UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Anastasiia Shalygina

Using Rule Mining for Automatic Test Oracle Generation

Master's Thesis (30 ECTS)

Supervisor: Dietmar Pfahl
Co-Supervisor: Rudolf Ramler

Tartu 2020

Using Rule Mining for Automatic Test Oracle Generation

Abstract:

Software testing is essential, but also one of the most costly and time-consuming activities in the software development process. The mechanism to determine the correct output of the system under test for a given input space is called test oracle. The test oracle problem is a known bottleneck in the software testing process. It describes the sophisticated task of distinguishing between correct output and faulty output of a system under test, for which no universal solution exists that works for any system under test. One of the artifacts that can be used to derive test oracles automatically is a previous version of the system under test. Such oracles can be used to validate new versions, for example, the versions where some corrective modifications were made.

In this thesis, we develop a methodology for test oracles generation using internal state data extracted from the system under test during the test suite execution. We derive a test oracle in the form of rules using the association rule mining technique. The oracle can be used to validate new or modified versions of the system under test. We test our method on a Stack class from a custom version of the Java collection classes and discuss the lessons learned during our experiments with the class under test. We were able to show the feasibility of our method, but it has several disadvantages that restrict its capability to generate oracles. We also discuss the limitations of our study. As a main result, we were able to develop a new method and an experimental setup to generate test oracles using internal states of the system under test. We were able to derive a partial test oracle for the Stack class using our method.

Keywords:
test oracle, test oracle problem, test oracle automation, machine learning methods in software testing, association rule mining, internal state

CERCS: P170 Computer science, numerical analysis, systems, control

Reeglikaeve kasutamine automaatseks testi oraakli genereerimiseks

Lühikokkuvõte:

Tarkvara testimine on elementaarne osa, kuid samas ka üks kulukamaid ja aeganõudvamaid osasid tarkvaraarenduse protsessist. Mehhanismi testitava süsteemi korrektse väljundi leidmiseks ette antud testjuhu jaoks nimetatkse testi oraakliks. Testi oraakli automatiseerimise probleem on teadaolev kitsaskoht tarkvara testimise protsessis. See kirjeldab keerukat ülesannet eristamaks testitava süsteemi korrektset väljundit ebakorrektsest väljundist, millel ei ole universaalset lahendust mis toimiks kõikide süsteemide korral. Üks artefakt, mida saab kasutada testi oraakli tuletamiseks on testitava süsteemi eelnev versioon. Selliseid oraakleid saab kasutada uute versioonide valideerimiseks, näiteks kui uues versioonis on tehtud parandavaid korrektuure.

Antud töös arendame me välja metoodika testi oraaklite genereerimiseks kasutades testitava süsteemi sisemise oleku andmeid, mis on saadud teste jooksutades. Me tuletame testi oraakleid, mis on esitatud reeglite kujul, kasutades assotsiatsioonireeglite leidmise meetodit. Antud oraakli eesmärk on valideerida uusi või muudetud testitava süsteemi versioone. Me testime oma meetodit rakendades seda Stack klassil, mis kuulub kohandatud Java kollektsiooni klasside hulka ja arutame mida õppisime eksperimendist antud testitava klassiga. Meil õnnestus demonstreerida oma meetodi teostatavust. Siiski on antud meetodil mitmed nõrgad küljed, mis piiravad selle võimekust oraakleid genereerida. Samuti toome välja antud uurimustöö piirangud. Põhitulemusena arendasime uue meetodi ja eksperimentaalse ülesehituse oraaklite genereerimiseks kasutades testitava süsteemi sisemisi olekuid. Kasutades antud meetodit õnnestus meil tuletada osaline testi oraakel Stack klassi jaoks.

Võtmesõnad:

testi oraakli probleem, testi oraakli automatiseerimine, masinõppe meetodid tarkvara testimises, assotsiatsioonireeglite leidmine, sisemine olek

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Software testing is an essential activity for quality assurance in software development. It involves examining the system behavior in order to detect hidden defects. However, it is known as a very costly and time-consuming activity. According to [KM16], software testing utilizes approximately 40%-50% of total resources, 30% of total effort, and 50%-60% of the total cost of software development. Although the testing phase is a major challenge in software development, it should be considered as a possibility to optimize the quality of software, reduce cost, make the product more profitable and improve the user experience. Therefore, automating the testing process may significantly reduce the total cost of software development.

Automated testing 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. Automated testing has numerous advantages such as avoiding human errors, speeding up the testing process, better testing time estimation.

Automation of the testing process has several aspects. One of the challenging aspects is the automation of the test input generation; the significant part of a testing process is finding successful test inputs, i.e., such inputs that can reveal faults in SUT. Another challenge is to determine what should be the output of a system after the execution of the test cases. This challenge refers to one of the most significant problems in software testing automation, i.e., the test oracle problem.

A test oracle is a mechanism that determines the correct output of SUT for a given test case input. Automation of the test oracle is quite a sophisticated task. None of the existing solutions is proven to be fully adequate and universal. Thus, it creates a bottleneck that prevents a general software testing process automation.

In this research, we focus on a level of software testing called unit testing that tests every program unit (or component) separately. We study the possibility to generate test oracles for Java units using internal states of the class (unit) under test. In particular, we explore the internal state data and apply machine learning methods to construct a model that can identify interesting relations in the data. Knowing these relations may help to verify new or modified versions of the software. We develop a methodology based on association rule mining to mine interesting relations in the internal state of SUT. We discuss our observations received during the experiments with this approach. We conclude whether it is useful and can be applied to the real systems and also discuss limitations.

1.1 Motivation

Although numerous techniques for test oracle generation that have been developed in recent years, no one of the techniques is completely universal and usually requires some

additional data. It can be system specifications, existing successful test cases, etc. Quite often, no formal system specification exists, or existing specifications are not complete and do not describe all the system's features. In many cases, it is also not possible to derive test oracles based on other artifacts required for existing techniques. Thus, testers still often have to write the test oracles manually. That leads to human errors, increased time of the testing stage, and growing software costs. Therefore, the problem of the test oracle automation remains relevant, and there is a need for new methods or artifacts which can help the testers to fully or partially solve this problem.

1.1.1 Oracle Problem

The term "test oracle" was introduced in the paper by William E. Howden in 1978 [How78]. Howden defines oracle as an instrument that can be used to check the correctness of the test output. He distinguishes different kinds of test oracles. The most common kind is the one that can be used to check the correctness of output for the given test case input, and the other kinds can be used to check the correctness of value traces of selected program variables [How78]. According to the paper [How78], formally defined test oracles may consist of tables of values, algorithms for hand computation, or formulas in the predicate calculus while informally defined oracles are often simply the assumed ability of the programmer to recognize correct output.

The informally defined oracles later obtain the name of human oracles. The cost of a human tester effort to define the test oracle refers to the oracle cost problem [PMH10]. Since human oracles are very ineffective in terms of time, software costs, and high probability of the human mistake, one should concentrate on the formally defined test oracles and its' automation. The existing methods of test oracle automation described in chapter 2. The literature survey on the topic of oracle automation presented in the same chapter, and also it is shown that existing methods are very limited, and this topic should obtain more attention from the researchers.

1.1.2 General Ideas

The internal state of an object in object-oriented design is a set of all the object's attribute values. The majority of object-oriented programs are built around the objects with a mutable internal state, and the execution of methods can modify the state of the program. One of the fields for studying is the use of observable internal states of SUT to derive test oracles. N. Li and J. Offutt in [LO17] claim that the program state observed during tests execution, can be used to reveal faults. According to [LO17], execution of the tests in the location of the fault made in the program, must cause an incorrect internal program state.

The idea of using internal state data lies back on the assumption that relations in the state should remain unchanged or should not change significantly if a new version of the program being tested. Sometimes the new version of the program means that a

similar program rewritten in another programming language or in a different version of the language for some purpose. In this case, we definitely can confirm that the relations in states should not be changed if the new version is correct. Another case is when some components of the program modified, or some new functionality added. Then, the behavior of already existing not modified classes should remain unaltered. However, the behavior of such classes could be violated by interaction with the modified or new components if its' implementation is not entirely correct. Thus after each modification of the software tester should check all the components to make sure that the program works correctly.

We can illustrate the idea of using internal state data for test oracles generation with the following example.

Consider class Alpha presented on the Figure 1. Class Alpha has 5 public methods – setX, setY, getX, getY, getSum and getProduct. We can logically separate these methods into two groups – "set" group and "get" group. Calling methods from the first group can change the state of the object instance, whereas methods from the second group can give us some information about the instance of Alpha. In this thesis, we call the methods of the second type "internal state methods". In other words, we call "internal state methods" such methods that reflect the behavior of the class and can be accessed externally by calling it with the class instance.

The proposed approach to observe the internal state of the object is a black-box approach. We do not have direct access to the internal state of the tested object because we do not access private member variables or call private methods. We only consider the externally visible state of the object exposed by public methods that are also available to the user. Such an approach may be regarded as a limitation, but reflects the usual practice in unit testing.

```
public class Alpha{
    private int x = 0;
    private int y = 0;

    public void setX(int x) {this.x = x;}
    public void setY(int y) {this.y = y;}

    public int getX() {return x;}
    public int getY() {return y;}
    public int getSum() {return x+y;}
    public int getProduct() {return x*y;}
}
```

Figure 1. Example of class under test

Table 1 demonstrates example of the values that would be returned by "get" methods (internal state methods) after calling "set" methods with different parameters. One can notice that content of some columns depends on the values in the other columns. For instance, the values in "getSum()" and "getProduct()" columns are the sum and the product of values in columns "getX()" and "getY()" respectively. Thus, one can say that there exists some relation between columns "getX()", "getY()" and "getSum()" and "getProduct()". If the content of "getX()" and "getY()" changes, then content of "getSum()" and "getProduct()" also changes.

Table 1. Internal state of class Alpha

	getX()	getY()	getSum()	getProduct()
setX(1)	1	0	1	0
setX(2)	2	0	2	0
setY(1)	2	1	3	2
setY(2)	2	2	4	4
setY(1)	2	1	3	2
setY(2)	2	2	4	4

Consider class Alpha to be a class under test and let T_{Alpha} be a test suite for the class Alpha. Executing T_{Alpha} on the class Alpha and using some helper program, we can extract internal state data similar to the data in the Table 1. The idea is to study these data and to build a machine learning model able to learn relations in the data and later use it for validation of new versions of the class under test.

1.2 Research Objectives

In our research, the focus not on getting the fully-working method of deriving test oracles using internal states of SUT. The main objective of this research is to explore the possibility of test oracles generation using SUT states and association rule mining techniques. The steps to conduct this research can be listed as follows:

1. To study existing test oracle automation methods and provide a comprehensive review of literature on these methods.

2. To develop a methodology for the generation of test oracles on the base of internal state data extracted during the test suite execution on the system or class under test.

3. To produce an experimental setup and provide the full description of all the components such as helper programs, datasets, used tools and algorithms.

4. To test the method on a simple Java class with a well-known and easily observable behavior in order to evaluate the method's performance.

5. To discuss the advantages, disadvantages, and limitations of this method as an implication of conducted experiments.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides a survey of literature on test oracle automation techniques and, in particular, on derived test oracles and on the previous work on the application of machine learning to test oracle problem. It also provides literature on association rule mining algorithms.

Chapter 3 introduces rule-based method for mining test oracles. It describes the process of data collection and preparation, rule mining model training, method application, and analysis of the method.

Chapter 4 provides experiments described in Chapter 3 on the Stack class.

Chapter 5 describes the main lessons learned during experiments with the rule mining for test oracles and discuss the limitations of our study.

Chapter 6 draws our conclusions.

Finally, in the Appendix, we provide some additional materials to Chapter 4 and a link to the GitHub repository that contains all the project code.

2 Background

In the following chapter, we present a survey of literature that covers the topic of test oracle automation. We describe the main issues in this field as a motivation for the need for new methods of test oracle automation. Additionally, we cover the topic of association rule mining, which is a machine learning technique we use in our research.

In section 2.1, we consider several papers on test oracle automation problem and the existing methods to approach it, we give an overview of different types of test oracles and define the type of oracle to which the method presented in this thesis can be assigned. We consider several works that use machine learning to generate test oracles.

In section 2.2, we provide background on association rule mining and motivate the choice of the algorithm that we use for our experiments.

2.1 Overview of Test Oracle Automation Methods

In the research on software testing, much work focuses on automation of the general process. Nevertheless, the challenge of test oracle automation gets significantly less attention than other testing process aspects, although the oracle problem can be considered as a bottleneck that prevents simpler creation of automated tools and methods.

According to Barr et al. [ETBY15], the problem of automatically generating test inputs, that involves finding such inputs that cause execution to reveal faults, has been the subject of research interest for nearly four decades. Numerous successful findings have been presented in this field. However, none of these methods include a comparison of the generated test inputs with respect to the expected behavior, i.e., none of these methods address the oracle automation problem. Thus, even if for some software, the test inputs creation process can be automated, testers still would struggle to determine the correct output for the test inputs in order to distinguish between error-revealing and passed test cases.

When a SUT has a detailed and full-specified description of the intended behavior, or the code itself is designed such that it implements well-understood contract-driven development approaches [ETBY15], the test oracle problem is ameliorated by the presence of an automatable test oracle. Such oracles are free from costly human intervention and fully cover all the test inputs. However, quite often, no full specification of the system exists. In this case, it might be possible to construct partial test oracles. Such oracles can determine whether the system's output is correct only for some inputs. For instance, metamorphic testing [TYCY98] allows deriving such partial test oracles on the basis of known relationships between outputs of the multiple SUT inputs. However, for many systems, testers do not have formal specifications, or it is not possible to construct automated partial test oracles like it is for distributed systems described in [Hie11]. Therefore, testers still have to check the SUT behavior manually, which leads to the human oracle cost problem [SAS13], [PMH10].

In the survey [ETBY15], four cases of test oracles have been considered: implicit test oracles, the case when no test oracle presented, specified, and derived oracles. Implicit test oracles refer to the detection of "obvious" faults of the system, such as program crash [ETBY15]. The case when no oracle presented refers to human oracles. Specified test oracles use the system's formal specifications to define the correct behavior. Derived test oracles use various artifacts or properties from which the test oracle can be constructed. For example, it can be a previous version of the system, documentation, or system executions. In this research, we focus on the derived test oracles and, in particular, on the use of previous versions of the system as an artifact for derived oracles.

2.1.1 Derived Test Oracles

When specified test oracle is not available, testers fall back on the derived test oracles. This situation appears very often since even if the specification exists, it may not be updated for the new versions of the system. Therefore, it not fully cover all the system features.

Derived test oracles can be divided into several subgroups. In [ETBY15] authors describe cases of pseudo-oracles [DW81], metamorphic relations [TYCY98], regression test suites [YH10], system executions [Dil00] and textual documentation [Sch]. Further we concentrate our attention on the literature about regression test suites since it is the most related type of derived test oracles to the approach which we introduce in this thesis.

Regression testing relies on the assumption that the SUT previous version can play a role of the test oracle for the existing functionality of the system. If the modifications made to the SUT were just the corrective ones and the functionality remains the same, then the test oracle for the previous version S_i can serve as an oracle for the corrected version S_{i+1}. We denote these test oracles as O_i and O_{i+1}.

For example, in [Xie06] T. Xie presents an automatic approach to generate a unit-test suite with regression oracle checking. The supporting tool for this approach, called Orstra, first executes the test suite and collects the class under test's object states exercised by the test suite [Xie06]. Then, Orstra creates assertions for asserting behavior of the object states [Xie06]. Later, when the class under test is changed, the augmented test suite is executed to check whether assertion violations are reported [Xie06].

In this thesis, we derive and explore the method to provide test oracles for corrective modifications of the system. Our approach is also based on the use of states of the previous version of SUT to generate a test oracle for the new (modified) version.

Another type of modification that can be made to SUT is perfective modifications. These are such modifications that add new features to the system. In this case, the test oracle for the new version should be augmented with oracles for the new features. Denote this augmentation as ΔO. Then, $O_{i+1} = O_i + \Delta O$.

There are several augmenting techniques for the new version test suite exists [XKK$^+$10]. However, the more complex task is to not only find an augmentation for test inputs but

also to augment the test oracle in accordance with these inputs. In this way, test suite augmentation could be extended to augment the existing oracles as well as the test data [ETBY15].

2.1.2 Machine Learning for Test Oracle Automation

Machine learning techniques have been successfully used to alleviate many software engineering activities [ZT02]. Since the oracle problem is a software engineering aspect that gets less attention then others in the scientific literature [DDB+19], very few works have been presented in the area of machine learning application to test oracle problem.

We have found several works that approach the test oracle generation problem with machine learning. Here we briefly describe five such papers. All the approaches proposed in the mentioned papers, generally fit the type of machine learning called supervised learning. Supervised learning refers to learning such a function that maps input data to an output based on input-output pairs given as an example. Examples of supervised learning are classification and regression problems.

One of the works where machine learning used for test oracle generation is a paper by R. Braga et al. [BNR+18]. In this paper, R. Braga et al. describe the process of test oracle generation for the web application. As training data, they use historical usage data from an application. The data collected by inserting a capture component into the application under test. They use AdaBoostM1 and Incremental Reduced Error Pruning (IREP) algorithms as classifiers to distinguish between the correct and incorrect behavior of the software.

In [WYW11] F. Wang at al. examine how to construct test oracles for reactive programs without explicit specifications automatically. Their approach turns test traces into feature vectors, and the feature vectors are used to train the machine learning algorithm. In particular, they use SVM classified for this purpose.

Other three articles [VSB14], [JWC+08], [VLK02] investigate usage of artificial neural networks (ANNs) to generate test oracles.

Authors of [VSB14] train a neural network and a decision tree algorithm on test case inputs and test case outputs to later predict outputs for other test case inputs. They also compare these two machine learning models. They test the models on the triangle problem where all the inputs and all the outputs of SUT are integers.

Paper [JWC+08] also presents the case of ANN being applied to the triangle problem. The authors discuss whether it is possible to use ANN for oracle generation. They conclude that for some problems, this method would not work directly because it should be a need for more preprocessing and analysis. One such example is when SUT's inputs are not integers.

In [JWC+08] multi-layer neural network is trained to generate oracles for a small credit card approval application by using randomly generated test data that conform to the specifications. The neural network also was trained on the integer data. The resulting

model classifies whether the program output is correct or not. Authors of the paper applied mutation testing [JH11] to generate faulty versions of the original program.

Unlike the described above methods, our method lies in unsupervised learning. It gives us an advantage that we do not have to create the datasets with annotated correct and faulty examples, and we do not have to define a classification or regression problem. Also, we try to make our rule-based method work for any input data – integers, strings, booleans, etc.

2.2 Association Rule Mining

Our main interest in this research is to automatically learn interesting relations in the internal state of the system under test. One of the approaches to learn such relations is the association rule mining technique.

Association rule mining is a rule-based unsupervised machine learning method that allows discovering relations between variables in large databases. Rule mining algorithms aimed to identify strong rules in transactional databases using some measures of interestingness. The two most popular measures are support and confidence.

These two measures described in detail in [TR14]. For association rule $X \rightarrow Y$, support is a proportion of records that contain $X \cup Y$ to the overall records in the database. Confidence is defined as the proportion of the number of records that contain $X \cup Y$ to the overall records that contain X, where, if the ratio outperforms the threshold of confidence, an association rule $X \rightarrow Y$ can be generated [TR14].

There are several algorithms for association rule mining exists. In the paper by S. Solanki and J. Patel [SP15] authors describe eight different algorithms. The most basic algorithm for mining association rules is the Apriori algorithm. Apriori was presented in 1994 by R. Agrawal and R. Srikant in [AS94]. According to [AS94], problem of discovering all association rules can be decomposed into two subproblems:

1. Find all sets of items (itemsets) that have support above the predefined threshold for support (minimum support).

2. Use the itemsets to generate the association rules.

Thus, the Apriori algorithm works in two phases. First, it looks for frequent itemsets with the desired support threshold. The itemsets then combined to obtain candidate sets. Second, candidate sets are pruned according to the property of support called downward-closure property. This property guarantees for a frequent itemset that all of its subsets are also frequent. Thus, no infrequent itemset can be a subset of some frequent itemset. By the use of downward-closure property, the Apriori algorithm can find all frequent itemsets and thus all the desired rules.

Another algorithm described in [SP15] is an extension of the Apriori, which is aimed to reduce space and time complexity. This algorithm called CT-Apriori. In this approach, a compact transaction tree (CT tree) constructed by compressing the original transaction

tree. It helps to reduce the running time and the storage space but still requires candidate sets generation, which may take some storage space.

One more basic algorithm which has an advantage over Apriori by space-time complexity is FP-Tree. The FP-tree structure stores compressed and vital information about frequent patterns and develops an efficient FP-growth mining algorithm for mining complete frequent patterns [SP15]. According to [SP15], FP-Tree does not generate candidate sets, only two database scans are required during the rule mining procedure, and thus, it is more efficient and scalable then Apriori.

FP-tree algorithm has an improvement called Dynamic FP-tree. It provides a dynamic addition of itemsets without any candidate generation. It gives better performance in terms of space and time in comparison to all the algorithms mentioned above.

The last type of association rule mining algorithms is fuzzy algorithms. Such algorithms imply fuzzy logic in association rule mining. The rule mining is done after dividing the database into fuzzy regions.

For our experiments, we decided to use the Apriori algorithm. It is the most popular algorithm of all the described above. We use Python3 for data analysis, for rule mining, and for rules application. Apriori has several implementations in different Python3 libraries and good documentation provided for some of these libraries. Furthermore, we assumed that we would not have too big datasets, and the time and space savings are not so significant on the stage when we start to develop the new method to generate test oracles. The most important part is to explore the possibility of using association rules for this task. Thus, we decided to choose the most popular and easily available algorithm for rule mining.

3 Methodology

In the following chapter, we present a rule mining method for test oracles generation based on the SUT internal state. The proposed method diagram has shown in Figure 2.

The whole process starts with the extraction of the internal 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 training dataset received, and the rule mining process starts. After that, one gets a set of rules using that a rule engine can be constructed. With the rule engine, internal states extracted from the new version of the SUT can be validated. The final result after the validation is an answer to whether the new version's state is valid or not. Thus, in the end, we receive an oracle that answers the question if the system's output will be correct for the given test cases inputs.

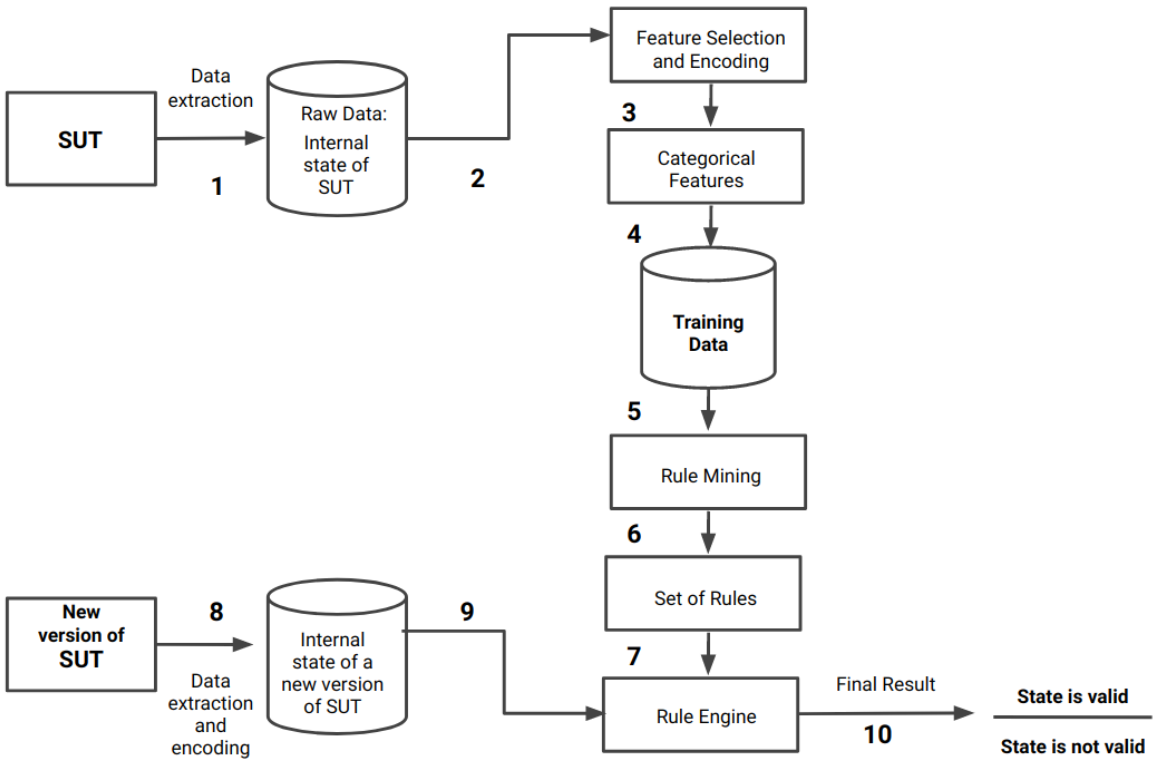

Figure 2. Overview of the method for rule-mining based oracle generation

In this research, we use Java collection classes for our experiments. The collection can be found in the project GitHub repository (see Appendix for the link). These classes are quite simple, and their behavior is well-known. The main class we experimented with is Stack. An advantage of using a Stack class as a SUT is that the results we get can be evaluated first manually since the behavior of this class is easily understandable. Also, it helps to develop the methodology and experimental setup and avoid human errors that can happen more likely for complex systems with a variety of different methods.

3.1 Data Extraction

For the machine learning model training, first, the training data should be collected. The raw dataset extraction process consists of two main steps:

1. create a test suite for the valid version of SUT
2. execute the test suite, track internal states and save raw data to the text file

3.1.1 Test Suite Generation

One of the challenges in a data collection part is to create various test suites for the class under test. Unite-tests manual writing is a challenging, tiresome, and time-consuming activity. For the experiments, we wanted to have several different test suites for every class we experimenting with. As a solution to this problem, we decided to use a framework which can generate tests automatically. In this research, we are using Randoop [PE07], [PLEB07] – one of the most popular random JUnit tests generators for Java.

Randoop relies on a feedback-directed random testing technique [PLEB07] that benefits from the feedback obtained after the execution of tests when they are created. It helps to avoid illegal and redundant inputs.

One run of Randoop outputs two test suites. The first suite contains contract-violating tests. These tests reveal such scenarios when SUT violates the API contract. Randoop contains default contracts, and custom contracts can be additionally passed as arguments when there is a need.

Another test suite contains regression tests that capture aspects of the current implementation of SUT. Regression tests help to find inconsistencies between different versions of the system. In our experiments, we use regression test suites since the goal is to create test oracles based on the correct version of SUT and to use these oracles for new versions validation.

According to the original paper [PE07], Randoop creates method sequences incrementally, by randomly selecting a method call to apply and using arguments from previously constructed sequences. When the new sequence created, it is executed and then checked against contracts. Sequences that don't show violations are output as regression tests.

While sequences that show contract violations are output as contract-violating tests. If SUT is valid, Randoop outputs only a regression test suite. To generate new sequences, only the sequences that behave normally are used.

Randoop allows us to generate test suites with several parameters passed together with a class under test as command-line arguments. Two important parameters that we use in our experiments are *testlimit* and *randomseed*.

The test limit parameter helps to limit the number of tests in the test suite. Random seed parameter allows us to produce multiple different test suites since Randoop is deterministic by default. Therefore, these two parameters allows us to generate many test suites of different size containing various test cases. Thus, we can extract several different datasets for the class under test by executing such test suites.

3.1.2 Driver Program

Another challenge in the data extraction part is to track internal states of the class under test while executing a test suite and save it to the text file for further analysis. For this purpose, we construct a special program that helps to track and save the internal state of the class under test. We call this program driver.

The main idea behind the driver is that methods of the class under test can be logically divided into two categories – the methods which change a state of the class instance and the methods which reflect the state or, in other words, give some information about the object instance. We call such methods internal state methods. Thus, if we know what methods of the class under test are internal state methods, we can construct helper class that tracks and saves the information that internal state methods return if called right after test case execution.

Consider Stack class (Figure 3) to illustrate the idea described above. Stack implementation contains seven public methods: push(), pop(), clear(), peek(), isEmpty(), size() and toString(). Method push() 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.

One can notice that methods push(), pop() and clear() modify the instance of Stack class when they are called. On the opposite, peek(), isEmpty(), size() and toString() just tell some information about the Stack class instance when they are called. Thus, in this case peek(), isEmpty(), size() and toString() are internal state methods.

We construct the driver for Stack class such that in the driver class, we create an instance of the Stack class. The driver class also contains public methods push(), pop() and clear() which call original push(), pop() and clear() using the instance of the Stack class. Additionally, driver similarly implements peek(), isEmpty(), size() and toString() but these methods are marked as private. Furthermore, the driver has a method for writing internal states to the CSV file. This method used whenever push(), pop() or

```
public class Stack{

        public Object push( Object x ){...}

        public Object pop( ){...}

        public void clear( ){...}

        public Object peek( ){...}

        public boolean isEmpty( ){...}

        public int size( ){...}

        public String toString( ){...}

}
```

Figure 3. Stack class methods

clear() methods called during tests execution.

Such a construction 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 instance of Stack class. Thus, only push(), pop(), and clear() methods will be called by the test sequences. Furthermore, the states characterized by peek(), isEmpty(), size() and toString() will be saved to CSV file while executing the test suite.

The process of internal state data extraction with the driver class and Randoop for the test suite generation presented in Figure 4. The diagram in the Figure 4 corresponds to the step 1 in the Figure 2. The process consists of four steps:

1. Produce the driver for the SUT
2. Ran Randoop on the driver class to get the test suite
3. Execute the test suite on the driver class
4. Get the resulting CSV file with state data

The driver class can be implemented for any class under test where we can separate internal state methods from other methods. It allows us to extract internal state data easily. However, manually implementing the driver class is not preferable since it would require to write the helper program for every class we want to test. This activity would take some effort and extra time and also may lead to human errors. Our solution to avoid manual writing of the drivers is to make a driver class generator using Java reflection [McI98].

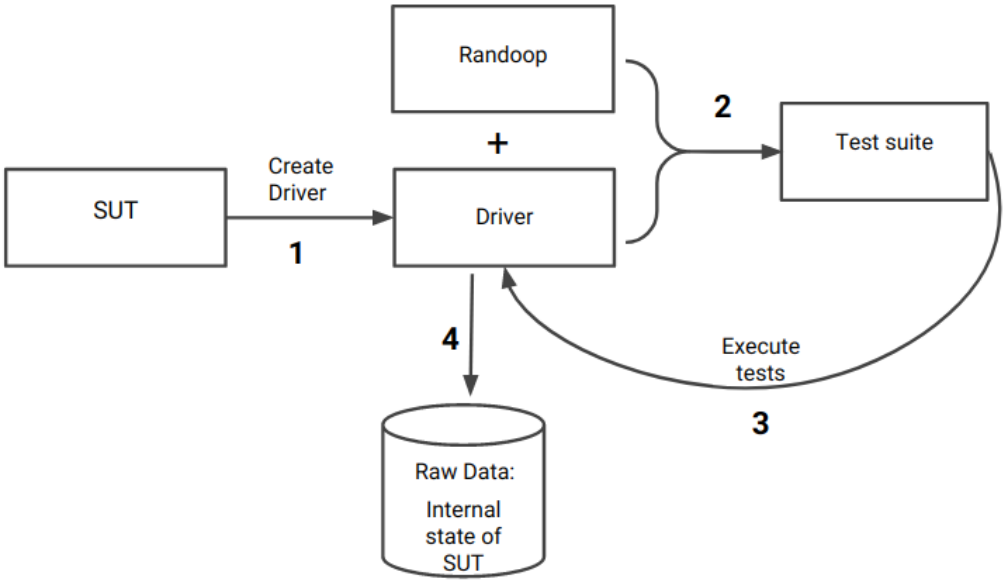

Figure 4. Internal state data extraction

Reflection is a feature in the Java programming language that allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program [McI98]. Using Java reflection, it is possible to obtain the names of all members of the class and display them. Thus, we can obtain all the methods names for the class under test. By using the names of the methods, it is possible to construct the desired functionality for the driver class.

However, our method of the driver generation with Java reflection still requires a manual indication and setting of the internal state methods. It is a separate task – to explore the possibility of automatic classification of the class under test methods as internal state methods and methods which modify the internal state of a class instance. Therefore, we decided to make a manual setting of the internal state methods in our driver generator and leave an opportunity to improve this part later if the whole oracle generation method will be recognized useful.

3.2 Rule Mining

In this section, we describe the rule mining process. The process consists of steps 2-7 on the Figure 2. First, the raw data should be prepared to use it for the machine learning model training. Second, parameters for the rule mining algorithm should be chosen with

respect to the dataset. Third, the set of rules should be explored and saved to construct the rule engine. Finally, the rule engine can be used to validate new versions of the program.

3.2.1 Feature Selection and Encoding

When the raw internal state data saved in the CSV format, the next step is to explore the data, select useful features, and make feature encoding. After that, we should be able to apply the Apriori algorithm to extract a set of rules from the data. An example of the first ten rows of the raw data for the Stack class presented in Figure 5.

size	is_empty	to_string	peek
0	True	NaN	EmptyObject
1	False	1.0	1.0
0	True	NaN	EmptyObject
1	False	1.0	1.0
0	True	NaN	EmptyObject
0	True	NaN	EmptyObject
0	True	NaN	EmptyObject
1	False	1.0	1.0
0	True	NaN	EmptyObject
0	True	NaN	EmptyObject

Figure 5. Stack raw data example

We begin the data preparation from the exploration of the data columns to decide what columns can play a role of features for our task. Consider this process on the Stack class data as an example.

Each data column (Figure 5) corresponds to one of the internal state methods. In the "size" column, all the values are numeric, "is_empty" column is categorical and contains only "True" or "False" records. The other two columns – "to_string" and "peek" are more difficult to work with in terms of machine learning algorithms. These columns contain string representations of the Stack class instance and the top of the Stack class instance, respectively.

To apply the rule mining algorithm, it makes sense to categorize the features that are not categorical already. For instance, we use a type of the object in the "peek" column instead of the original content of "peek" for the Stack class data. Thus, we have a restricted set of possible values that correspond to the "peek" column, and this feature becomes categorical.

Every class from the Java collection classes that we use for the experiments contains a toString() method. We decided to discard this feature because the strings from this column should be encoded in such a way that the machine learning algorithm would be able to interpret it properly. It is a separate task that requires additional research. Thus, we are not using this feature since we develop a new method to generate test oracles and do not want to make our initial task too complicated. Our main goal is to get the data for experiments and to check whether the proposed rule mining approach is suitable for test oracles generation.

After the feature selection, we explore every column using the data analysis capabilities of Python3 and decide on the way to encode it to apply the Apriori algorithm later on the resulting dataset.

On the Figure 6 we display a resulting dataset for the Stack class that we use for training. At the end, for the Stack class we have 3 features – "size", "is_empty" and "peek_obj_type". The first one is the original "size" column that we have seen in the raw data. We do not categorize it because the "size" column always contains some restricted set of possible values in our experiments. Since Stack class has three methods which can be called by test sequences – pop(), push(), clear() – and a number of these methods calls in Randoop will be distributed more or less uniformly, the size of a class instance will never become too big. Usually, for any size of the test suite, it will be no bigger than 6. We experimented with different sizes of test suites, and the size of the Stack always remained quite small. Thus, it makes sense to binarize the size or to leave this column as it was in the raw dataset. In this example, we leave the "size" column similar to the raw "size" column.

Looking at another two features – "is_empty" is categorical already and does not require additional engineering. Instead of the "peek" column (Figure 5) we use the type of an object in this column – "peek_obj_type". We do that to categorize and generalize the string representations of the objects in the column "peek".

3.2.2 Rule Mining Model Parameter Tuning

We use the Apriori algorithm from Efficient-Apriori [McC18] Python3 library for the rule mining. Another significant step is to study the algorithm and to explore how the parameters of this algorithm affect rules extraction.

According to Efficient-Apriori documentation [McC18], the algorithm works in two phases. Phase 1 iterates over the transactions (in our case, data rows) several times to build up itemsets of the desired support level. Phase 2 builds association rules of the desired confidence given the itemsets found in Phase 1. The desired support and confidence level thresholds can be set with min_support and min_confidence parameters of the algorithm.

Support is a frequency of which items in the rule appear together in the dataset [McC18].

size	is_empty	peek_obj_type
1	False	class java.lang.Float
0	True	EmptyObject
1	False	class java.lang.Float
0	True	EmptyObject
0	True	EmptyObject
...	...	...
1	False	class drivers.StackNewDriver
0	True	EmptyObject
2	False	class drivers.StackNewDriver
1	False	class java.lang.Float
1	False	class java.lang.Float

Figure 6. Stack training data example

Confidence is a probability of Y given X, i.e. $P(Y|X) = confidence(X \rightarrow Y)$, given a rule $X \rightarrow Y$.

Support and confidence can be calculated as follows: given a rule $\{X \rightarrow Y\}$,
$support(\{X \rightarrow Y\}) = \frac{count\ of\ \{X,Y\}}{N\ of\ rows\ in\ the\ dataset}$,
$confidence(\{X \rightarrow Y\}) = \frac{count\ of\ \{X,Y\}}{count\ of\ \{X\}} = \frac{support(\{X,Y\})}{support(\{X\})}$,
where
$support(\{X\}) = \frac{count\ of\ \{X\}}{N\ of\ rows\ in\ the\ dataset}$,
$support(\{X,Y\}) = \frac{count\ of\ \{X,Y\}}{N\ of\ rows\ in\ the\ dataset}$.

We study support and confidence parameter sensitivity to choose the thresholds for our data. We run the rule mining multiple times with support and confidence thresholds from the interval [0,1] with step 0.1 on several datasets that correspond to different test suites. As a result, we get support and confidence sensitivity plots where we display the parameters thresholds on the x-axis and the number of extracted rules on the y-axis. We also compare sets of rules extracted from the datasets.

Thus, we were able to determine such thresholds using that we can extract the same rules from the dataset corresponding to any test suite. Therefore, we can get a set of rules that should hold for the data extracted with any test suite for the current class under test.

3.2.3 Extraction of the Set of Rules

We apply the Apriori algorithm with the found support and confidence thresholds to our dataset considering data rows as transactions. Since we found such thresholds with

which the set of rules will be similar for any test suite, we use one of the datasets for rules extraction. For example, for Stack, we used data corresponding to the test suite with random seed 0 and the test limit of 8000.

We use such a test limit because we first determined what an optimal suite size limit for the class under test is. Since Randoop has limited capabilities to generate test inputs, we can obtain such a test suite that will be big enough, and no new rows will occur in the resulting dataset if we would increase the test limit.

We conducted experiments with datasets for several different test suites with a gradually growing test limit parameter and with different random seeds. We extracted rules with zero support and confidence from these datasets. After some test limit threshold, the set of rules remained the same. Thus, simply no new combinations of data were generated by Randoop, and it is possible to find such a test limit that would give us all possible combinations of data rows for current seed. For example, for the Stack class, this threshold was about 6000 tests in the test suite for three different random seeds. For the Stack class, we decided to limit our test suite size to 8000. With this threshold, we can be sure that all the useful data would be extracted for any random seed.

We extract rules from the dataset corresponding to the test suite of the defined test limit and any random seed. We use the support and confidence thresholds described above. Then, we save the extracted set of rules to pickle file for experiments.

3.3 Method Application

To apply our rules to the new version of SUT, we first have to extract new internal state data. A diagram of this process has shown on the Figure 7. This diagram corresponds to the step 8 in the Figure 2. To extract and save the data, we utilize the same driver as for the valid version of SUT. We execute existing test cases on the driver and receive raw data in the CSV format. Then, we make the same feature selection and encoding as for the SUT version that was used for rules extraction (the valid SUT version).

To check how well the method performs, we introduce defects to the class under test. For example, in the Stack class we use three internal state methods – peek(), size() and isEmpty(). We modify each of them individually and also make all possible combinations of these modifications. Thus, for the Stack we get seven modifications:

Mod1: peek()

Mod2: isEmpty()

Mod3: size()

Mod4: Mod1 + Mod2 + Mod3

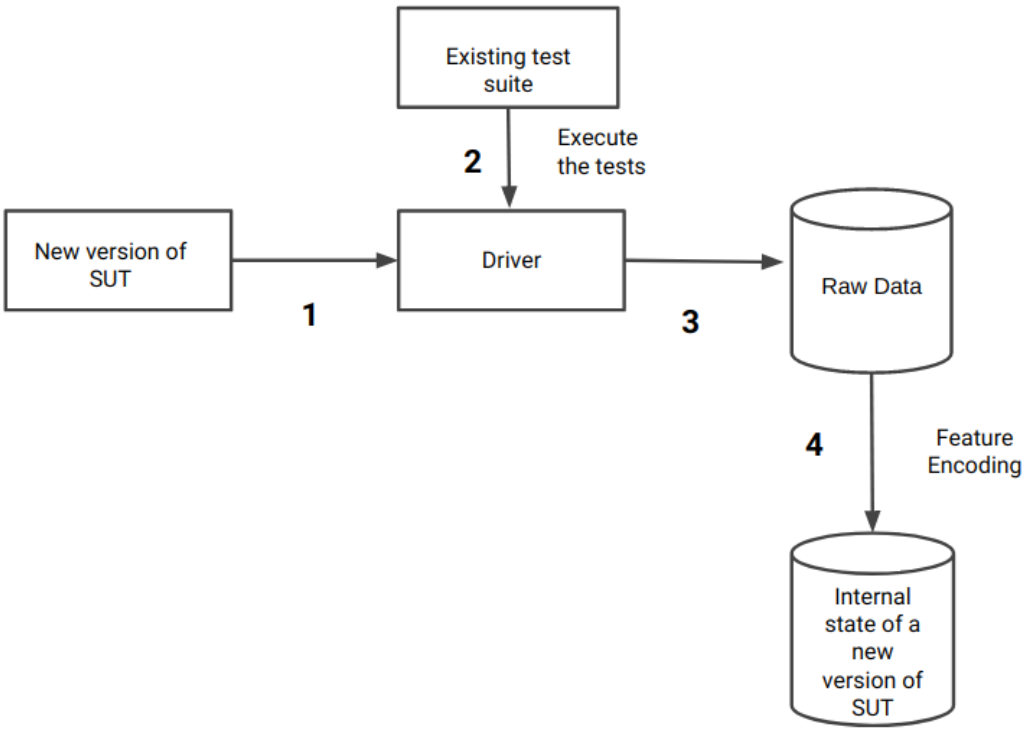

Figure 7. New version of the SUT. Internal state data extraction

Mod5: Mod1 + Mod2

Mod6: Mod1 + Mod3

Mod7: Mod2 + Mod3

We injected defects such that some of them will lead to apparent faulty behavior in the internal states while some are not so easily observable from the data but still make the method not correct. Thus, we will be able to observe in what cases the extracted rules cover the defects, and in what cases the rules are not strong enough to detect the defect.

3.3.1 Internal State Validation

The next stage is to validate the internal state of a modified system with our rules. For that, we construct a rule engine from the set of rules with the help of rule-engine [McI19] Python3 library. This library allows creating rule objects from logical expressions. These

rules then can be applied to arbitrary objects to evaluate whether they match or not [McI19]. It helps to efficiently utilize the initial rules to validate the internal state of a new version of SUT. Thus, our rule engine is a set of rule objects constructed from the initial set of rules.

To validate the internal state data, we first take a rule from our set of rules. Denote this rule $\{X \rightarrow Y\}$. In this example, X is a left-hand side (LHS) of the rule, and Y is a right-hand side (RHS). The Efficient-Apriori [McC18] library has a functionality to extract the LHS and RHS of the rule.

We take LHS of the rule and select the records that match the LHS from the dataset we want to validate. Then, we check whether the whole rule matches these records or not. Thus, we would know if some records (data rows) matched the LHS part and do not match the RHS part. We consider such a case as the rule violation, and therefore, this record will be considered as not valid.

We conduct this procedure for every rule from the extracted set. It can be summarized in the following points:

1. Pick a rule from the set of rules
2. Take LHS
3. Select records which match LHS
4. Check whether these records match the rule
5. Print out/save the records which doesn't match
6. Repeat steps 1-5 for every rule

At the end, we will know whether the tested internal state dataset contains violated records or not. If it contains violations, then the new version of SUT is not correct.

Since rules are derived from the dataset of internal states that correspond to the valid version of SUT and every record in this dataset is valid, the rules will not show the violations in the original state. In the data that corresponds to the modified SUT, some records may be similar to the valid version records, and some may not. Thus, it makes sense to validate only the records from the new internal state data that did not occur before in the valid state data. Therefore, we use for the validation only the records that we have not seen in the training dataset. It makes the validation process faster.

3.3.2 Assessment of the Rules

Finally, we have to assess the performance of our rule-based method. We should evaluate how strong the rules are, how well they cover the dataset, what defects they can detect, and what they can not, etc.

For this purpose, we make the following analysis:

1. We calculate the percentage of rows in the training dataset that are covered by the rules.
2. For unique data rows in the training dataset, we check how many rules match every row. We display these rows with an addition of a column, which for every row shows the number of rules matched.
3. We check the number of new rows in every dataset that corresponds to the modification.
4. We check and display the number of unique rows in the new rows.
5. We check and display the number of unique rows in the rows from the modified dataset that existed in the valid dataset.
6. We apply rules to the new rows and observe what data rows are incorrect.
7. We check the rules that were violated.
8. We check what rows in the new rows are covered by the rules and what are not covered.
9. We calculate the percentage of covered and not covered rows in the new rows.
10. We check how many times every rule matched during the validation.
11. We check what modifications the rules are able to detect and what they are not.

After that steps, we can assess how strong our rules are, what modifications we can detect with this set of rules, and what we can not. Then, we can make conclusions about the rule mining method performance.

4 Experiments and Results

In this chapter, we provide a report for the experiments described in the Methodology. We are experimenting with the Stack class from the Java collection classes.

Section 4.1 provides experiments described in the section "Rule Mining Model Parameter Tuning" (3.2.2) from the Methodology chapter.

Section 4.2 gives a report for the rules extraction process described in the section "Extraction of the Set of Rules" (3.2.3). It presents extracted set of rules for the Stack class.

Section 4.3 provides analysis described in the section "Method Application" (3.3).

4.1 Support and Confidence Parameters Sensitivity

To apply the Apriori algorithm from Python3 Efficient-Apriori [McC18] library, we first have to set minimum support and minimum confidence parameters suitable for our data. To choose the minimum support and minimum confidence thresholds for the Stack class data, we conduct experiments to study the support and confidence parameter sensitivity.

For this purpose, we generated Randoop test suites of random seeds 0, 20, and 100, and the test limit of 50000 for the Stack class. Thus, we have three different datasets to compare the behavior of the Apriori algorithm on the Stack class data. We choose such a big test limit to make sure that for every random seed, we will generate all the possible tests since Randoop has it is own stopping criteria for the test suit size. Thus, if we choose a big enough test limit parameter, Randoop would stop by itself. If we use a big enough test limit every time for every random seed, then for a particular random seed, we should receive the same dataset for every run of test suite generation.

Rules were extracted from these three datasets with min_support thresholds from the interval [0,1] with a step 0.1 and min_confidence=0 and otherwise, min_confidence from the interval [0,1] with a step 0.1 and min_support=0. The number of rules extracted for every minimum support and minimum confidence threshold was saved. After that, we produced support and confidence sensitivity graphs. These graphs have shown on the Figures 8 and 9 respectively.

As it can be seen from the Figures 8 and 9, the graphs for datasets of different seeds are quite similar in both cases – for the support and confidence. Also, these two parameters are dependent on each other, as one can notice from the formulas in section 3.2.2.

From these graphs, we can conclude that the algorithm, in combination with our data for the Stack class, is quite sensitive to the changes in parameters, especially in the case of minimum support. When increasing the support threshold, the number of rules drops

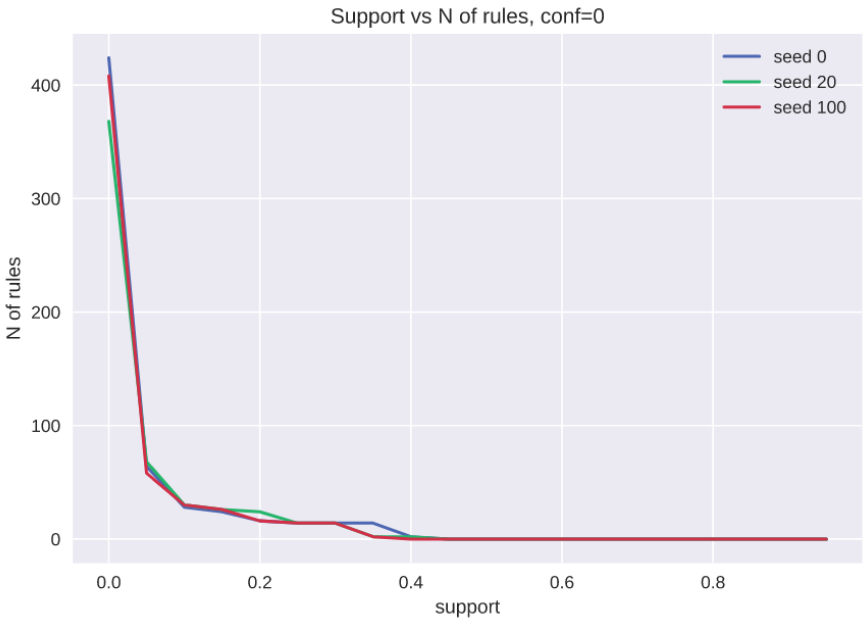

Figure 8. Support parameter sensitivity graph

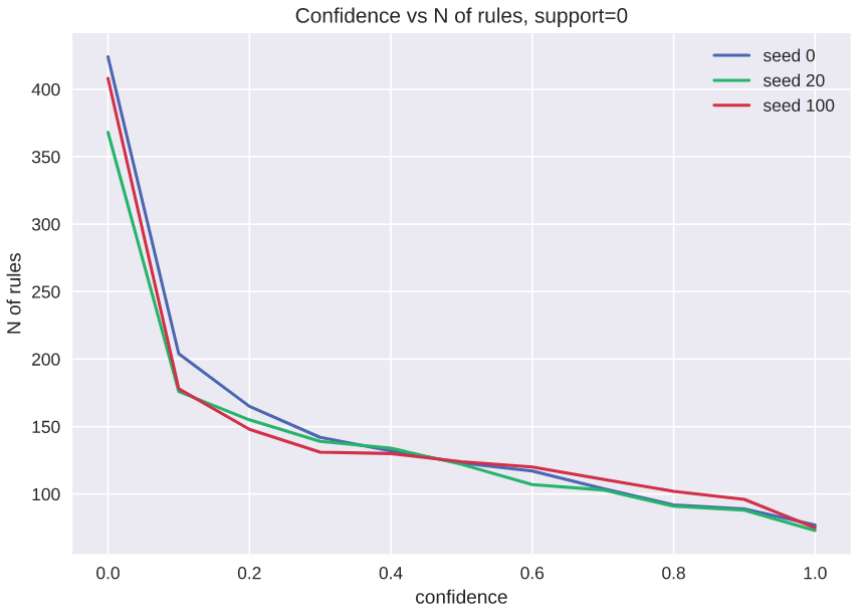

Figure 9. Confidence parameter sensitivity graph

very fast, and after the threshold of 0.4, no rules were extracted. The number of rules on the confidence graph decreases more smoothly.

One can notice that the lines on support and confidence sensitivity graphs intersect in several places and then converge. It gave us an idea that since the number of extracted rules is the same, for example, for the confidence of 0.4, it could be the same set of rules for every random seed dataset. Thus, it may be possible to find such minimum support and minimum confidence parameters combination that would give us the same set of rules for any test suite for the current class under test.

We checked this hypothesis, and indeed we were able to find such thresholds that the extracted set of rules was the same for every Stack class dataset when using these thresholds. Usage of such a set of rules gives us confidence that the extracted rules will be true for any test suite and therefore, we would be able to validate the class under test data that corresponds to any test suite, not only to the suite we used for rules extraction from the valid data.

4.2 Rules Extraction from the Stack Class Dataset

For the rule engine construction for our class under test, we extract a set of rules from the data that corresponds to the valid version of the Stack class. First, we determine such thresholds for support and confidence parameters, that the rule mining algorithm extracts the same set of rules from any of the datasets that correspond to different Randoop-generated test suites.

For that, we have generated test suites with a test limit of 8000 and random seeds 0, 10, 15, 20, 30 for Stack. Then, we collected datasets for these test suites. We gradually increased support and confidence parameters starting from 0.1 with a step 0.1 and checked how many rules were extracted for every dataset on each step. When the number of extracted rules in the set became the same for every test suite, we stopped. After that, we checked that the sets of rules were similar to each other for every dataset. We saved these support and confidence thresholds. For the Stack class for the data as in Figure 6 these thresholds are 0.3 for support and 0.9 for confidence. After that, we saved this set of rules to the pickle file for future use. The set of 13 rules displayed on the Figure 10.

As it can be seen from the Figure 10, almost all the rules seems do be overlapping and thus, some of the rules looks redundant. However, the rules, for example, $\{size == \text{"0.0"}\} \rightarrow \{is\_empty == \text{"}true\text{"}\}$ and $\{is\_empty == \text{"}true\text{"}\} \rightarrow \{size == \text{"0.0"}\}$ are actually two different rules and we should explore whether these rules are redundant or not.

```
1. {size == "1.0"} -> {is_empty == "false"}
2. {size == "0.0"} -> {is_empty == "true", peek_obj_type == "emptyobject"}
3. {peek_obj_type == "emptyobject"} -> {is_empty == "true", size == "0.0"}
4. {is_empty == "true"} -> {peek_obj_type == "emptyobject", size == "0.0"}
5. {peek_obj_type == "emptyobject"} -> {size == "0.0"}
6. {size == "0.0"} -> {peek_obj_type == "emptyobject"}
7. {peek_obj_type == "emptyobject", size == "0.0"} -> {is_empty == "true"}
8. {is_empty == "true", size == "0.0"} -> {peek_obj_type == "emptyobject"}
9. {is_empty == "true"} -> {peek_obj_type == "emptyobject"}
10. {peek_obj_type == "emptyobject"} -> {is_empty == "true"}
11. {is_empty == "true", peek_obj_type == "emptyobject"} -> {size == "0.0"}
12. {size == "0.0"} -> {is_empty == "true"}
13. {is_empty == "true"} -> {size == "0.0"}
```

Figure 10. Set of rules extracted from the Stack class data

We are going to use this set of rules for the new versions of Stack class validation. Consider a case when something went wrong with the new version of the Stack class, and it works not in the correct way. Imagine that the new version dataset contains such rows:

1. size="0.0", is_empty="false", peek_obj_type="emptyobject"
2. size="3.0", is_empty="true", peek_obj_type="emptyobject"

In this case, these two rules would not be redundant since the rule $\{size == "0.0"\} \rightarrow \{is\_empty == "true"\}$ can detect that something is wrong in 1 but cannot detect that something is wrong in 2 since the rule LHS does not match 2. On the opposite, the rule $\{is\_empty == "true"\} \rightarrow \{size == "0.0"\}$ can detect that something is wrong in 2 but cannot do the same in 1 because again, LHS does not match.

Therefore, we should keep all the rules in the set and analyze it as described in section 3.3.

4.3 Rules Application and Analysis

In this section, we describe experiments with analysis and application of the set of 13 rules for the Stack class. First, we describe our analysis of the original dataset that we used for the rules extraction. We call it a valid dataset. Second, we provide the analysis regarding all the Stack class modifications described in section 3.3.

4.3.1 Valid Dataset Analysis

To check how strong our rules are and to analyze the performance of the rules, we apply the set of rules to the valid dataset and analyze its behavior on these data. Here we cover points 1 and 2 from the section 3.3.2.

Valid dataset analysis visualization displayed in the Figure 11. Valid dataset (test suite: test limit = 8000, random seed = 0) consists of 111325 rows, nearly 80% of these rows are covered by the rules (Figure 10). The dataset contains many duplicate rows. The number of unique rows in the data is 48, and 27% of these rows are covered.

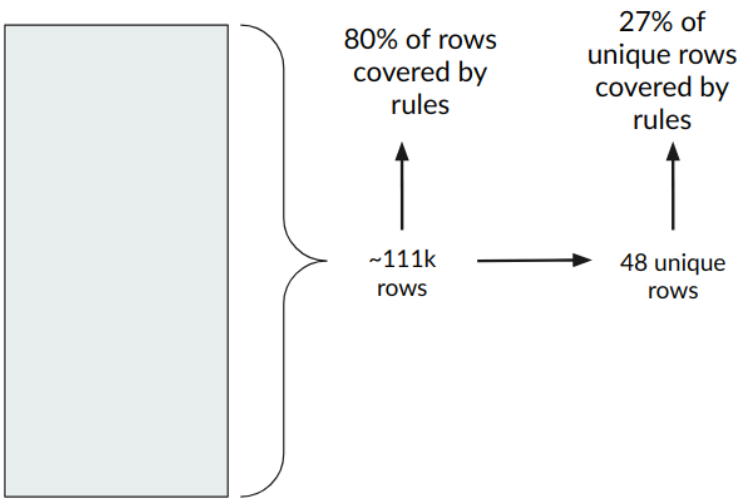

Figure 11. Valid dataset coverage

In Figure 12 we show the 48 unique rows from the valid data. Rows are sorted in the descending order by the number of occurrences. In the column "N_matched_rules" we show how many rules from the set of 13 rules are matched the corresponding data row.

From the column "N_matched_rules" we can observe that 12 rules from the set match the first row – the one with the highest number of occurrences. The other rows match whether 1 rule or 0. Thus, 12 rules from our set are overlapping in some sense, and one

rule does not intersect with other rules. We can see that also from the set of rules in Figure 10.

size	is_empty	peek_obj_type	N_matched_rules	occurence_count
0.0	true	emptyobject	12	44056
1.0	false	class java.lang.float	1	18972
1.0	false	class drivers.stacknewdriver	1	6791
1.0	false	class java.lang.class	1	6162
2.0	false	class java.lang.float	0	5869
1.0	false	class java.lang.long	1	5781
2.0	false	class java.lang.boolean	0	3651
2.0	false	class drivers.stacknewdriver	0	2933
1.0	false	class java.lang.byte	1	2733
1.0	false	class java.lang.string	1	2675
3.0	false	class java.lang.class	0	2166
1.0	false	class java.lang.short	1	2130
3.0	false	class drivers.stacknewdriver	0	2013
2.0	false	class java.lang.class	0	1585
4.0	false	class drivers.stacknewdriver	0	824
2.0	false	class java.lang.double	0	737
3.0	false	class java.lang.double	0	335
2.0	false	class java.lang.short	0	310
2.0	false	class java.lang.long	0	185
3.0	false	class java.lang.float	0	177
3.0	false	class java.lang.boolean	0	151
2.0	false	class java.lang.string	0	142
4.0	false	class java.lang.class	0	137
2.0	false	class java.lang.byte	0	135
2.0	false	class java.lang.integer	0	99
3.0	false	class java.lang.long	0	92
3.0	false	class java.lang.byte	0	75
5.0	false	class drivers.stacknewdriver	0	70
4.0	false	class java.lang.float	0	70
1.0	false	class java.lang.integer	1	42
4.0	false	class java.lang.short	0	41
1.0	false	class java.lang.double	1	35
4.0	false	class java.lang.boolean	0	31
3.0	false	class java.lang.integer	0	22
1.0	false	class java.lang.boolean	1	20
1.0	false	class java.lang.object	1	19
5.0	false	class java.lang.class	0	17
1.0	false	class java.lang.character	1	6
4.0	false	class java.lang.byte	0	6
3.0	false	class java.lang.object	0	6
6.0	false	class java.lang.long	0	6
3.0	false	class java.lang.character	0	5
3.0	false	class java.lang.string	0	5
5.0	false	class java.lang.boolean	0	2
4.0	false	class java.lang.long	0	2
6.0	false	class drivers.stacknewdriver	0	2
6.0	false	class java.lang.class	0	1
5.0	false	class java.lang.short	0	1

Figure 12. Valid dataset, unique rows

Overall, we can observe quite good coverage for the whole dataset (∼80%), but coverage of the unique rows is much worse. Thus, the validation with the rules will work well only in some cases, which are included in the 27% of covered unique data. These are the rows that will appear in the dataset with a very high probability. However, other rows that may also hide undesirable defects will remain uncovered. To prove that, we check our method on the defective modifications of the Stack class.

4.3.2 Modifications Analysis

To check our rules performance, we inject defects to the Stack class as described in the section 3.3. Original vs modified peek(), isEmpty() and size() methods displayed in the Figures 13-15.

As one can notice, some of these modifications affect internal state such that it will be quite easy to detect that something is wrong because the states obviously will look incorrect. For example, isEmpty() method (Figure 14) 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 obviously faulty state size="2.0", is_empty="true", peek_obj_type="some object type". However, the other two modifications are more tricky. The modification of peek() will return not an 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.

```
public Object peek( )
{
    if( isEmpty( ) )
        throw new EmptyStackException( );

    return items.get( items.size( ) - 1 );
}
```

```
public Object peek( )
{

    if( isEmpty( ) )
        throw new EmptyStackException( );

    if (items.size() >= 2){
        return items.get( items.size( )-2);
    }
    else {
        return items.get(items.size() - 1);
    }

}
```

Figure 13. Original vs modified peek() method

```java
public boolean isEmpty( )
{
    return size( ) == 0;
}
```

```java
public boolean isEmpty( )
{
    if (size( ) == 2){
        return size( ) == 2;
    }
    else {
        return size() == 0;
    }
}
```

Figure 14. Original vs modified isEmpty() method

```java
public int size( )
{
    return items.size( );
}
```

```java
public int size( )
{
    if (items.size() > 0){
        return items.size( )+1;
    }
    else {
        return items.size();
    }
}
```

Figure 15. Original vs modified size() method

We explore the modifications that have shown in Figures 13-15 and also all possible combinations of these modifications. Further in this section, we cover points 3-11 from section 3.3.2. The visualizations corresponding to the points 3-11 from section 3.3.2 can be found in the Appendix, section "Appendix to Chapter 4", Figures 17-30.

We can see from the Figures 17-30 that some violations were detected by the rules and some were not. Rules violations were found in the Modification 2 – isEmpty() – and in the modifications that are combinations of the modified isEmpty() and other methods. Thus, with our method, we were able to detect only the modifications that give obvious incorrect states.

Table 2. Summary of Modifications 1-7 analysis

Modification .	**N of rows**	**N of new rows**	**N of old rows**	**N of unique new rows**	**N of unique old rows**	**N of rules LHS matched new unique rows**	**N of unique rows covered by rules**	**N of rules violated**
1. peek	111325	465	110860	5	41	0	0	0
2. isEmpty	56546	9605	46941	1	24	7	1	5
3. size	111325	798	110527	16	32	0	0	0
4. peek+size+isEmpty	665	665	0	1	0	7	1	5
5. peek+isEmpty	56546	9870	46676	3	24	7	1	5
6. peek+size	111325	3362	107963	15	31	0	0	0
7. isEmpty+size	10697	7741	2956	2	4	7	2	5

Table 3. Rules matched and rules violated (rules from Figure 10)

Modification .	**Rules that's LHS matched new unique rows**	**Rules that were violated**
2. isEmpty	3, 4, 5, 9, 10, 11, 13	3, 4, 5, 11, 13
4. peek+size+isEmpty	2, 4, 6, 8, 9, 12, 13	2, 4, 6, 8, 9
5. peek+isEmpty	3, 4, 5, 9, 10, 11, 13	3, 4, 5, 11, 13
7. isEmpty+size	3, 4, 5, 9, 10, 11, 13	3, 4, 5, 11, 13

Table 2 summarizes Figures 17, 19, 21, 23, 25, 27, 29. Table 3 complements Table 2 and shows what rules matched and what rules were violated for the modifications 2, 4, 5, 7, i.e. for the modifications where violations were detected.

We notice from Table 2 that the datasets that correspond to such modifications where violations were not detected have the same number of rows as in the valid dataset. On the opposite, datasets that correspond to the modifications where violations were detected, have much fewer rows than the valid dataset. The explanation of that is the following. When some tests from the test suite are failed during the suite 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 similar to the valid data since we execute the same test suite both for the valid and for the modified data extraction. This means that our set of rules detected only such modifications for that the test suite already showed that class under test is incorrect.

The question is why we were still able to detect some faulty rows in the data by our set of rules after the tests failed. Does it mean that the test suite was not strong enough, and will the correction of the defects found by the test suite lead to the fully correct class, and no rules will be violated anymore?

Consider the implementation of pop() from the Stack class (Figure 16). We can notice that the implementation of pop() depends on isEmpty(). Method pop() uses isEmpty() internally and throws an EmptyStackException if the Stack size is equal to 2 if we consider Modification 2. Therefore, since pop (that is called by the tests from the test suite) depends on the implementation of isEmpty(), changes to isEmpty() will probably change the behavior of pop, and that might be detected by the tests.

```
public Object pop( )
{
    if( isEmpty( ) )
        throw new EmptyStackException( );

    return items.remove( items.size( ) - 1 );
}
```

Figure 16. Method pop(), Stack class

However, not only the tested methods depend on the internal state methods, but internal state methods are also dependent on each other. For example, implementation of isEmpty() uses size() method. Randoop does not create assertions for the private method calls, so rows like the row "size=2.0, is_empty=true, peek_obj_type=emptyobject" (Figure 30) were not detected by tests but were detected by the rules. However, this faulty row was caused by the same defects in the class under test as the faulty rows that were detected by the test suite. Thus, the tests already detected these modifications, and the test suite was not weak. If we would fix the defect found by the test suite before checking

with the rules, we assume that no rules would be violated. Therefore, the rule method is redundant in this case.

We conclude that this shows the weakness of our approach since we restrict the callable interface of the SUT via the generated adapter (the driver class). Despite that, we were able to construct a partial test oracle for the Stack class using our approach since we were able to detect the same violations that were detected by the Randoop test suite, and since we generated the set of rules that can be used for validation of some cases. However, it seems to be redundant in this case because the rules are too simplistic to be able to detect some tricky defects.

5 Discussion

In this thesis, we studied the test oracle problem. We provided a survey of the literature on test oracle automation and showed that there is a lack of methods for test oracle automatic generation. We also provided literature on the application of machine learning for test oracle automation and association rule mining algorithms.

We presented a hypothesis about the possibility of using the internal state of the SUT to derive test oracles. In particular, we explored a possibility to generate test oracles from the internal state data in the form of rules using association rule mining technique.

We developed a methodology for test oracle generation using association rule mining and internal states of the SUT extracted during the test suite execution. We provided experimental setup to assess the rule-based method and conducted experiments on the Stack class from the Java collection classes. We analyzed our method performance. Further, in this chapter, we list and discuss our observations and findings and our study limitations.

5.1 Lessons Learned

Our experiments on the Stack class indicate that it is possible to validate the modified version of the class under test with the rule-based method presented in this research. However, the analysis showed that it was possible to detect only quite simple violations in the Stack class. These violations were also detected by the test suite that we used to extract internal states from the class under test. Unfortunately, some of the Stack class modifications where we injected more tricky defects, remained undetected neither by the test suite nor by the extracted rules. Therefore, we were able to construct the test oracle that can only partially cover a need for an oracle for the class under test.

One of the positive outcomes of our approach is the following. Our method is based on the association rule mining technique, i.e., an unsupervised machine learning technique. Therefore, our approach does not require to collect the annotated dataset as it is for the models described in the section 2.1.2. It makes the process of data collection much easier.

Another advantage of our method is that unlike the models described in the section 2.1.2, 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 a mixed type data. Thus, our method can be generalized for the inputs of any type, not only for integers. It removes some limitations from the type of the SUT.

One of the disadvantages of our method is the driver class that we use to extract internal state data during the test suite execution. To generate the driver for the class under test, we have to indicate internal state methods manually. However, for efficiency reasons, it would be better to have an automatic indication of internal state methods. Unfortunately, there is no simple way to indicate it automatically. Also, in the case of

the manual indication, for some classes, it may not be so clear what methods should be marked as internal state methods. Then, the person that makes the indication should decide what methods can give useful information about the class under test, and it can lead to not an optimal solution.

Another disadvantage is that the driver restricts the callable interface of the class under test. In the driver class, we mark internal state methods as private ones. Randoop does not generate tests for these methods and, consequently, these methods will not be tested. Therefore, if the new version of SUT has a defect in the internal state method, it may lead to the situation when this defect will not be detected. It happened for some of our Stack class modifications. Thus, the driver construction reduces the power of the test suite. If the driver class were written differently, the method could give a completely different result.

One more disadvantage of our approach is that the one who replicate such a method for some SUT should decide how to encode and represent the raw internal state data accurately. For example, for the Stack class, we decided to drop the possible feature "to_string" because it is hard to represent in a meaningful way for the machine learning algorithm. Also, we had to categorize some features. Some of the features such as "is_empty" were already categorical but for some of them we had to define the categories to map the raw data. For instance, we mapped the "peek" feature for the Stack class to categories using the type of the object in the "peek" column ("peek_obj_type" feature). We decided to encode it in such a way because it was the simplest way to categorize it. However, maybe it was not an optimal solution, and maybe there exists a way to encode this feature better. Unfortunately, we do not have strict criteria for the best solution, and thus, it can lead to not the optimal feature engineering.

The rule-based method replication will require additional time for internal state methods manual indication, feature selection, feature engineering from the person who does it for some SUT. These tasks are not easy since it requires to make the decisions that can affect the overall result and the method performance. It is a disadvantage since we want to automate the test oracles generation, but in our case, one still has to spend some time on these decisions.

We summarize the findings described above in the following points:

1. We were able to construct partial test oracle for the Stack class using the rule-based method.

2. The rule mining method uses an unsupervised machine learning algorithm for the oracles extraction. Therefore, there is no need to collect an annotated dataset as for the classification or regression tasks.

3. The rule mining method can be applied to the SUT with any type of input data.

4. The driver class generation requires the manual indication of internal state methods for the class under test.

5. The driver class restricts the callable interface of the SUT, which reduced the power of the test suite.

6. The data preprocessing and feature selection for the rule mining are quite complex tasks that may require some time and decision-making power. Also, there are no criteria for an optimal feature encoding.

5.2 Limitations

We highlight several limitations of our study of the rule-based method for test oracles generation.

First, the method was tested only on a single class under test. Furthermore, we choose quite a simple class with well-known behavior and a few class methods. If we would test the method on a different simple class or a different class with more complex relations between methods, we could observe more interesting insights during the experiments.

Second, the defects injection to the class under tests were made manually. Our idea for further studies is to combine the method with mutation testing [JH11]. Using mutation testing, we could automatically generate mutants, i.e., faulty versions of the original program. Thus, more modifications of the class could be made, and more phenomena could be visible.

Third, the set of rules we used for the class validation is quite small, and many rules are overlapping. It happened because of our method of selecting the rules. We found such thresholds for minimum support and minimum confidence that the rules in the set would be valid for any test suite that we use for the data extraction. However, if we use lower threshold values, more rules would be generated, and maybe it would be possible to receive a more powerful set of rules. If we decrease support and confidence thresholds, the set of rules would be different for every test suite. Therefore, we could validate with these rules only the data that were extracted using the test suite for which the rules set was generated. Then, the way of the class validation should have been modified according to this rules set property.

Thus, these are three points that could be explored further, and that could give us more insights on the rule mining method for test oracles generation. Furthermore, the driver class could be constructed in a different way. It also may change the results and should be mentioned as an opportunity for further exploration.

6 Conclusion

In this thesis, we studied the test oracle problem and different methods to approach this problem. We proposed a hypothesis on how to alleviate or solve the problem using internal states of the SUT and the association rule mining algorithm. We checked our hypothesis by developing the methodology and conducting experiments on a simple Java class.

We showed that the test oracle automation is a very important aspect of the software engineering process, and there is a lack of the generic methods to derive test oracles automatically. We also provided some literature on approaching this problem with supervised machine learning methods. We provided a review of association rule mining algorithms and motivated our choice of the algorithm in the review.

We presented a methodology for test oracle automatic generation using internal states of the SUT extracted during the test suite execution. We created test oracles in the form of rules using the Apriori algorithm. We created an experimental setup and provided experiments aimed to indicate whether our method is useful or not and to show the limitations of the method. We conducted our experiments using Stack class from the Java collection classes.

We discussed the advantages and disadvantages of the rule-based method. We showed that we were able to construct partial test oracle for the Stack class. Furthermore, our method has such advantages as a possibility to apply it to the SUT with different types of input data and that there is no need to collect an annotated dataset. However, we concluded that the driver class that we use for internal state data extraction affects the performance of our method because it restricts the callable interface of the SUT.

We listed the limitations of our study and the directions for further exploration. The limitations of our study are that the rule-based method was tested only on one class under test. Furthermore, the defects in the modifications of the class under test aimed to assess the method performance were injected manually. Finally, the selection of the rules could be made in a different way.

Thus, the perspectives of further research are as follows. The driver program could be written in a different way to avoid the problem with the restriction od the callable interface of the SUT. The rule-based method could be tested on other, maybe more complex classes. Additionally, defects in the classes could be injected using mutation testing to avoid the manual method. Finally, the method for rules selection could be reconsidered and modified in order to get a more powerful set of rules.

Taking into account all the said above, we conclude that we successfully developed the methodology for mining test oracles using the internal state of the SUT. We were able to construct partial test oracle for the Stack class. However, the rule-based method has several disadvantages, and further research should be done due to the study limitations.

7 Acknowledgments

This thesis has been supported by Software Competence Center Hagenberg GmbH.

I would like to thank my supervisor from the University of Tartu, Dietmar Pfahl, for his continuous support, guidance, and helpful discussions. Also, I would like to thank Rudolf Ramler, my co-supervisor from Software Competence Center Hagenberg, for the idea of this project and his guidance, especially in the first stages of the project. Besides, I would like to express my gratitude to Claus Klammer for the technical support and the implementation of driver generation.

References

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 09 1994.

[BNR+18] Ronyerison Braga, Pedro Neto, Ricardo Rabêlo, José Santiago, and Matheus Souza. A machine learning approach to generate test oracles. In *Proceedings - Conference: the XXXII Brazilian Symposium*, pages 142–151, 09 2018.

[DDB+19] Vinicius H. S. Durelli, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego Dias, and Marcelo P. Guimarães. Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212, 2019.

[Dil00] Laura K. Dillon. Automated support for testing and debugging of real-time programs using oracles. *ACM SIGSOFT Software Engineering Notes*, 25(1):45–46, 2000.

[DW81] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *In Proceedings of the ACM'81 Conference*, pages 254–257. ACM, 1981.

[ETBY15] Phil McMinn Muzammil Shahbaz Earl T. Barr, Mark Harman and Shin Yoo. The oracle problem in software testing: A survey. In *IEEE Transactions on Software Engineering*, volume 41, pages 507–525. IEEE, 2015.

[Hie11] Robert M. Hierons. Oracles for distributed testing. In *IEEE Transactions on Software Engineering*, volume 38, pages 629–642. IEEE, 2011.

[How78] William E. Howden. Theoretical and empirical studies of program testing. In *IEEE Transactions on Software Engineering*, volume 4, pages 293–298. IEEE, 1978.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[JWC+08] Hu Jin, Yi Wang, Nian-Wei Chen, Zhi-Jian Gou, and Shuo Wang. Artificial neural network for automatic test oracles generation. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 727–730, 2008.

[KM16] Divya Kumar and K.K. Mishra. The impacts of test automation on software's cost, quality and time to market. In *Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016*, pages 8–15. ScienceDirect, 2016.

[LO17] Nan Li and Jeff Offutt. Test oracle strategies for model-based testing. *IEEE Transactions on Software Engineering*, 43(4):372–395, 2017.

[McC18] Glen McCluskey. Efficient-apriori documentation. https://efficient-apriori.readthedocs.io/en/latest/, 2018.

[McI98] Spencer McIntyre. Using java reflection. https://www.oracle.com/technical-resources/articles/java/javareflection.html, 1998.

[McI19] Spencer McIntyre. Rule engine documentation. https://zerosteiner.github.io/rule-engine/, 2019.

[PE07] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, 2007.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, 2007.

[PMH10] Mark Stevenson Phil McMinn and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *International Workshop on Software Test Output Validation (STOV 2010)*, pages 1–1. ACM, 2010.

[SAS13] Phil McMinn Sheeva Afshan and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 352–361. IEEE, 2013.

[Sch] Rolf Schwitter. English as a formal specification language. In *Proceedings. 13th International Workshop on Database and Expert Systems Applications*.

[SP15] Surbhi K. Solanki and Jalpa T. Patel. A survey on association rule mining. In *2015 Fifth International Conference on Advanced Computing Communication Technologies*, pages 212–216, 2015.

[TR14] Karthikeyan Thirunavu and N. Ravikumar. A survey on association rule mining. *International Journal of Advanced Research in Computer and Communication Engineering*, 3, 01 2014.

[TYCY98] S. C. Cheung Tsong Yueh Chen and S. W. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science The Hong Kong University of Science and Technology, 1998.

[VLK02] Meenakshi Vanmali, Mark Last, and Abraham Kandel. Using a neural network in the software testing process. *International Journal on Intelligent Systems*, 17:45–63, 01 2002.

[VSB14] Vineeta, Abhishek Singhal, and Abhay Bansal. Generation of test oracles using neural network and decision tree model. *Proceedings of the 5th International Conference on Confluence 2014: The Next Generation Information Technology Summit*, pages 313–318, 11 2014.

[WYW11] Farn Wang, Li-Wei Yao, and Jung-Hsuan Wu. Intelligent test oracle construction for reactive systems without explicit specifications. In *Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011*, pages 89–96, 12 2011.

[Xie06] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proceedings - ECOOP 2006 – Object-Oriented Programming: 20th European Conference*, pages 380–403, 07 2006.

[XKK+10] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 257–266, 01 2010.

[YH10] Shin Yoo and Mark Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 2010.

[ZT02] Du Zhang and Jeffrey J.P Tsai. Machine learning and software engineering. In *Software Quality Journal - SQJ*, volume 11, pages 22 – 29, 02 2002.

Appendix

I. Appendix to Chapter 4

Modification 1: peek()

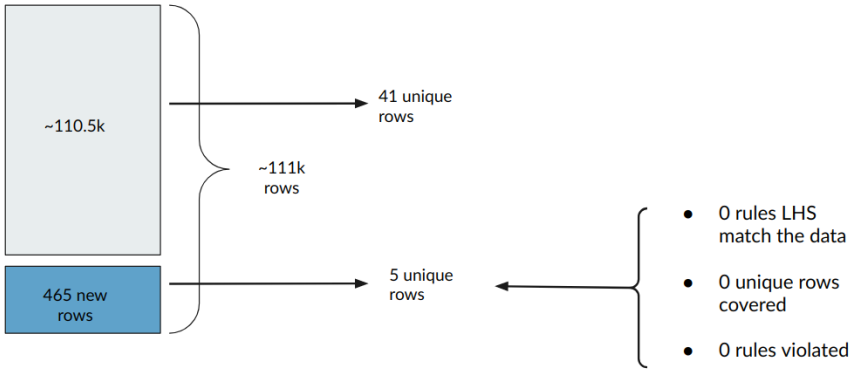

Figure 17. Modification 1 analysis

size	is_empty	peek_obj_type
3.0	false	class java.lang.short
4.0	false	class java.lang.double
5.0	false	class java.lang.byte
5.0	false	class java.lang.float
4.0	false	class java.lang.integer

Figure 18. Modification 1. Unique rows in the new rows

Modification 2: isEmpty()

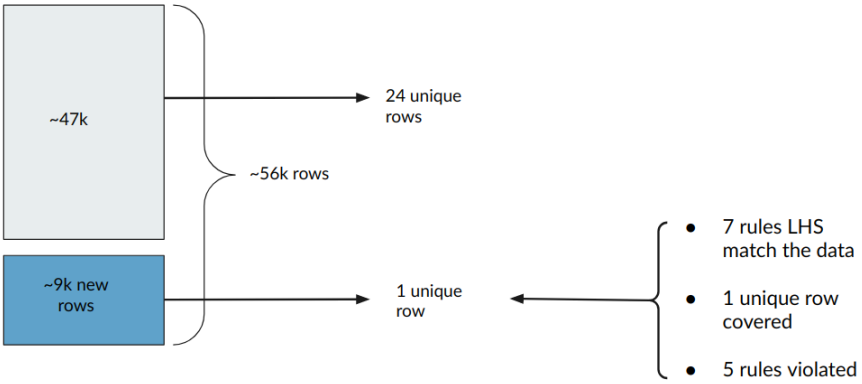

Figure 19. Modification 2 analysis

size	is_empty	peek_obj_type
2.0	true	emptyobject

Figure 20. Modification 2. Unique rows in the new rows

Modification 3: size()

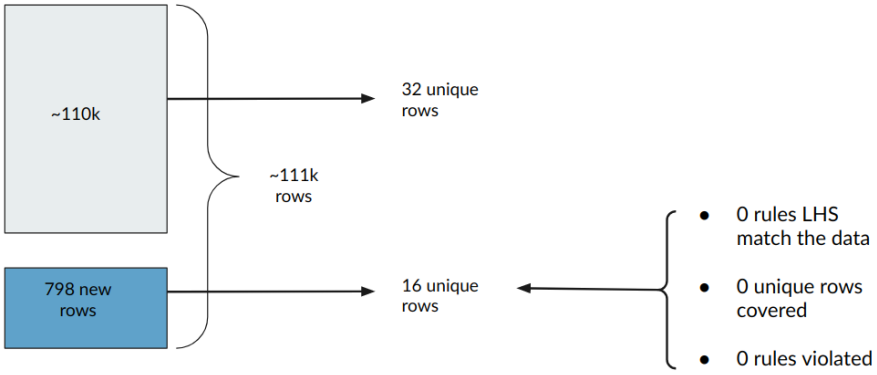

Figure 21. Modification 3 analysis

size	is_empty	peek_obj_type
3.0	false	class java.lang.short
4.0	false	class java.lang.double
5.0	false	class java.lang.float
4.0	false	class java.lang.string
4.0	false	class java.lang.integer
5.0	false	class java.lang.byte
2.0	false	class java.lang.object
4.0	false	class java.lang.object
4.0	false	class java.lang.character
2.0	false	class java.lang.character
7.0	false	class java.lang.long
5.0	false	class java.lang.long
7.0	false	class java.lang.class
6.0	false	class java.lang.boolean
7.0	false	class drivers.stacknewdriver
6.0	false	class java.lang.short

Figure 22. Modification 3. Unique rows in the new rows

Modification 4: peek() + size() + isEmpty()

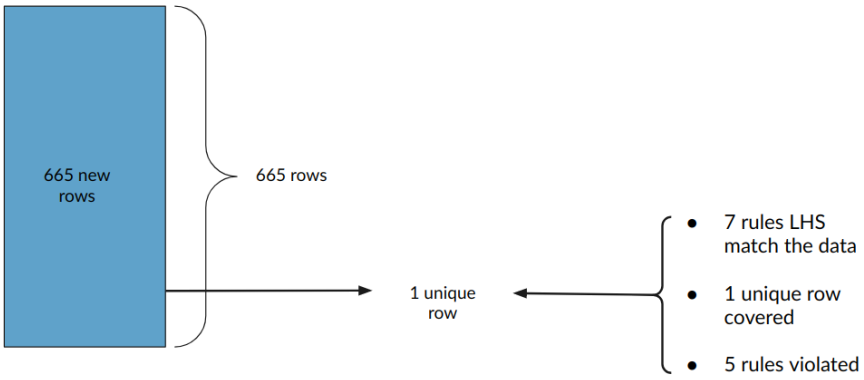

Figure 23. Modification 4 analysis

size	is_empty	peek_obj_type
0.0	true	class java.lang.string

Figure 24. Modification 4. Unique rows in the new rows

Modification 5: peek() + isEmpty()

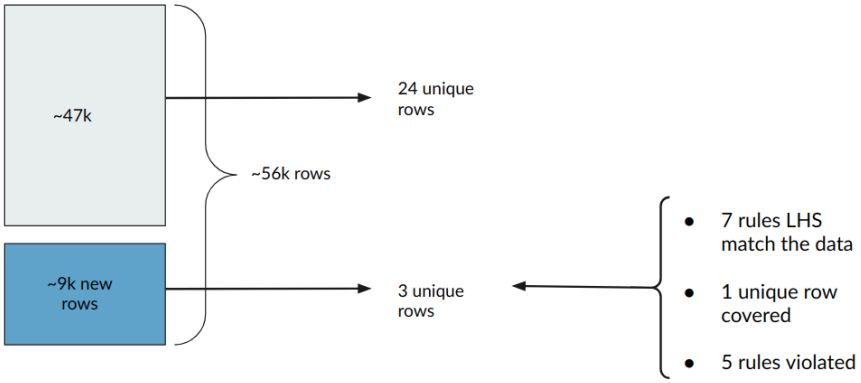

Figure 25. Modification 5 analysis

size	is_empty	peek_obj_type
2.0	true	emptyobject
3.0	false	class java.lang.short
4.0	false	class java.lang.double

Figure 26. Modification 5. Unique rows in the new rows

Modification 6: peek() + size()

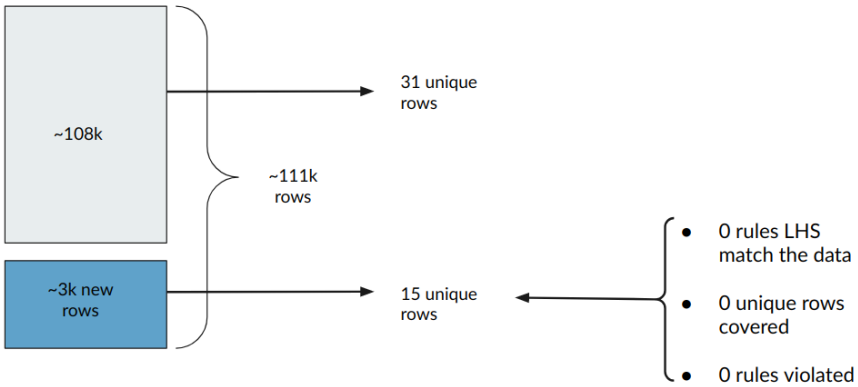

Figure 27. Modification 6 analysis

size	is_empty	peek_obj_type
3.0	false	class java.lang.short
4.0	false	class java.lang.double
5.0	false	class java.lang.float
5.0	false	class java.lang.byte
5.0	false	class java.lang.double
6.0	false	class java.lang.short
6.0	false	class java.lang.byte
2.0	false	class java.lang.object
2.0	false	class java.lang.character
6.0	false	class java.lang.float
7.0	false	class drivers.stacknewdriver
5.0	false	class java.lang.integer
4.0	false	class java.lang.integer
5.0	false	class java.lang.long
6.0	false	class java.lang.boolean

Figure 28. Modification 6. Unique rows in the new rows

Modification 7: isEmpty() + size()

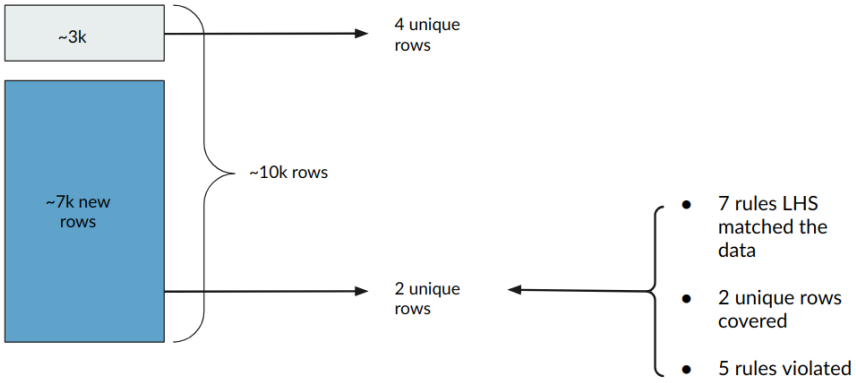

Figure 29. Modification 7 analysis

size	is_empty	peek_obj_type
2.0	true	emptyobject
4.0	false	class java.lang.double

Figure 30. Modification 7. Unique rows in the new rows

II. Code

All the project code is located in the following GitHub repository:
https://github.com/rramler/ste2019-rule-mining

An access to the repository could be granted by sending a email to:
Rudolf.Ramler@scch.at

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Anastasiia Shalygina**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Using Rule Mining for Automatic Test Oracle Generation,

   supervised by Dietmar Pfahl and Rudolf Ramler.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Anastasiia Shalygina
05/05/2020