting the rows that are exclusively different from the first version of the SUT. *Activity 2.3.2* removes duplicate rows. This is done for optimisation. If two or more rows are the same, then they will have the same results.

Activity 2.3.3 (apply the ruleset against the new unique rows) is responsible for applying the rule-set against the new unique rows. A rule will have two sides, *e.g.*, let's consider the rule $(X \rightarrow Y)$, in this rule, X is a left-hand side (LHS) of the rule, and Y is a right-hand side (RHS). The procedure for applying the ruleset against the new unique rows works as follows: $i)$ pick a rule from the set of rules, $ii)$ use LHS, $iii)$ select values which match LHS, $iv)$ check whether these values match the RHS, $v)$ save the values which don't match, and $vi)$ repeat steps $i - v$ for every rule. In the end, we know that whenever the state dataset contains values that violate rules, the new version of the SUT is not correct. Since only those rows in the state dataset corresponding to the modified SUT that are different from rows in the state dataset corresponding to the unmodified (correct) SUT have the potential to violate rules, it makes sense to only analyze the different rows.

4. Results

The full set of data generated during our experiments as well as all scripts can be found in our GitHub repo[2]. In our experiment, we use the Stack Class of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate. The Stack implementation contain seven public methods: *push()* which puts an item on the top of the stack, *pop()* removes and returns an item from the top, *clear()* clears the stack, *peek()* returns an item from the top of the stack without deleting it, *isEmpty()* tests if stack is empty, *size()* returns size of the stack and, finally, *toString()* returns a string representation of the object.

The methods *push(), pop()* and *clear()* modify instances of the Stack class when they are called. On the other hand, *peek(), isEmpty(), size()* and *toString()* provide information about the Stack object state when they are called and, thus, *peek(), isEmpty(), size()* and *toString()* are state methods. The test driver for Stack class creates an instance of the Stack class and contains public methods *push(), pop()* and *clear()* which call the original *push(), pop()* and *clear()* using the instance of the Stack class. Additionally, the driver implements *peek(), isEmpty(), size()* and *toString()* but these methods are private. Furthermore, the driver has a method for writing states to the CSV file. This method is used whenever *push(), pop()* or *clear()* methods are called during test execution. This set-up allows us to run Randoop test suite generation not on the original Stack class but on the driver class, where the public methods are the ones that modify the Stack objects. Thus, only *push(), pop(),* and *clear()* methods are called in the test sequences, and the state data captured by *peek(), isEmpty(), size()* and *toString()* is saved to a CSV file.

4.1. Phase I - Rule Set Generation

Step 1.1 - State data acquisition: Two different reports are the output of this step, *Test Report (pass / failed)* and *State Report*. The regression testing generates the *Test Report* . Test Driver generates the *State Report*. The *State Report* provides seven main features *testID*, *intanceID*, *size*, *isEmpty*, *peek_obj_type*, *pushInput*, and *calledMethod*. The features *testID* and *instanceID* provide an identification of the test generate by Randoop. One test can have multiples instances. Thus, *instanceID* is the identification of those instance belonging to the same test. The feature *size* tells the size of the Stack, and is a numerical feature. *isEmpty* feature contains the values "True" or "False". *isEmpty* is "True" when the Stack is empty, otherwise it will be "False". *peek_obj_type* tells us the element at the top of the Stack. *calledMethod* tells us which method was called, *i.e.,* push, pop, or clear.

Table 1
Summary of the support and lift metrics of the rule-set extracted from the Stack class data using ARM with different number of feature

DS†	NR†	Support			Lift		
		Max	Mean	Min	Max	Mean	Min
FS-3	14	0.403	0.381	0.283	2.539	2.391	1.946
FS-4	36	0.337	0.273	0.225	2.539	2.475	2.243
FS-8	439	0.269	0.227	0.205	3.808	3.404	2.545
FS-9	676	0.305	0.231	0.201	4.392	3.237	2.169
FS-10	1450	0.279	0.225	0.203	4.404	3.492	2.442

†Data Set, †Number of rules

Step 1.2 - State data pre-processing: In this step, we sorted the dataset based on the *testID* and *intanceID*, this is done to find the sequence of Stack size, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. For example, it is possible to add characteristics that indicate the previous state. To distinguish from the original features, we add a *_p* at the end of the feature name, this means "previous". Then, the previous states are named as: *size_p*, *isEmpty_p*, *peek_obj_type_p*, *calledMethod_p*. Then, we removed the unnecessary rows, this is, rows whit not state information. For instance, the *Test Driver* writes a rows with the name *"Constructor"* in the feature *calledMethod* which indicates that a new Stack was created. This information it is not related to the state, thus, should be dropped. Finally, we encoded the features should be encoded as categorical if there is a need. In the context of our data, we encoded the feature *size*, and *size_p* since they are not categorical features.

We create five different datasets which contain different numbers of features. The created dataset are named with the prefix FS, which stands for *Feature Selection*. To distinguish the different dataset, they have been named with the number of characteristics that were used, *i.e.,* FS-X where X is the number of features. The datasets created are the following: ***FS-3*** comprises the features *size*, *isEmpty*, and *peek_obj_type*. ***FS-4*** contains the features used in FS-3 plus *called_Method*. ***FS-8*** comprises the features used in *FS-4* and their previous values, which are *size_p*, *isEmpty_p*, *peek_obj_type_p*, and *calledMethod_p*. ***FS-9*** uses the FS-8 features and the feature *pushIputh*. Finally, ***FS-10*** uses all the features.

Step 1.3 - Rule mining: We apply the Apriori algorithm with minimal support and maximum confidence thresholds, *i.e.*, 0.2 and 1, to each dataset. Table 1 provides a comparison between the number of rules, support and lift values for each dataset. As per Table 1, we can observe that the average support ratio decrease when the number of features used is increased. The average is closer to the threshold value set up. Table 1 also shows

[2]https://github.com/aduquet/Using-Rule-Mining-for-Automatic-Test-Oracle-Generation

the number of rules that can be generated does not have linear behaviour since it depends on the number of items belonging to each feature. From Table 1 column lift, we can observe that the lift ratio is increasing when the number of features used is increased. This is the opposite of the support ratio. A lower lift ratio means that the probability of occurrence of a determinate rule is weak since the LHS and RHS are near to be independent between themselves.

4.2. Phase II

Step 2.1 - State data pre-processing: In this step, we created new versions of the Stack class. We introduce defects to the class under test. For example, in the Stack class we use three state methods: *peek()*, *size()* and *isEmpty()*. We modify each of them individually and also make all possible combinations of these modifications. Table 2 summarises the main modifications made to the SUT, and it provides the meaning of the terminology used to refers to the different version of the SUT. These modifications are done on purpose and manually, as we want to understand the potential of ARM to detect and locate faults using the state information of the SUT.

Table 2
Summary of the modifications injected to the Stack class

Modification	$peek()$	$isEmpty()$	$size()$
Mod_{1-p}	✓	–	–
Mod_{2-e}	–	✓	–
Mod_{3-s}	–	–	✓
Mod_{4-pe}	✓	✓	–
Mod_{5-ps}	✓	–	✓
Mod_{6-es}	–	✓	✓
Mod_{7-pes}	✓	✓	✓

Some of the modifications affect the state such that it will be quite easy to detect that something is wrong. For example, *isEmpty()* method is modified such that it returns *isEmpty=True* in the cases when size of the Stack class instance is 2 or 0. Thus, we would get an obviously faulty state *size="2.0"*, *is_empty="true"*. The modification of *peek()* will not return the object on the top of a Stack class instance but the previous one in the cases when stack size is greater or equal than 2. Modification of *size()* would return incorrect size for the Stack class instances that contain one or more objects. Thus, the states would look like the correct ones, and the dataset would not contain faulty-looking rows.

Step 2.2 - State data pre-processing: In this step, the same process of Step 1.2 was performed on the new data.

Step 2.3 - Apply the rule-set to the new SUT versions: We applied the ruleset against the state data of new versions by following the activities described in Section 3.2.

4.3. RQ1: How effective is the rule mining approach?

Table 3 summarises the not modified version and the seven modifications of the Stack class regarding the results obtained by the regression test, columns Regression test (pass / failed), and the information obtained during the test execution generated by the Test Driver, *i.e.,* the State Data. We notice from Table 3 that some datasets that correspond to modifications, *e.g.,* Mod_{2-e}, Mod_{4-pe}, Mod_{6-es}, and Mod_{7-pes}, do not have the same number of rows as in the No-Mod dataset. This is because some tests from the test suite are failed during the test execution the state in these cases will not be written to the CSV file. When no tests from the suite are failed, all the states will be written to the file. Therefore, the number of rows will be equal to the Not-Mod data since we execute the same test suite both for the Not-Mod and for the modified data extraction, *e.g.,* Mod_{1-p}, Mod_{3-s}, and Mod_{5-ps}.

As Table 3 column "Regression test" shows, only four of seven modifications have failed tests (Mod_{2-e}, Mod_{4-pe}, Mod_{6-es}, and Mod_{7-pes}). We see that the number of failed tests is the same for Mod_{2-e} and Mod_{4-pe} datasets. Also, the number of failed tests are the same for Mod_{6-es} and Mod_{7-pes}. The state data columns also shows the same behaviour, but regarding the number of rows generated. The common aspect between all these datasets is *isEmpty* method modification. In particular, Mod_{2-e} and Mod_{4-pe} have 1452 failed test. The Mod_{4-pe} modification is the combination of the modified methods *peek* and *isEmpty*. However, it seems that the regression test spots the fault related to *"isEmpty"* only. Furthermore, the Mod_{1-p}, which is the modification of Peek, none regression tests failed. This fact confirms the regression tests failed of Mod_{4-pe} are belonging to *isEmpty* modification only.

Same as Mod_{1-p}, none regression test failed in Mod_{3-s}. In this modification *"size"* method is modified. The number of tests failed in Mod_{7-pes} and Mod_{6-es} are different from Mod_{2-e} and Mod_{4-pe}. *Are the regression tests spotting faults in the other modified methods, i.e., "peek" and "size" when combined in this way?* The modification of *size()* returns the incorrect size for the Stack class instances that contain one or more objects, *i.e.,* when the size of the Stack is greater than 0, the modification would return the correct size plus one. For instance, if the $size = 1$, the modified version will return $size = 2$. That is why Mod_{6-es} increase the number of failed tests because the size modified method increases the number of $size = 2$, then triggers the modified *isEmpty()* when the size of the Stack class is 2 or 0. From the Table 3 we can ask ourself, *why the regression tests are failing only in the isEmpty() method?* When analysing in detail the report provided by regression test, we can find that during the execution of the test, there is an exception

Table 3
Summary of the no modified version and the seven modifications of the Stack class regarding the results obtained by the regression test, and the information obtained during the test execution generated by the test driver

Dataset	Regression test		State data																	
	Pass	Failed	Total # of rows	Total # of new rows	Total # of UR†					Total # of new UR†					Total # of new UR† that violate rules					
					FS-3	FS-4	FS-8	FS-9	FS-10	FS-3	FS-4	FS-8	FS-9	FS-10	FS-3	FS-4	FS-8	FS-9	FS-10
No-Mod	2037	0	71604	0	47	76	320	320	320	0	0	0	0	0	0	0	0	0	0
Mod_{1-p}	2037	0	71604	11882	45	73	232	308	398	28	43	97	166	189	0	0	0	38	38
Mod_{2-e}	585	1452	37786	6390	25	33	108	108	108	1	2	20	20	20	1	2	20	20	20
Mod_{3-s}	2037	0	71604	43405	47	76	320	320	320	46	74	237	237	237	0	0	19	19	19
Mod_{4-pe}	585	1452	37786	7792	26	33	106	119	133	14	20	46	59	66	1	2	31	42	49
Mod_{5-ps}	2037	0	71604	43405	44	73	232	308	398	44	71	156	232	279	0	0	19	50	57
Mod_{6-es}	25	2012	7160	5497	5	9	17	17	17	5	7	8	8	8	1	2	6	6	6
Mod_{7-pes}	25	2012	7160	5497	3	7	14	16	17	3	5	6	8	8	1	2	4	7	7

†Unique rows

that checks if the Stack is empty or not. By making the modification in the *IsEmpty()* method, we generate the situation where the exception is generated, thus allowing the test to not finish its execution and report it as a failure.

Table 4
Percentage of new unique rows (among all-new unique rows) that violate at least one rule

Dataset	% of new unique rows that violate rules				
	FS-3	FS-4	FS-8	FS-9	FS-10
No-Mod	-	-	-	-	-
Mod_{1-p}	0	0	0	22.89	20.11
Mod_{2-e}	100	100	100	100	100
Mod_{3-s}	0	0	8.02	8.02	8.02
Mod_{4-pe}	7.14	10	67.39	71.19	74.24
Mod_{5-ps}	0	0	12.18	21.55	20.43
Mod_{6-es}	20	28.57	75	75	75
Mod_{7-pes}	33.33	40	66.67	87.50	87.5

In Table 3, the column "Total # of unique rows" indicates that the number of unique rows increases when the number of features increases. This is because by adding more features, we increase the heterogeneity in the data. The number of new unique rows refers to those rows that are different from the unmodified SUT version. The idea of using state information is based on the assumption that the relationship in state when testing a new version of the SUT must remain unchanged or must not change significantly. Therefore, we can conclude that the rows of the modified versions that are different from the unmodified version are failures. Up to this point our method is capable of detecting failures without the need to use ARM.

We are interested in understanding whether the information provided by the resulting ARM model would have confirmed the failure detection that is already given when identifying new rows in the state dataset of a modified SUT. Therefore, we measured the proportion of rules that are violated by new rows. In Table 3, the column "Total # of new unique rows that violate rules" shows the total number of new unique rows that violate at least one rule. It turns out that only for the *isEmpty()* modification always all rows also trigger a rule violation. It is less often the case when other methods are modified (either individually or in combination). Only when the largest feature sets (FS-9 and FS-10) are used, there is always at least one new row in the dataset that also violates at least one rule. This result is weaker than what we can already see by just looking at new rows but, recalling that none of the regression tests failed when only methods *size()* and *peek()* were modified, rule violations seem to occur in a more balanced way. This let us hope that we might be able to exploit this when answering research question RQ2.

4.4. RQ2: What information regarding fault localisation can the method offer?

Regarding the first research question, we concluded that failure detection effectiveness improves by comparing state data even without using ARM. However, neither analyzing the traces of failing tests (all of them failed when executing the *pop()* nor inspecting the information provided in the new unique rows provided any helpful information that would guide fault localization. Therefore, we set our hopes in a systematic analysis of the rules that are violated by new unique rows.

As a starting point for fault localization, it is necessary that at least one rule be violated by at least one new single row. Table 5 summarises the total number of rules generated per dataset, and the number of rules violated among all the rules generated. As Table 5 shows, from FS-8 to FS-10, more than a hundred rules need to be analysed to be able to localise the fault. To reduce and optimize the number of rules, we construct a rule hierarchy. Let us consider the following set of rules: *(i)* $A, B, C \rightarrow D$ *(ii)* $A, B \rightarrow D$ *(iii)* $C, B \rightarrow D$, and *(iv)* $C \rightarrow D$. The rule *(i)* contains the same items of rules *(ii)*, *(iii)*, *(iv)* in implicitly. Therefore, having only rule *(i)* is sufficient for

Table 5
Total number of rules generated per-each dataset, and the number of rules violated among all the rules generated

DS	RG†	Total # of rules Violated (the row match with LHS but NOT with RHS)													
		Mod_{1-P}		Mod_{2-e}		Mod_{3-s}		Mod_{4-pe}		Mod_{5-ps}		Mod_{6-es}		Mod_{7-pes}	
		RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*	RV$^{\perp}$	OSR*
FS-3	14	0	0	3	-	0	0	3	-	0	0	3	-	3	-
FS-4	36	0	0	4	-	0	0	4	-	0	0	4	-	4	-
FS-8	439	0	0	73	16	95	95	73	56	95	95	113	98	113	98
FS-9	676	12	10	142	61	95	95	224	191	307	297	297	269	297	272
FS-10	1450	12	10	224	65	285	285	236	191	297	397	314	269	325	272

†Rules generated, $^{\perp}$ Number of rules violated * Optimal set of rules

interpretation purposes.

Once the number of rules is reduced, we can analyse them more efficiently. The key to being able to analyse the set of rules violated is to answer the following question: *Which item is violating the rule?;* A rule violation occurs when some items in a row match the LHS of the rule, but at least one items in the same row do not match the RHS of the same rule. Based on the above, the item that generate the violation of the rule must be present more frequently in the RHS than in the LHS of the set of violated rules. Then, to locate the fault, we quantify the occurrence of the main methods in the LHS with respect to their occurrence in RHS. If the LHS/RHS ratio approaches zero, the fault has been located. Let's consider Table 6 the set of rules violated by Mod_{1-p} of **FS-9**, the total number of rules violated are 10. Please note the following nomenclature: *A: peek, B: isEmpty, C: size. D: calledMethod,* and *E: inputPush.* Let's compute the relation LHS/RHS per each item, *i.e., A, B, D* and *E.* $A = 3/7 = 0.428$, $B = 2/2 = 1$, $D = 6/2 = 3$, and $E = 6/3 = 3$. The value closest to zero is $A = 3/7 = 0.428$, which corresponds to the peek method. We can conclude that the fault has been located, and that it corresponds to the peek method.

Table 7 reports modifications located using our approach. According to the results shown in the table, six out of seven modifications could be located when using nine or ten features. An interesting aspect that stands out from the results is the localization of the modifications where the *isEmpty* method is involved. When using fewer features, *e.g.,* FS-4, the *isEmpty* modification is always located, which is not the case when using more features.

Note that our method can only point out one fault per analysis. Therefore, one must apply the analysis repeatedly. Once a fault has been located, it must be removed, the tests must be run again, and if there are still new unique rows in the state dataset, we must try to spot the next fault. Thus, if we had started out with the case where all three methods were modified, and we had used models with 8, 9, and 10 features, we would have first located the fault in *(*with all feature sets), then in *peek()* (with FS-9 and FS-10), and then in *isEmpty()* (with FS-8).

Table 6
Set of violated rules of Mod_{1-p} with FS-9

LHS	RHS
D = push, E = objType	***A = objType***
E = objType	*D = push,* ***A = objType***
B = True, E = objType	***A = objType***
D = push	*B = True,* ***A = objType***
D = push, B = False, E = objType	***A = objType***
E = objType	*D = push, B = False,* ***A = objType***
D = push, ***A = objType***	*E = objType*
D = push, B = False, ***A = objType***	*E = objType*
D = push, ***A = objType***	*E = objType, B = False*
E = objType	***A = objType***

Table 7
Fault localised using ARM approach

Modification "Bug"	Modification localised by ARM				
	FS-3	FS-4	FS-8	FS-9	FS-10
Mod_{1-p}	✗	✗	✗	p	p
Mod_{2-e}	-	e	e	pushInput	pushInput
Mod_{3-s}	✗	✗	s	s	s
Mod_{4-pe}	✗	e	method	p	p
Mod_{5-ps}	✗	✗	s	s	s
Mod_{6-es}	-	e	s	s	s
Mod_{7-pes}	-	e	s	s	s

5. Threats to Validity

In the context of our proof-of-concept validation, two types of threats to validity are most relevant: threats to internal and external validity.

With regards to internal validity, it is not fully clear in which situations the iteratively applied heuristic that we propose to localize faults is successful, i.e., points to a rule element that carries useful information. It is clear that our method only can capture behaviour of the program that has not been changed and, thus, behaves equal at the level of granularity at which we capture object states. In addition, our positive results might depend on the number and type of faults injected. A more systematic and more comprehensive analysis is required to explore the limitations of our method with regards to both failure exposure and fault localization.

With regards to external validity our study is rather limited since we only use one well-understood class in our experiments. Thus, the actual scope of effectiveness of our proposed method for object-oriented programs is yet to be determined.

6. Related Work

A good overview of methods proposed to automatically generate test oracles can be found in [2] and [4]. As far as we know, ARM has not yet been used for this purpose by other researchers. A field that has recently received attention in the context text oracle generation is metamorphic testing as metamorphic relations can be used as test oracles[10].

7. Conclusion

We presented a new ARM-based method for the detection and localization of faults using the state data from SUT executions. In our proof-of-concept, we used the Stack class from the Java collection framework as the SUT. To test our method, we generated seven faulty versions of the SUT. The results obtained have shown that our approach is capable of detecting failures and locating faults. The ARM-approach mainly benefits fault localization.

An advantage of our method is that our method tested not on the SUT with only integer inputs and outputs but on the class under test where we can have any type of inputs, and the state data consequently is also of mixed types of data. Thus, our method can be generalized for inputs of any type, not only for integers. It removes some limitations on the type of SUT that can be analyzed.

One of the weaknesses of our method is the need of a test driver that we use to extract state data during the test suite execution. To generate the test driver for the SUT, we have to identify the state extracting methods manually. For efficiency reasons, it would be better to have an automatic identification of state extracting methods. Unfortunately, there is no simple way to do this. Also, in the case of the manual identification, for some classes, it may not be so clear what methods should be marked as state extracting methods.

Given the limitations of our study, more experiments have to be conducted to empirically test our proposed method for fault detection and localization. We are currently focusing on extending our experiments in two directions. First, we will add more kinds of fault injections to test the sensitivity of our method with regards to the type of faults in a program. We will systematize this by using mutation. Second, we will apply our proposed method to more classes in the Java collections framework and beyond.

Acknowledgments

This research was partly funded by the Estonian Center of Excellence in ICT research (EXCITE), the IT Academy Programme for ICT Research Development, the Austrian ministries BMVIT and BMDW, and the Province of Upper Austria under the COMET (Competence Centers for Excellent Technologies) Programme managed by FFG, and by the group grant PRG887 of the Estonian Research Council.

References

[1] T. M. Abdellatif, L. F. Capretz, D. Ho, Software analytics to software practice: A systematic literature review, in: 2015 IEEE/ACM 1st Int'l Workshop on Big Data Software Engineering, 2015, pp. 30–36.

[2] R. Braga, P. S. Neto, R. Rabêlo, J. Santiago, M. Souza, A machine learning approach to generate test oracles, in: Proc. of the XXXII Brazilian Symp. on Softw. Eng., SBES '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 142–151.

[3] K. Patel, R. M. Hierons, A partial oracle for uniformity statistics, Softw. Quality Journal 27 (2019) 1419–1447.

[4] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, S. Yoo, The oracle problem in software testing: A survey, IEEE Trans. on Softw. Eng. 41 (2015) 507–525.

[5] L. Zhou, S. Yau, Efficient association rule mining among both frequent and infrequent items, Computers and Mathematics with Applications 54 (2007) 737 – 749.

[6] S. K. Solanki, J. T. Patel, A survey on association rule mining, in: 2015 Fifth Int'l Conf. on Advanced Computing Communication Technologies, 2015, pp. 212–216.

[7] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: Proc. of the 20th Int'l Conf. on Very Large Data Bases, VLDB '94, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994, p. 487–499.

[8] G. McCluskey, Efficient-apriori documentation, 2018.

[9] A. Bhandari, A. Gupta, D. Das, Improvised apriori algorithm using frequent pattern tree for real time applications in data mining, Procedia Computer Science 46 (2015) 644 – 651.

[10] B. Zhang, et al., Automatic discovery and cleansing of numerical metamorphic relations, in: Proc. 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019), 2019, pp. 235–245.