# **Technical report:** Using Rule Mining for Automatic Test Oracle Generation

Alejandra Duque Torres, Dietmar Pfahl, and Rudolf Ramler

**Abstract:**

**Software testing is essential, but also one of the costliest and time-consuming activities in the software development process.** **However, software testing has historically been a task that has been recognised to be time-consuming and tedious as well as expensive, given the size and complexity of large-scale software systems. Such cost and time involved in testing can be managed through test automation. Software testing, automated or not, has four major steps: tests suite generation, predicting the outcomes of the tests, execute the SUT with the suite tests to obtain the actual outcome and compare the predicted outcome against the actual outcome to obtain a verdict (pass/fail). There are two significant challenges in the testing process: find successful test inputs, i.e., such inputs that can reveal faults in SUT, and determine what should be the output of a system after the execution of the test cases. The second challenge refers to one of the most significant problems in software testing automation, i.e., the test oracle problem. In this research, we developed a methodology to generate tests oracles using the state information of the System Under Test (SUT) during the execution of the tests. We derive our approach in the form of rules using the Association Rule Mining (ARM) technique. ARM attempt to find relationships or associations between categorical variables in large transactional data sets. Therefore, we wanted to investigate the potential of ARM to model SUT state. In particular, we were interested in understanding whether the information provided by the resulting model can help to verify the correct operation of the SUT new versions. We focus at a level of software testing called unit testing, which tests each unit or component of the SUT separately. Furthermore, this research uses the Stack Class of the Java Collection framework as SUT. The rule mining approach can detect that something is wrong; this is when a rule is violated. Then, by analysing the rules violated, it is possible to localise the fault. However, in terms of time-consuming, the rule mining approach takes much time, in particular when the number of rules is high, for instance. We provide an analysis of different performance metrics and discuss the results obtained.**

1. **INTRODUCTION:**

Software testing is an essential activity during the software development process as it helps ensure the correct operation of the final software [1]. It tries to expose the hidden defects of the system or software that is under test (from now on only SUT) through Test Scripts, that is, a set of instructions that are executed against the SUT to verify that the system works as expected [2]. However, software testing has historically been a task that has been recognised to be time-consuming and tedious as well as expensive, given the size and complexity of large-scale software systems [4]. Such cost and time involved in testing can be managed through test automation. Test automatisation refers to the writing of special programs that are aimed to detect defects in the SUT and to using these programs together with standard software solutions to control the execution of test suites. It is possible to use test automation to improve the process effectiveness, for example, by reducing the risk for human errors and make the tests more repeatable.

Software testing, automated or not, has four major steps: tests suite generation, predicting the outcomes of the tests, execute the SUT with the suite tests to obtain the actual outcome and compare the predicted outcome against the actual outcome to obtain a verdict (pass/fail). There are two significant challenges in the testing process: find successful test inputs, i.e., such inputs that can reveal faults in SUT, and determine what should be the output of a system after the execution of the test cases. The second 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. Although several pieces of research have been done to provide test oracle automatically, none of them could completely automate all test oracle activities in all circumstances.

Inspired and motivated by the limitations above, we developed a methodology to generate tests oracles using the state information of the SUT during the execution of the tests. We refer to the SUT state as the set of all the values of the attributes of an object. We assume that most programs have objects with a mutable state, and the execution of methods can modify the state of the program. The idea of using the state information lies back on the assumption that relation in the state when testing a new version of the SUT should remain unchanged or should not change signally. Thus, knowing these relations may help to verify new or modified versions of the software.

The methodology uses information related to the state of the SUT to construct a model that can identify interesting relations in the data. The methodology uses the Association Rule Mining (ARM) technique to construct the model. ARM is a rule-based technique that belongs to unsupervised machine learning (ML) methods [5]. The algorithms used in ARM attempt to find relationships or associations between categorical variables in large transactional data sets [6]. In particular, we were interested in understanding whether the information provided by the resulting model can help to verify the correct operation of the SUT new versions. More specifically, we wonder if we can detect and locate faults in new versions of the SUT. We tackle our goal by answering the following research questions:

* **RQ1: How well does the rule mining approach represent the behaviour of the SUT state?**
* **RQ2: How strong/strict the rule mining is?**
* **RQ3: What faults can the rule mining approach detect?**
* **RQ4: What information regarding faults detection and location can the methodology offer?**

In the context of our study, we focus at a level of software testing called unit testing, which tests each unit or component of the SUT separately. Furthermore, this research uses the Stack Class of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate. Then, the goal is to explore the state data from the Stack Class and apply ARM to build a model that can identify interesting relationships in the data in the form of rules. Knowing these relationships can help to verify new or changed versions of the software. The rest of the report is structured as follows. In Section II, we elaborate on the concept of ARM. We describe the proposed methodology and detail the different analysis performed in Section III. In Section IV, we present the answers to our research question. Finally, we discuss our results in Section V.

1. **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. ARM is a rule-based unsupervised ML method that allows discovering relations between variables or items in large databases. ARM has been highly used in other fields as business analysis [3], medical diagnosis [4], and census data to find out patterns they never knew existed. ARM process consists of at least two steps [5]: (i) finding all the frequent itemsets that satisfy minimum support thresholds and, (ii) generating strong association rules from the frequent derived itemsets by applying minimum confidence threshold. Below we define important terminology regarding ARM:

* ***Itemset****:* Let be a set of different items in the dataset . Itemenset is a set of items, which is a subset of . is an itemset with different items.
* ***Association rule****:* Consider a dataset , having number of transactions containing a set of items. An association rule exposes the relationship between the itemset with .
* ***Consistent rule:*** The set of association rules containing itemset, which is locally as well as globally frequent in large data.
* ***Inconsistent rule:***The set of association rules containing itemset, which is frequent locally but not frequent globally or wise a versa are the inconsistent rules. Inconsistent rules are non-conforming patterns in the dataset.
* ***Support****:* The support is the percentage of transaction in the dataset that contains that contain both itemsets and . The support of an association rule : s
* ***Confidence:*** The confidence is the percentage of transactions in the database with itemset that also contains the itemset Y. The confidence is calculated using the conditional probability which is further expressed in terms of itemset support:
* ***Lift/Interest:*** Lift/Interest is used to measure frequency and together if both are statistically independent of each other. The lift of rule is defines as . A lift value one indicates and appear as frequently together under the assumption of conditional independence. In this case and are said to be independent of each other.

1. **Methodology Proposed**

Overall, the automatic test oracle generation using ARM 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 state data from the first version of SUT is received, and the rule mining process starts. After that, one gets a set of rules using that 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. All the process can be split into two main phases.

Figure 1 presents a rule mining method for tests oracles generation based on the SUT state information. The methodology is made up of two main phases. Overall, Phase one is responsible for the ruleset generation, i.e., the rule mining part. Therefore, the result of phase one is the ruleset. The ruleset is the representation or modelling of SUT state behaviour. Phase two is in charge of applying the ruleset to the new SUT version. Therefore, the result of this phase could be seen as a fault report for new SUT versions. Both phase one and two consists of three steps, and each step have internal processes. Below we describe in detail each phase and its steps.

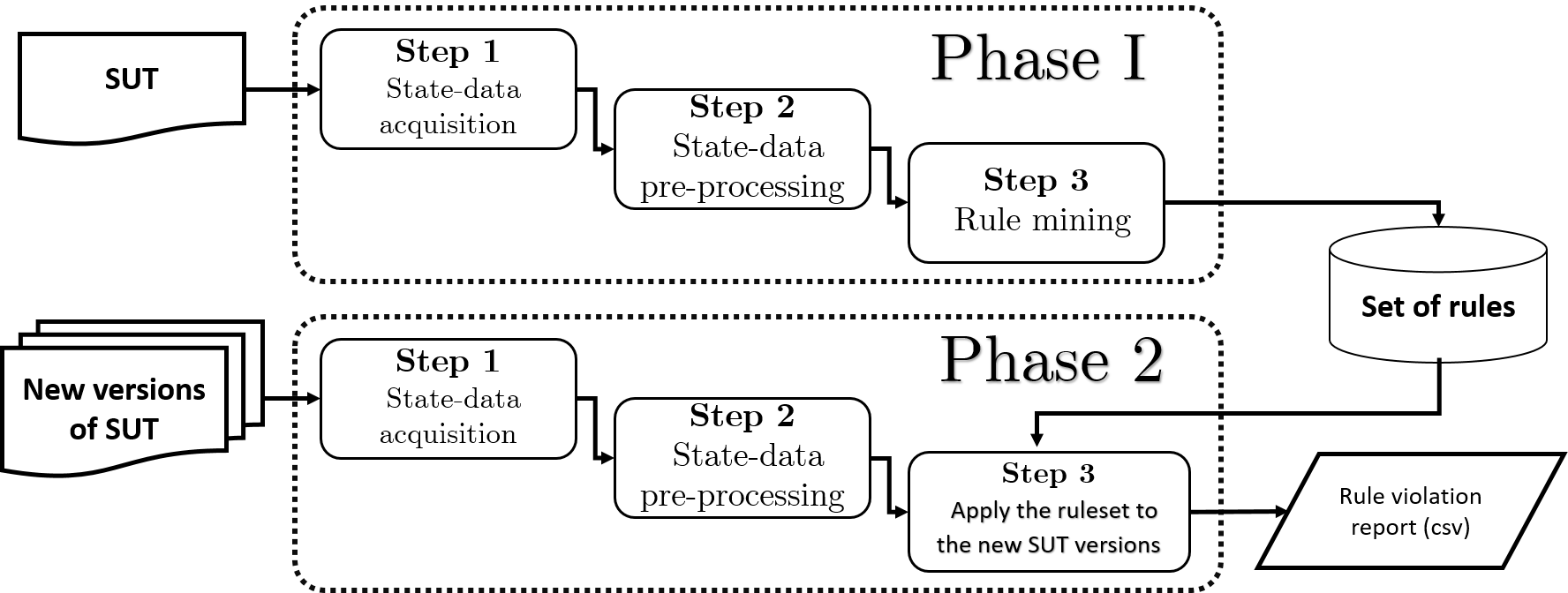


Figure 1 Overview of the method for rule-mining based tests oracles generation

* 1. **Phase I**
     1. **Step 1.1 – *State data acquisition***

Figure 2 provides the flowchart for extracting the SUT state data. Two activities are necessary to accomplish the data acquisition step. The first activity is responsible for the generation of tests, and the second activity is responsible for the execution of the tests against SUT, track the state, and save raw data with the SUT state information.

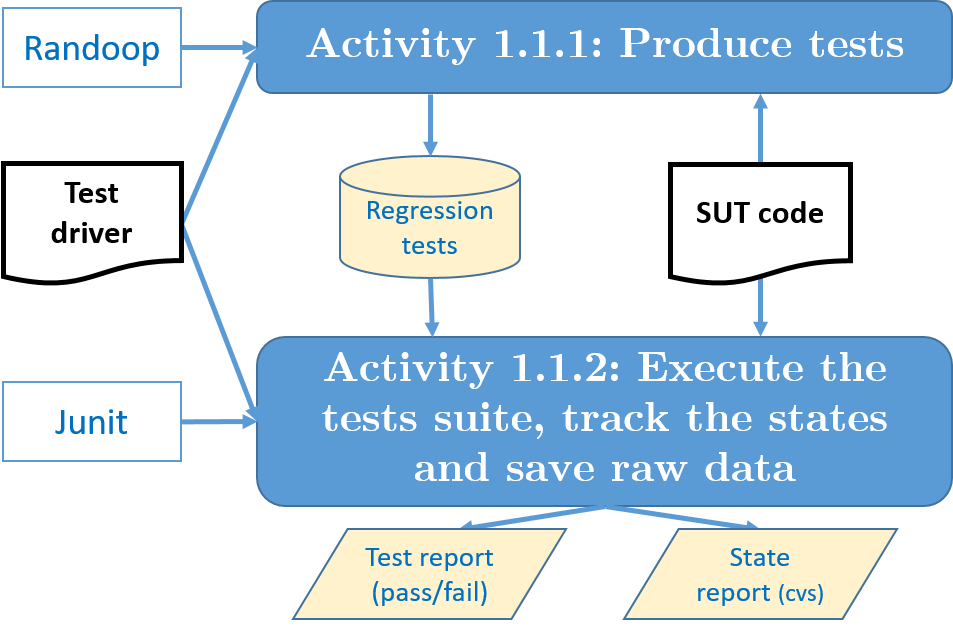


Figure 2 Flowchart for extracting the SUT state data

* **Activity 1.1.1, Produce tests:** One of the challenges in a data acquisition step is to create various test suites for the SUT. Unite--tests manual writing is a challenging, tiresome, and time-consuming activity. To avoid the unite—test manual writing, we used Randoop to generate unite--tests automatically, which is one of the most popular random JUnit tests generators for Java [6]. Randoop relies on feedback--directed random testing technique hat benefits from the feedback obtained after the execution of tests when they are created. It helps to avoid illegal and redundant inputs, e.g., 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. The second 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.

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 do not 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 [6].

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 test limit and a random-seed. 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 allow 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

* **Activity 1.1.2. Execute the test suite, track the states and save raw data:** To track the states of the SUT while running the test suite and save it to the text file for later analysis, we built a special program that helps to track and save the information of the state of the SUT. We call this program Test Driver. The main idea behind the Test Driver is that the methods of the SUT can be logically divided into two categories: the methods that change a state of the class instance of the SUT, and the methods that reflect the state or, in other words, give some information about the object instance. We call such methods as state methods. The test driver track and store the information returned by state methods if they are called immediately after the test case execution. Such information is saved in a CSV file.
  + 1. **Step 1.2 –** ***State data pre-processing***

Figure 3 provides the flowchart for pre-processing the state data. This data pre-processing step is made up of three activities:

* **Activity 1.2.1, Sorting, augmenting and cleaning:** This activity is responsible for ensuring that the data is correct, consistent and useable. Figure 4 shows the pseudo algorithm created for this task. The algorithm has three main functions: sort, aug, and clean. The sort function is responsible for sorting the dataset based on the TestId and InstanceId, this is done to find interesting sequences in the data, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. For example, it is possible to add characteristics that indicate the previous state. The clean function removes the rows that are no needed or inconsistent rows.
* **Activity 1.2.2, Encoding:** In this activity, the data is prepared according to the requirements of the rule mining algorithm. For example, Apriori [7], which is the algorithm used in this work, works only with categorical features. We categorise and generalise the numerical inputs into string representations.
* **Activity 1.2.3 Feature selection:** In data analysis Feature Selection is the process where automatically or manually are select those features which contribute most to the model. Having irrelevant features in the data can decrease the accuracy of the models. The feature selection is carried out to explore the different performance of the methodology when different features are used. In particular, this report, we use five different Datasets which contain different numbers of features. FS means Feature Selection, the number which is next to FS indicates the total number of features used.

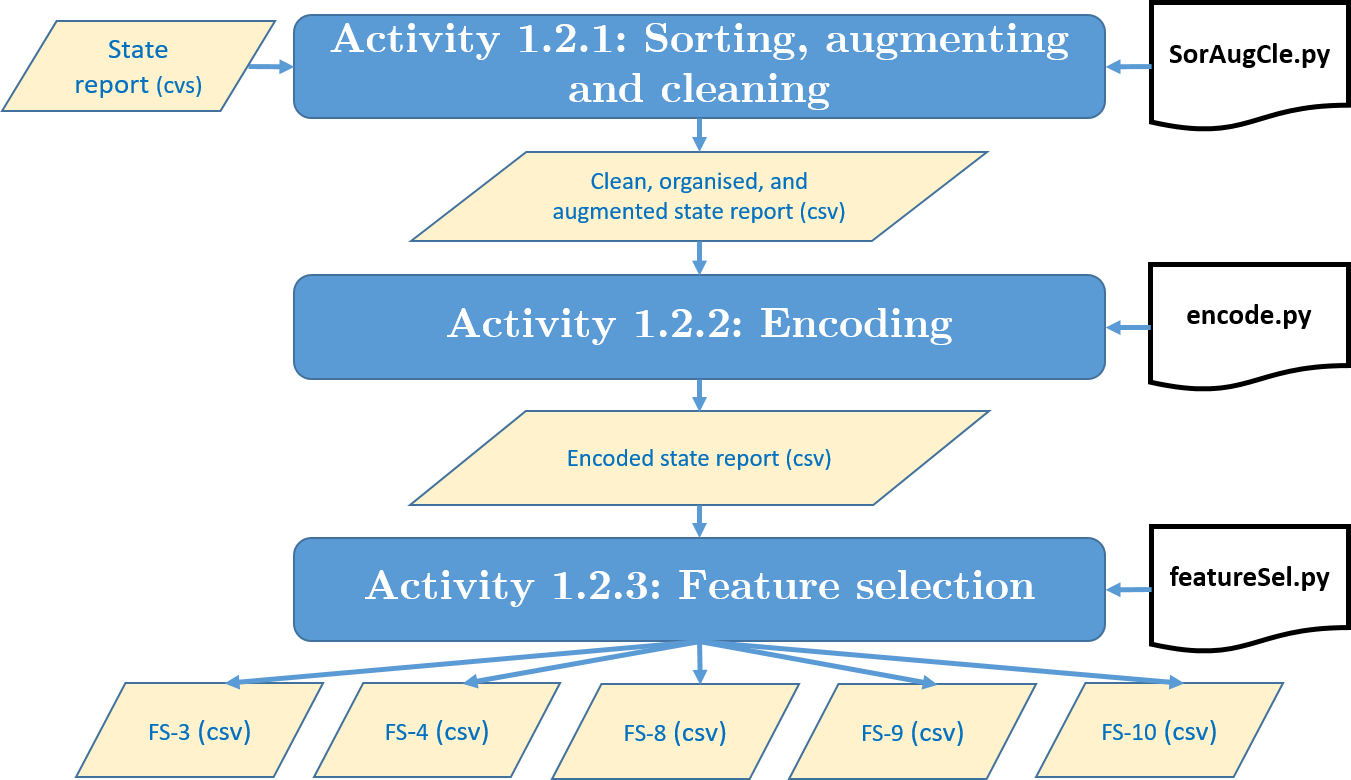


Figure Step 1.2 - State data pre-processing

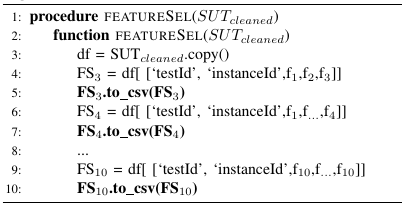
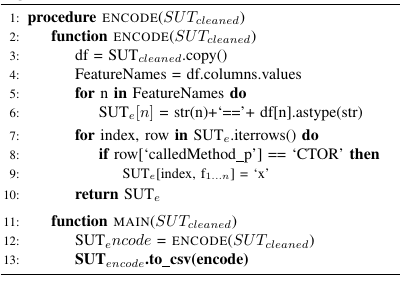
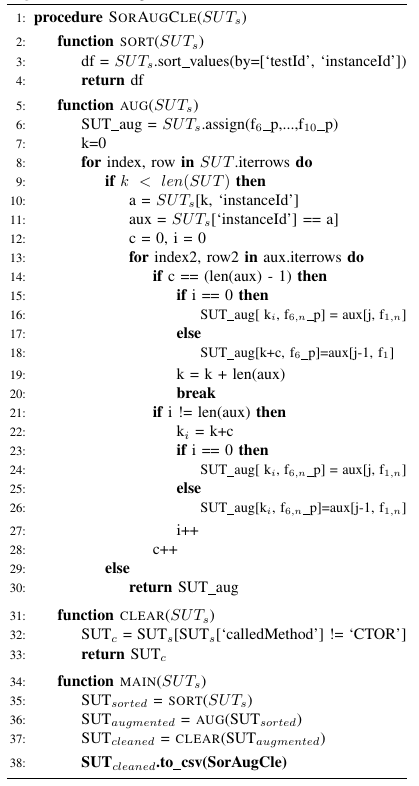


Figure 4 Algorithms SorAugCle, encode, and featureSel

* + 1. **Step 1.3 –** ***Rule generation***

Figure 5 provides the flowchart of rule generation step or rule mining. A large number of ARM algorithms have been reported in the literature. However, the best know mining algorithm is the Apriori algorithm; We decided to use it for our approach. Apriori has several implementations in different Python3 libraries and good documentation provided for some of these libraries. Thus, we decided to choose the most popular and well-known available algorithm for rule mining. We use Python3 for data analysis, rule mining, and rules application.

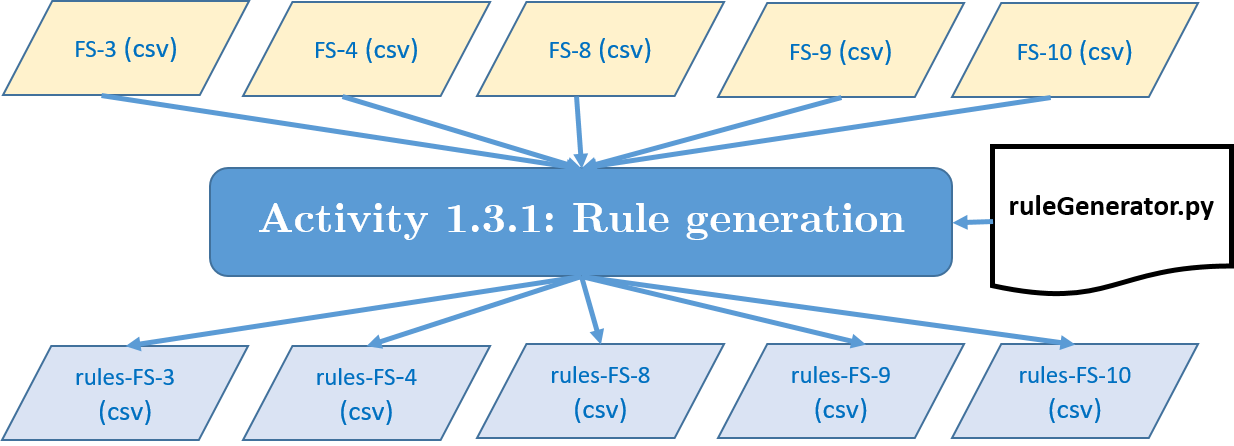


Figure 5 Rule generation process

We apply the Apriori algorithm with minimal *support* and maximum *confidence* thresholds, i.e., 0.2 and 1, to each dataset considering data rows as transactions. We apply the Apriori algorithm with minimal support and maximum confidence thresholds, this is 0.2 and 1 respectively, to each dataset considering data rows as transactions. The generated rules are of the form: . A rule can be interpreted as "if itemset X occurs in a transaction, then itemset Y will also likely occur in the same transaction". A rule will have two sides, In this example, X is a left-hand side (LHS) of the rule, and Y is a right-hand side (RHS). Each rule generated will have at least three metrics: Confidence, Support, and Lift. The support of the rule can not be lower than the support value configured as a threshold in the Apriori algorithm. The confidence of the rule can not be higher than the confidence value configured in the Apriori algorithm.

* 1. **Phase II**
     1. **Step 2.1 – *State data acquisition.*** Figure 6 provides the flow chart for extracting the status data from the new version of the SUT. Unlike step one of phase one, in this step, only one activity is necessary; this is the test execution activity. In phase one, we assume that the first version of the SUT is correct; then, we build test suites using Randoop and the test driver. The same tests generated are used to test the new versions of the SUT in step one of phase two.

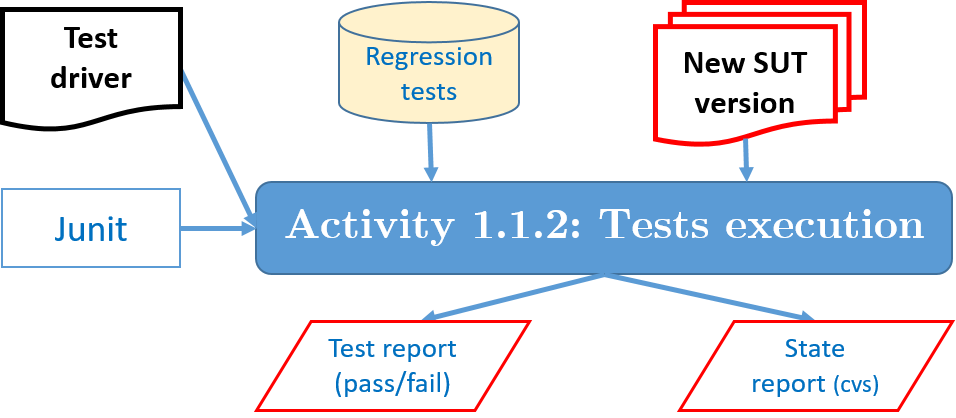


Figure Step 2.1 State data acquisition

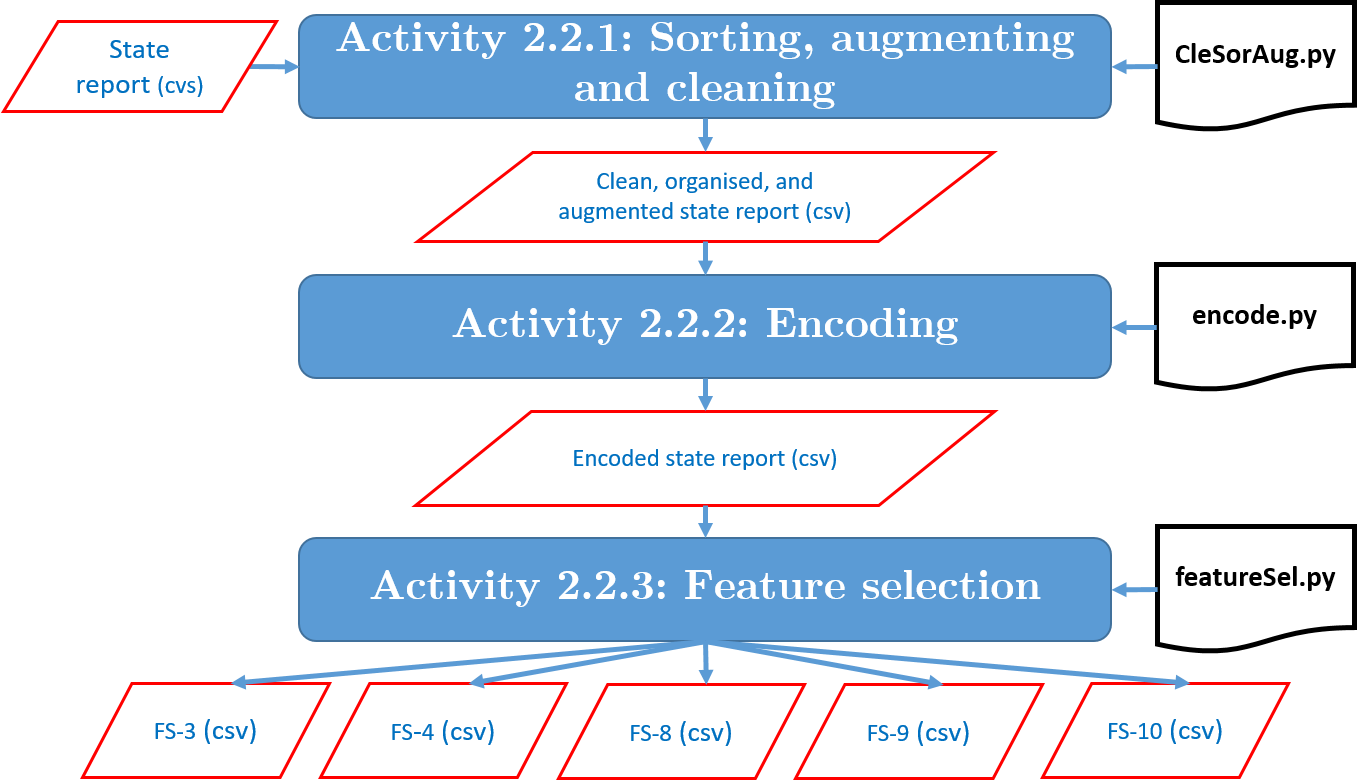
* + 1. **Step 2.2 –** ***State data pre-processing.*** As in step two of phase one, step two of phase two performs the same function as in phase one as Figure 7 shows. This is, the State Report data is prepared according to the requirements of the rule validation algorithm. Also, in this pre-processing step, the feature selection is carried out on the modified versions.In this step are used the same algorithms than in step 1.2.

Figure Step 2.2 - State data pre-processing

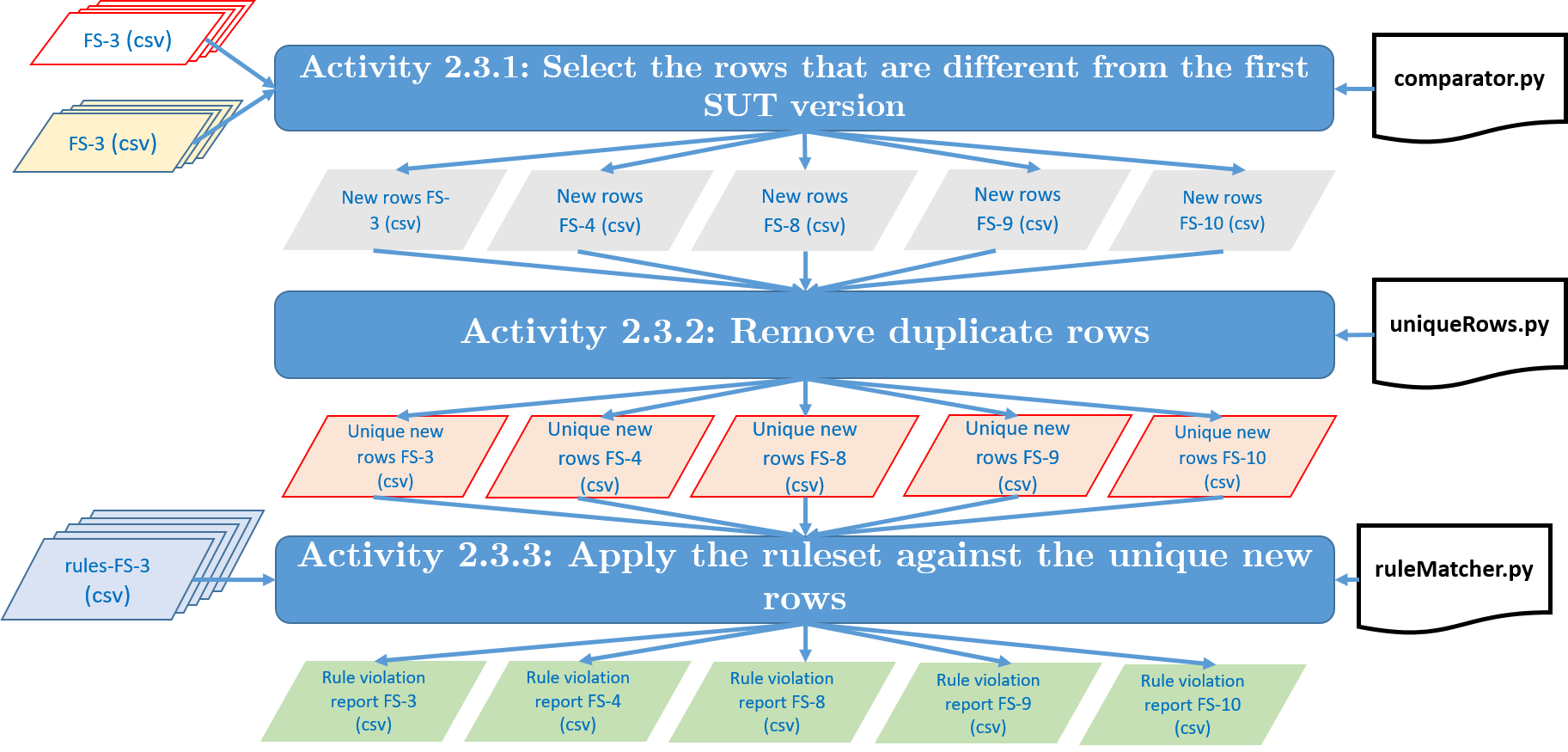
* + 1. **Step 2.3 –** ***Apply the ruleset to the new SUT versions.*** Figure 8 provides the flowchart for applying the ruleset to the new SUT versions. This data pre-processing step is made up of three activities:

Figure 8 Step 2.3 Apply the ruleset to the new SUT versions

* **Activity 2.2.1, Select the rows that are different from the first SUT version:** This activity is in charge of comparing and selecting the rows that are exclusively different from the first version of the SUT. This activity uses the comparator.py algorithm, Figure 9 Algorithm comparator. Although the comparison between two files seems simple, in our context, it is different. The test driver helps to track and store the information returned by state methods if they are called immediately after the test case execution. However, if the new version of SUT fails the regression test, the test will end its process, and the test driver will save the state information at that time. Then the test execution will continue with the next test suite until it is complete the entire test suite. Some regression tests may fail for some modified versions of the SUT, then, the CSV file generated by the test driver for those modified versions will have a different size than the no modified version.

The Comparator compares the sizes; if they are equal, it compares the two files row by row. If they are of different sizes, the Comparator creates subsets based on the Instance Id. Then compare row by row of the sub-sets. In both cases, same size or different size, we labelled with "Yes" or "Not" the rows in a new column named "diff". "Yes" is the row is different, i.e., it is a new column, and "Not" if the row is the same.

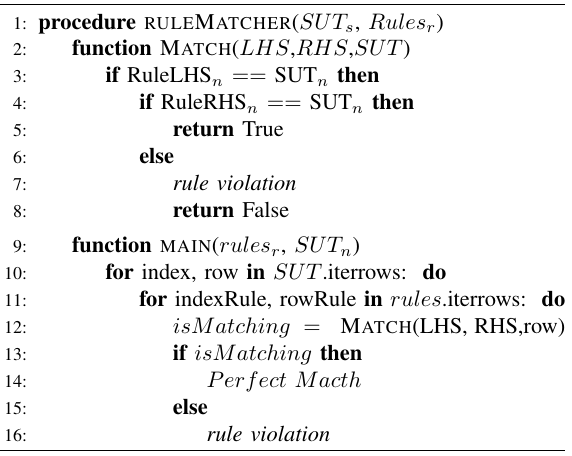
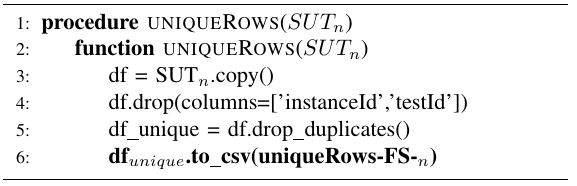
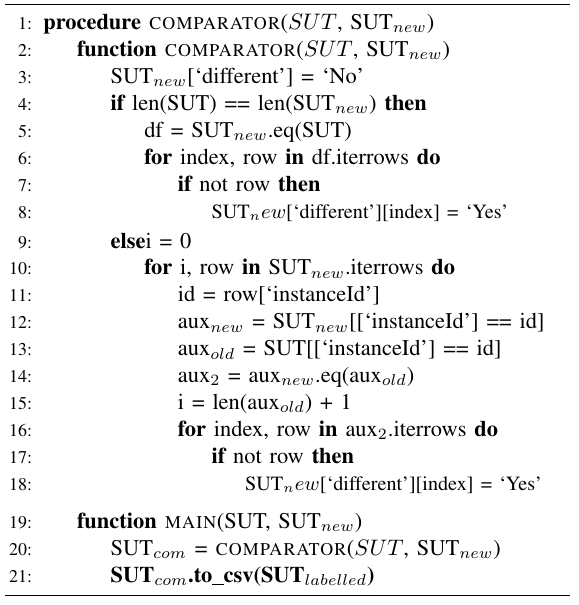
* **Activity 2.2.2, Remove duplicate rows:** In this activity, we remove the rows that are duplicate. We eliminate the rows that are duplicates, for optimisation reasons. If two or more rows are the same, then they will have the same results.
* **Activity 1.2.3 Apply the ruleset against the new unique rows:** This activity is responsible for applying the ruleset against the new unique rows, the algorithm in charge of this activity is the ruleMatcher.py. Overall, the algorithm first takes a rule from the set of rules. Then, it takes LHS of the rule and selects the records that match the LHS from the dataset we want to validate. Then, it checks 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 summarised 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 don't match
      6. Repeat steps 1-5 for every rule

In the end, we will know whether the tested 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 states that correspond to the valid version of SUT and every record in his 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 correct version records, and some may not. Thus, it makes sense to validate only the records from the new 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 dataset from the not modified version. It makes the validation process faster.

Figure 9 Algorithms Comparator, uniqueRows, and ruleMatcher



1. **Results**

In the context of our study, we focus at a level of software testing called unit testing, which tests each unit or component of the SUT separately. Furthermore, this research uses the Stack Class of the Java Collection framework as SUT. This class was chosen as its state behaviour is well known and easy to manipulate. Then, the goal is to explore the state data from the Stack Class and apply ARM to build a model that can identify interesting relationships in the data in the form of rules. In this section, we provide the expected output of each phase and their steps. Also, we answer the research questions.

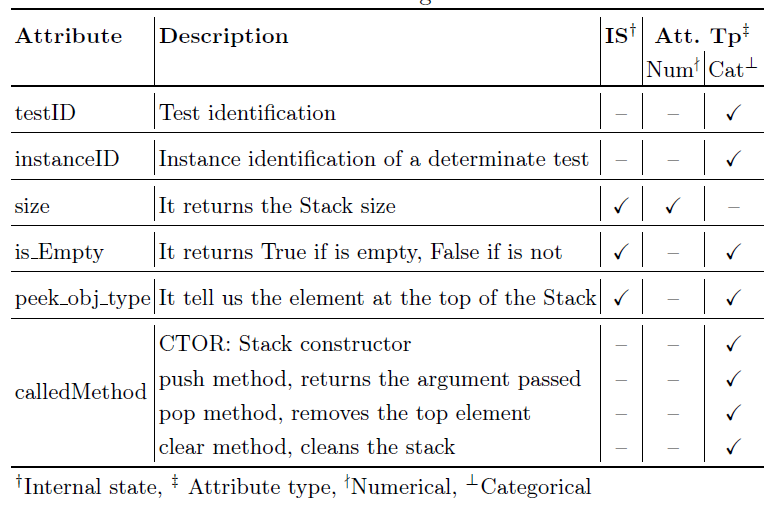
* 1. **Phase 1**
     1. ***Step 1.1. State data acquisition***. The first activity is responsible for the generation of tests, and the second activity is responsible for the execution of the tests against SUT, track the state, and save raw data with the SUT state information. Two different reports are the output of this step, Test Report (pass / failed) and State Report. The regression testing generates the test report (pass/fail). The Test Driver generates the State Report. In the Stack class, the state data is characterised or modelling by the 'getters methods', i.e., peek(), isEmpty(), size() and, method Called. State Report is a CSV file with the values of peek(), isEmpty(), size() and, method Called, during the execution of the test. Table 1 summarises the attributes/features extracted. The attribute "*testID*" and "*instanceID*" provide an identification of the test generate by Randoop. One test can have multiples and ID instances. The attribute "*size*", all the values are numeric, "*is*\_*Empty*" attribute is categorical and contains only "*True*" or "*False*" records. We use a type of the object "*peek*\_*obj*\_*type*". Since Stack 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.

Table 1 Attributes/features extracted by the Test Driver

* + 1. **Step 1.2 –** ***State data pre-processing.*** This data pre-processing step is made up of three activities:Activity1.2.1 *Sorting, augmenting and clean.* The sort function is responsible for sorting the dataset based on the TestId and InstanceId, this is done to find the sequence of sizes of the Stack, and be able to model those sequences. When the dataset is ordered, it is possible to add more information. For example, it is possible to add characteristics that indicate the previous state. Then instead five features" *size*", "*isEmpty*", "*peek\_obj\_type*", and" *calledMethod*", "pushInput", we can get fice more. To distinguish from the first five we add a "\_p" ar the end of the name, this indicates "previous”, i.e., "*size*\_p", "*isEmpty\_p*", "peek\_obj\_type\_p*"*, "*calledMethod\_p*", and "*pushInput\_p*". The cleanup function removes the rows that have the Constructor since this is not state information. The Constructor only indicates that a new stack was created. The Activity 1.2.2 Encoding, we encode the feature "*size*", and "*size*\_p" since are no categorical features.

In the activity 1.2.3 Feature selection, we create five different Datasets which contain different numbers of features. The created data sets are named with the prefix FS, which stands for Feature Selection. To distinguish the different data sets, they have been named with the number of characteristics that were used, i.e., FS-X where X is the number of characteristics

* **FS-3:** In this feature selection, FS-3, It is used the features "*size*", "*isEmpty*", and *"peek\_obj\_type"* only.
* **FS-4:** In this feature selection, FS-4, It is used the features "*size*", "*isEmpty*", and *"peek\_obj\_type"* and "*called\_Method*".
* **FS-8:** In this feature selection, FS-8, we used the features" *size*", "*isEmpty*", "*peek\_obj\_type*", and" *calledMethod*", and their previous values, which are distinguished by adding **"***\_p***"** at the end of the feature name, i.e., "*size*\_p", "*isEmpty\_p*", "peek\_obj\_type\_p*"*, and "*calledMethod\_p*".
* **FS-9:** In addition to the previous features used in FS-8, ”*size*”, “*isEmpty*”, “*peek\_obj\_type*”, ”*calledMethod*”, “*size*\_p”, “*isEmpty\_p*”, “peek\_obj\_type\_p*”*, and “*calledMethod\_p*”. The feature "pushIputh" is used in FS-9.
* **FS-10:** In this feature selection FS-10, we use all the feature and their previous values, i.e., ”*size*”, “*isEmpty*”, “*peek\_obj\_type*”, ”*calledMethod*”, “pushInput”, “*size*\_p”, “*isEmpty\_p*”, “peek\_obj\_type\_p*”*, “*calledMethod\_p*”, and “*pushInput\_p*”
  + 1. **Step 1.3 –** ***Rule generation,*** Table 2 describes the rules generated for FS-3 dataset. The total number of rules generated are 14; the confidence value for all of them is 1. The average of the support and Lift measurements are 0.381 and 2.391, respectively.

Table Set of Rules generated using size, isEmpty, and peek\_obj\_type

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FS–3** | | | | | | | | | |
| **LHS items** | **LHS** | **RHS items** | **RHS** | **Support** | | | **Lift** | | |
| **max** | **mean** | **min** | **max** | **mean** | **min** |
| 1 | 11 | 1 | 8 | 0.408 | 0.375 | 0.227 | 2.539 | 2.317 | 1.65 |
| 2 | 3 | 0.394 | 0.394 | 0.394 | 2.539 | 2.539 | 2.539 |
| 2 | 3 | 1 | 3 | 0.408 | 0.375 | 0.227 | 2.539 | 2.317 | 1.65 |
|  | | | | | | | | | |
| **Total number of rules** | | | 14 | 0.403 | 0.381 | 0.283 | 2.539 | 2.391 | 1.946 |

Table 3 describes the rules generated by the Rule Mining module. The total number of rules generated are 36; the confidence value for all of them is 1. The average of the support and Lift measurements are 0.273 and 2.475, respectively. The total number of attributes used are 4; thus, the maximum number of possible relations or rule generates is 3:1 (this relation refers to the size of the LHS and RHS).

Table Set of Rules generated using size, isEmpty, peek\_obj\_type and called\_Method as feature

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FS–4** | | | | | | | | | |
| **LHS items** | **LHS** | **RHS items** | **RHS** | **Support** | | | **Lift** | | |
| **max** | **mean** | **min** | **max** | **mean** | **min** |
| 1 | 19 | 1 | 12 | 0.549 | 0.353 | 0.227 | 2.539 | 2.317 | 1.65 |
| 2 | 6 | 0.394 | 0.311 | 0.229 | 2.539 | 2.539 | 2.539 |
| 3 | 1 | 0.228 | 0.228 | 0.228 | 2.539 | 2.539 | 2.539 |
| 2 | 14 | 1 | 11 | 0.394 | 0.285 | 0.207 | 2.539 | 2.377 | 1.65 |
| 2 | 3 | 0.229 | 0.229 | 0.229 | 2.539 | 2.539 | 2.539 |
| 3 | 3 | 1 | 3 | 0.229 | 0.229 | 0.229 | 2.539 | 2.539 | 2.539 |
|  | | | | | | | | | |
| **Total number of rules** | | | 36 | 0.337 | 0.273 | 0.225 | 2.539 | 2.475 | 2.243 |

Table 4 describes the rules generated. The total number of rules generated are 439; the confidence value for all of them is 1. The average of the support and Lift measurements are 0. 227 and 3.808, respectively. The total number of attributes used are 8; thus, the maximum number of possible relations or rule generates is 7:1 (this relation refers to the size of the LHS and RHS). However, one can see the maximum size of the set of rules generated is 5.

Table 4 Set of Rules generated using size, isEmpty, peek\_obj\_type and called\_Method, "size\_p", "isEmpty\_p”, "peek\_obj\_type\_p", "calledMethod\_p" as feature

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FS-8** | | | | | | | | | |
| **LHS items** | **LHS** | **RHS items** | **RHS** | **Support** | | | **Lift** | | |
| **max** | **mean** | **min** | **max** | **mean** | **min** |
| 1 | 37 | 1 | 23 | 0.549 | 0.331 | 0.205 | 2.95 | 2.556 | 1.65 |
| 2 | 12 | 0.394 | 0.292 | 0.205 | 2.95 | 2.745 | 2.539 |
| 3 | 2 | 0.229 | 0.217 | 0.205 | 2.95 | 2.745 | 2.539 |
| 2 | 194 | 1 | 74 | 0.394 | 0.225 | 0.2 | 2.95 | 2.501 | 1.65 |
| 2 | 72 | 0.229 | 0.208 | 0.2 | 4.81 | 3.894 | 1.821 |
| 3 | 39 | 0.208 | 0.207 | 0.2 | 4.81 | 4.799 | 4.671 |
| 4 | 9 | 0.208 | 0.208 | 0.208 | 4.81 | 4.81 | 4.81 |
| 3 | 153 | 1 | 75 | 0.229 | 0.208 | 0.2 | 2.95 | 2.487 | 1.65 |
| 2 | 60 | 0.208 | 0.207 | 0.2 | 4.81 | 4.038 | 1.821 |
| 3 | 18 | 0.208 | 0.208 | 0.208 | 4.81 | 4.81 | 4.81 |
| 4 | 49 | 1 | 34 | 0.208 | 0.207 | 0.2 | 2.95 | 2.461 | 1.65 |
| 2 | 15 | 0.228 | 0.228 | 0.228 | 4.81 | 3.942 | 1.821 |
| 5 | 6 | 1 | 6 | 0.208 | 0.208 | 0.208 | 2.95 | 2.462 | 1.65 |
|  | | | | | | | | | |
| **Total number of rules** | | | 439 | 0.269 | 0.227 | 0.205 | 3.808 | 3.404 | 2.545 |

Table 5 describes the rules generated. The total number of rules generated are 676; the confidence value for all of them is 1. The average of the support and Lift measurements are 0. 227 and 3.808, respectively. The total number of attributes used are 8. Thus, the maximum number of possible relations or rule generates is 7:1 (this relation refers to the size of the LHS and RHS). However, one can see the maximum size of the set of rules generated is 5

Table 5 Set of Rules generated using size, isEmpty, peek\_obj\_type and called\_Method, size\_p, isEmpty\_p, peek\_obj\_type\_p, calledMethod\_p, and inputpush as feature

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FS-9** | | | | | | | | | |
| **LHS items** | **LHS** | **RHS items** | **RHS** | **Support** | | | **Lift** | | |
| **max** | **mean** | **min** | **max** | **mean** | **min** |
| 1 | 37 | 1 | 31 | 0.549 | 0.318 | 0.205 | 4.405 | 2.508 | 1.65 |
| 2 | 24 | 0.394 | 0.299 | 0.205 | 4.836 | 2.785 | 1.821 |
| 3 | 9 | 0.394 | 0.279 | 0.205 | 4.836 | 2.84 | 2.539 |
| 4 | 1 | 0.229 | 0.217 | 0.205 | 2.95 | 2.745 | 2.539 |
| 2 | 279 | 1 | 105 | 0.394 | 0.237 | 0.2 | 4.836 | 2.493 | 1.65 |
| 2 | 107 | 0.394 | 0.22 | 0.2 | 4.836 | 3.548 | 1.821 |
| 3 | 55 | 0.229 | 0.208 | 0.2 | 4.81 | 4.374 | 2.539 |
| 4 | 12 | 0.208 | 0.206 | 0.2 | 4.81 | 4.775 | 4.671 |
| 3 | 238 | 1 | 118 | 0.394 | 0.215 | 0.2 | 4.836 | 2.478 | 1.65 |
| 2 | 93 | 0.229 | 0.207 | 0.2 | 4.81 | 3.681 | 1.821 |
| 3 | 27 | 0.208 | 0.205 | 0.2 | 4.81 | 4.527 | 2.539 |
| 4 | 83 | 1 | 58 | 0.229 | 0.207 | 0.2 | 2.95 | 2.443 | 1.65 |
| 2 | 25 | 0.208 | 0.205 | 0.2 | 4.81 | 3.682 | 1.821 |
| 5 | 11 | 1 | 11 | 0.208 | 0.204 | 0.2 | 2.95 | 2.438 | 1.65 |
|  | | | | | | | | | |
| **Total number of rules** | | | 676 | 0.305 | 0.231 | 0.201 | 4.392 | 3.237 | 2.169 |

Table 6 describes the rules generated. The total number of rules generated are 1450; the confidence value for all of them is 1. The average of the support and Lift measurements are 0. 225 and 3.492, respectively.

Table 6 Set of Rules generated using size, isEmpty, peek\_obj\_type and called\_Method, inputpush, size\_p, isEmpty\_p, peek\_obj\_type\_p, calledMethod\_p, inputpush\_p as feature

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **FS-10** | | | | | | | | | | |
| **LHS items** | **LHS** | **RHS items** | **RHS** | **Support** | | | **Lift** | | | |
| **max** | **mean** | **min** | **max** | **mean** | **min** |
| 1 | 85 | 1 | 35 | 0.549 | 0.317 | 0.205 | 4.405 | 2.515 | 1.65 |
| 2 | 33 | 0.394 | 0.298 | 0.205 | 4.836 | 2.83 | 1.821 |
| 3 | 15 | 0.394 | 0.276 | 0.205 | 4.836 | 2.884 | 2.539 |
| 4 | 2 | 0.229 | 0.217 | 0.205 | 2.95 | 2.745 | 2.539 |
| 2 | 459 | 1 | 131 | 0.394 | 0.241 | 0.2 | 4.836 | 2.521 | 1.65 |
| 2 | 158 | 0.394 | 0.221 | 0.2 | 4.836 | 3.513 | 1.821 |
| 3 | 113 | 0.229 | 0.208 | 0.2 | 4.81 | 4.365 | 2.539 |
| 4 | 48 | 0.208 | 0.207 | 0.2 | 4.81 | 4.801 | 4.671 |
| 5 | 9 | 0.208 | 0.208 | 0.208 | 4.81 | 4.81 | 4.81 |
| 3 | 551 | 1 | 189 | 0.394 | 0.215 | 0.2 | 4.836 | 2.512 | 1.65 |
| 2 | 213 | 0.229 | 0.208 | 0.2 | 4.81 | 3.76 | 1.821 |
| 3 | 121 | 0.208 | 0.207 | 0.2 | 4.81 | 4.575 | 2.529 |
| 4 | 28 | 0.208 | 0.208 | 0.208 | 4.81 | 4.81 | 4.81 |
| 4 | 280 | 1 | 134 | 0.229 | 0.208 | 0.2 | 2.95 | 2.489 | 1.65 |
| 2 | 112 | 0.208 | 0.207 | 0.2 | 4.81 | 3.848 | 1.821 |
| 3 | 34 | 0.208 | 0.208 | 0.208 | 4.81 | 4.57 | 2.95 |
| 5 | 68 | 1 | 47 | 0.208 | 0.207 | 0.2 | 2.95 | 2.471 | 1.65 |
| 2 | 21 | 0.208 | 0.208 | 0.208 | 4.81 | 3.857 | 1.821 |
| 6 | 7 | 1 | 7 | 0.208 | 0.208 | 0.208 | 2.95 | 2.478 | 1.65 |
|  | | | | | | | | | | |
| **Total number of rules** | | | 1450 | 0.279 | 0.225 | 0.203 | 4.404 | 3.492 | 2.442 |

Table 7 provides a comparison between the number of rules, support values and lift values for each FS. The different measurements of significance around support and confidence can be observed based on these threshold values.

Table 7 Support and Lift Ratio summarised per features selected

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **FS** | **Number of rules** | **support** | | | **lift** | | |
| **Max** | **mean** | **min** | **max** | **mean** | **min** |
| **FS-3** | 14 | 0.403 | 0.381 | 0.283 | 2.539 | 2.391 | 1.946 |
| **FS-4** | 36 | 0.337 | 0.273 | 0.225 | 2.539 | 2.475 | 2.243 |
| **FS-8** | 439 | 0.269 | 0.227 | 0.205 | 3.808 | 3.404 | 2.545 |
| **FS-9** | 676 | 0.305 | 0.231 | 0.201 | 4.392 | 3.237 | 2.169 |
| **FS-10** | 1450 | 0.279 | 0.225 | 0.203 | 4.404 | 3.492 | 2.442 |

Figure Max, Mean, and Min values of Support ratio per features selected

As per Figure 10 and Table 7, we can observe that the average support ratio decrease when the number of features used is increased. The average is closer to the threshold value set up. Table 10 also shows the number of rules that can be generated does not have linear behaviour since it depends on the number of items belonging to each feature.

Figure Max, Mean, and Min values of lift ratio per features selected

From Table 7 column lift and Figure 11, we can observe that the lift ratio is increasing when the number of features used is increased. This is the opposite of the support ratio. A lower lift ratio means that the probability of occurrence of a determinate rule is weak since the LHS and RHS are near to be independent between themselves. Then, a good indicator of a strict rule is its lift measure higher than the support threshold set up.

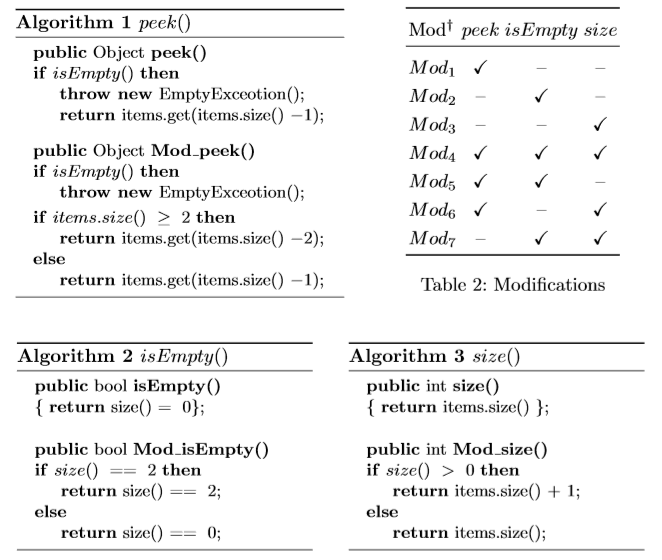
* 1. **Phase II**
     1. **Step 2.1 – *State data acquisition,*** In this step, we created new versions of the Stack class. Figure 12 shows the seven modifications made to the SUT. These modifications are done on purpose and manually, as we want to understand the potential of ARM to detect and locate faults using the status information from the SUT.

Figure 12 Modifications performed to the SUT

Table 8, 9 and 10 summarises information about the attributes extracted by the Test Driver and which are stored in CSV file for the first version of the SUT and the mutated ones. Following we provide the meaning of the terminology used to refers to the different version of the SUT

* **Not\_Mod:** First version of SUT
* **Mod1\_p:** Modification 1 (peek)
* **Mod2\_e:** Modification 2 (isEmpty)
* **Mod3\_s:** Modification 3 (size)
* **Mod4\_pes:** Modification 4 (peek + isEmpty + size)
* **Mod5\_pe:** Modification 5 (peek + isEmpty)
* **Mod6\_ps:** Modification 6 (peek + size)
* **Mod7\_es:** Modification 7 (isEmpty + size)

Table 8 Summary of attribute testId and InstanceId

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DS** | **# of Total rows** | **testId-Max** | **testId-Min** | **testId-Avg** | **testId-Unique** | **instanceId-Max** | **instanceId-Min** | **instanceId-Avg** | **instanceId-unique** |
| No\_mod | 86589 | 76 | 2 | 43 | 2037 | 17 | 1 | 6 | 14985 |
| Mod1-p | 86589 | 76 | 2 | 43 | 2037 | 17 | 1 | 6 | 14985 |
| Mod2-e | 46356 | 74 | 2 | 23 | 2037 | 15 | 2 | 5 | 8570 |
| Mod3-s | 86589 | 76 | 2 | 43 | 2037 | 17 | 1 | 6 | 14985 |
| Mod4-pes | 10385 | 17 | 2 | 5 | 2037 | 10 | 1 | 3 | 3225 |
| Mod5-pe | 46356 | 74 | 2 | 23 | 2037 | 15 | 2 | 5 | 8570 |
| Mod6-ps | 86589 | 76 | 2 | 43 | 2037 | 17 | 1 | 6 | 14985 |
| Mod7-es | 10385 | 17 | 2 | 5 | 2037 | 10 | 1 | 3 | 3225 |

In Table 8, the column named "# of Total rows" refers to the total of rows generated, the "*No\_mod*" dataset is the one who does not have mutants, and it is completely good, then, no tests are failed. The column "TestsId" provides information about the size of the tests generated by Randoop. In this sense, the columns "testId-Max", "testId-min", and "testId-avg" give the maximum, the minimum and the average size of the tests created by Randoop, and the sub-column Total provides the number of the different tests generated. The column named "InstanceId" refers to the number of new Stack created within the same test. The sub-columns "instanceId-Max", "instanceId-Min", and "instanceId-avg" give the maximum, the minimum size of the test and the minimum size of the test executions per test, and the sub-column Total provides the number of the total of new stacks created.

Table 9 Summary of attributes Size, IsEmpty, and peek\_obj\_type

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DS** | **# of Total rows** | **size\_max** | **size\_min** | **isEmpty\_True** | **isEmpty\_False** | **peek\_obj\_type-unique** | **peek\_obj\_type-Most frequent** | **peek\_obj\_type-Freq** |
| No\_mod | 86589 | 6 | 0 | 43184 | 43405 | 13 | EmptyObject | 43184 |
| Mod1-p | 86589 | 6 | 0 | 43184 | 43405 | 13 | EmptyObject | 43184 |
| Mod2-e | 46356 | 5 | 0 | 30086 | 16270 | 12 | EmptyObject | 23696 |
| Mod3-s | 86589 | 7 | 0 | 43184 | 43405 | 13 | EmptyObject | 43184 |
| Mod4-pes | 10385 | 4 | 0 | 9974 | 411 | 3 | class java.lang.String | 5086 |
| Mod5-pe | 46356 | 5 | 0 | 30086 | 16270 | 12 | EmptyObject | 23696 |
| Mod6-ps | 86589 | 7 | 0 | 43184 | 43405 | 13 | EmptyObject | 43184 |
| Mod7-es | 10385 | 4 | 0 | 9974 | 411 | 6 | class java.lang.String | 5086 |

In Table 9, the column "size\_max", and "size\_min" refer to the size maximum and minimum size of the Stack. As we can see from Table 9, the Maximum size achieved by the No\_mod version is 6, Since Stack 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. For modification 3 and 6 which are related to the size, the maximum is 7. However, this is because the method adds up one to the size when the size is more than 1. The columns "isEmpty\_true" and "isEmpty\_false" provide the number of rows that contain True or False, respective. In the No\_mod dataset, these two columns are no too much different each other, in contrast to Mod2-e, Mod4-pes, Mod5-pe and Mod7-es, which the difference between the number of True values y False values are almost half of them. This is mainly because the modification performed are related to the "isEmpty method". The columns "peek\_obj\_type-unique" refers to the total number of the possible type of values pushed in the Stack. The column "peek\_obj\_type-Most frequent" tells the object type most frequents, we know that "EmptyObject" is not a type. However, we decide to put it in this way when the Stack is empty. There are three possibilities to have an empty stack. i) The Stack is created for the first time. ii) The Stack had a size 1, and the method pop was executed, and iii) The Stack was cleaned by the Clean method. Given these three possible scenarios, we can expect that the most frequent type is "emptyObject."

Table 10 Summary of attributes Method and push Input

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **DS** | **# of Total rows** | **method-most frequent** | **method-Push** | **method-Pop** | **method-clear** | **pushInput-unique** | **pushInput-Most frequent** | **pushInput-Freq** |
| No\_mod | 86589 | push | 39311 | 15900 | 16393 | 13 | none | 47278 |
| Mod1-p | 86589 | push | 39311 | 15900 | 16393 | 13 | none | 47278 |
| Mod2-e | 46356 | push | 20750 | 8679 | 8357 | 12 | none | 25606 |
| Mod3-s | 86589 | push | 39311 | 15900 | 16393 | 13 | none | 47278 |
| Mod4-pes | 10385 | push | 3473 | 2036 | 1651 | 6 | none | 6912 |
| Mod5-pe | 46356 | push | 20750 | 8679 | 8357 | 12 | none | 25606 |
| Mod6-ps | 86589 | push | 39311 | 15900 | 16393 | 13 | none | 47278 |
| Mod7-es | 10385 | push | 3473 | 2036 | 1651 | 6 | none | 6912 |

In addition to the "testId", "instanceId", "isEmpty", "size", "peek\_obj\_type", and "method", we can extract the input type object when the method push is called; then, we will have other feature to analyse. The column "method-most frequent" refers to the method which is invoked more frequent. The columns "method-push", "method-pop", and "method-clear" refer to the total number of times that are invoked. In the documentation of Randoop says that the methods call in Randoop are distributed more or less uniformly; however, as Table 10 shows, the method push is almost twice time more call than others. The push method is invoked so many times to have different stack sizes. The columns "pushInput-unique" refers to the total number of possibles type of values pushed in the Stack. The column "pushInput -Most frequent" tells the most frequent push input, we know that none is not a method. However, we decide to put it in this way when the method called is different than push.

* + 1. **Step 2.2 –** ***State data pre-processing****:* In this step, the same process of phase 1 was performed.
    2. **Step 2.3 –** ***Apply the ruleset to the new SUT versions*:** In Activity 2.2.1, Select the rows that are different from the first SUT version:This activity is in charge of comparing and selecting the rows that are exclusively different from the first version of the SUT. Table 11 shows information about the size file of the different SUIT, i.e., the Stack information about the state of the Stack no modifies, and the modifies. As per Table 11 shows, Mod1\_p, Mod3\_s, and Mod6\_ps, the CSV file generated has the same number of rows than the No-mod version. In Mod2\_e, Mod4\_pes, Mod5\_pe, and Mod7\_es the number of rows are different from the no modify. We labelled with "Yes" or "Not" the rows in a new column named "diff". "Yes" is the row is different, i.e., it is a new column, and "Not" if the row is the same.

Table 11 State Report and Test Report

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DS** | **STATE REPORT (CSV)** | | | **TEST REPORT** | |
| **# of Total rows** | **# of rows that are NOT different from Non-mod (correct values)** | **# of rows that are different from Non-mod (incorrect values)** | **Test Run** | **Failures** | |
| Not-Mod | 71604 | 71604 | 0 | 2037 | 0 | |
| Mod1\_p | 71604 | 59722 | 11882 | 0 | |
| Mod2\_e | 37786 | 31396 | 6390 | 1452 | |
| Mod3\_s | 71604 | 28199 | 43405 | 0 | |
| Mod4\_pes | 7160 | 1663 | 5497 | 2012 | |
| Mod5\_pe | 37786 | 29994 | 7792 | 1452 | |
| Mod6\_ps | 71604 | 28199 | 43405 | 0 | |
| Mod7\_es | 7160 | 1663 | 5497 | 2012 | |

Table 12 summarised the results obtained in Activity 2.2.1 and Activity 2.2.2.

Table 12 Activity 2.2.1 and activity 2.2.2

|  |  |  |  |
| --- | --- | --- | --- |
| DS | # of uniques rows | # of unique rows that are NOT different from Non-mod (correct values) | # of unique rows that are different from Non-mod (incorrect values) |
| fs1\_original | 47 | 47 | 0 |
| fs1\_Mod1\_p | 45 | 17 | 28 |
| fs1\_Mod2\_e | 25 | 24 | 1 |
| fs1\_Mod3\_s | 47 | 1 | 46 |
| fs1\_Mod4\_pes | 4 | 1 | 3 |
| fs1\_Mod5\_pe | 26 | 12 | 14 |
| fs1\_Mod6\_ps | 45 | 1 | 44 |
| fs1\_Mod7\_es | 6 | 1 | 5 |
| fs2\_original | 76 | 76 | 0 |
| fs2\_Mod1\_p | 73 | 30 | 43 |
| fs2\_Mod2\_e | 33 | 31 | 2 |
| fs2\_Mod3\_s | 76 | 2 | 74 |
| fs2\_Mod4\_pes | 7 | 2 | 5 |
| fs2\_Mod5\_pe | 33 | 13 | 20 |
| fs2\_Mod6\_ps | 73 | 2 | 71 |
| fs2\_Mod7\_es | 9 | 2 | 7 |
| fs3\_original | 320 | 320 | 0 |
| fs3\_Mod1\_p | 232 | 135 | 97 |
| fs3\_Mod2\_e | 108 | 88 | 20 |
| fs3\_Mod3\_s | 320 | 83 | 237 |
| fs3\_Mod4\_pes | 14 | 8 | 6 |
| fs3\_Mod5\_pe | 106 | 60 | 46 |
| fs3\_Mod6\_ps | 232 | 76 | 156 |
| fs3\_Mod7\_es | 17 | 9 | 8 |
| fs4\_original | 320 | 320 | 0 |
| fs4\_Mod1\_p | 308 | 142 | 166 |
| fs4\_Mod2\_e | 108 | 88 | 20 |
| fs4\_Mod3\_s | 320 | 83 | 237 |
| fs4\_Mod4\_pes | 16 | 8 | 8 |
| fs4\_Mod5\_pe | 119 | 60 | 59 |
| fs4\_Mod6\_ps | 308 | 76 | 232 |
| fs4\_Mod7\_es | 17 | 9 | 8 |
| fs5\_original | 320 | 320 | 0 |
| fs5\_Mod1\_p | 398 | 209 | 189 |
| fs5\_Mod2\_e | 108 | 88 | 20 |
| fs5\_Mod3\_s | 320 | 83 | 237 |
| fs4\_Mod4\_pes | 17 | 10 | 7 |
| fs5\_Mod5\_pe | 133 | 67 | 66 |
| fs5\_Mod6\_ps | 398 | 119 | 279 |
| fs5\_Mod7\_es | 17 | 9 | 8 |

The Activity 1.2.3 Apply the ruleset against the new unique rows, was successfully performed, and the results are used to answer the research questions.

* 1. **RQ1: How well does the rule mining approach represent the behaviour of the SUT state?**

To address this research question, we apply the set of rules to the unique rows of each dataset. Then, we analysed its behaviour by checking how many rows are matching or violating at least one rule. Table 11 provides brief information about them. It is important to highlight that those datasets are the output of the test driver, i.e., State Report. Table 11, also provides information about the Test Report, which is the report generated when the different Regression tests are executed (pass/fail).

As per table 11, we can observe that the number of failed tests is the same for the Mod2\_e and Mod5\_pe datasets. Also, the number of failed tests are the same for Mod\_4 and Mod\_7. The Test log also shows the same behaviour, but with the number of total rows, Mod2\_e and Mod5\_pe have the same number of rows. The common aspect between all these datasets is that all of them have the isEmpty method modification. In particular, Mod2\_e and Mod5\_pe have 1452 failed test. The Mod5\_pe modification is the combination of the modified methods peek and isEmpty. However, it seems that the regression test spots the fault related to "isEmplty" only. Furthermore, as per table 11 shows, the Mod1\_p, which is the modification of Peek, none regression tests failed. This fact confirms the regression tests failed of Mod5\_pe are belonging to isEmpty modification only.

Same as Mod1\_p, none regression test failed in Mod3\_s. In Mod3\_s "size" method is modified; then, we can expect similar behave than Mod2\_e and Mod5\_pe. Mod4\_pes is the combination of the modifieds "peek", "isEmpty" and "size" methods. We know that none regression tests are failed in "peek" and "size" modifications. However, the number of tests failed in Mod4\_pes and Mod7\_es are different from Mod2\_e and Mod5\_pe. Are the regression tests spotting faults in the other modified methods, i.e., "peek" and "size" when combined in this way? The modification of size() returns the incorrect size for the Stack class instances that contain one or more objects, i.e., when the "size" of the Stack is greater than 0, the modification would return the correct size plus one. For instance, if the "size" is one, the modified version will return "size" equal to two. That is why the combination of "size" and "isEmpty" modifications increase the number of failed tests because the size modified method increases the number of size = 2, then triggers the modified isEmpty which returns "isEmpty= True" in the cases when the size of the Stack class instance is 2 or 0.

To measure how well the ruleset represents the behaviour of the SUT state, we measure the rule coverage ratio. That is the total number of rows that were matched, and the number of rows that were violated. Figure 9 shows this ratio. As per Figure 13, we can observe good coverage for the whole dataset for the FS-8, FS-9, and FS-10. In particular, both FS-9 and FS-10, a high ratio was expected, since they have much more number of rules than FS-3 or FS-4

Figure 13 Rule coverage ratio.

* 1. **RQ2: How strong/strict the rule mining is?**

We measured the soundness of the rule mining approach, by the number of rules generates, also by quantifying the proportion of rules matched and violated by the rows. For unique new data rows in each dataset, we check how many rows match every rule at least once. Also, we check how many rows are matching more than one rule, and analysed the number of rows that violate rules. We considered a rule violation when a determinate row is matching with LHS but is not matching with RHS. Summarises the results obtained.

To address this research question, we compared the number of unique rows that are different from Non-mod (incorrect values) with the number of unique rows that VIOLATE rules. Figure 14 shows the number of unique rows per each dataset used, as we can see, the number of unique rows depend on the number of features used.

Figure Total unique rows per dataset and per modification

Figure 15 shows the number of unique rows that are different from the Non-Modified dataset; we assume all the rows that are different as an incorrect value

Figure Number of unique rows that are different from Non-mod (incorrect values)

Table 12 summarised the number of rules that are matched and violated for each dataset and each FS configuration. Overall, all the rules set and all datasets, all the rules are matched, there is no one rule which is not used. In FS-3, 14 rules were generated; all the rules were used, matched or violated, the Mod1\_p, Mod3\_s and Mod6\_ps did not find rules violation. Mod2\_e, Mod4\_pes, and Mod7\_es found rules violation. However, they found the same number of violation and the same rules. The common aspect between all these datasets is that all of them have the "isEmpty" method modification. Then, it seems the violated rules are pointing out the same error isEmpty and no other modifications. In FS-4 it seems to happen the same than FS-3 even with more number of rules.

FS-8 has interesting behaviour to point out. First, similar to FS-3 and FS-4, there is no rule violation for mod1\_p, even with such a high number of rules, 439 rules. Second, the Mod2\_e and Mod5\_pe (coloured in blue) share the same amount of rules violated. We inspected manually and discover that the rules violated are the same. This is because the rules are not able to find any related to the modification Peek. Mod5\_ep is a combination of "peek" and "isEmpty", then the set of rules are pointing out the same "isEmpty" violations. The same situation happens with Mod3\_s and Mod6\_ps (coloured in green) since Mod6\_ps is a combination of "peek" and "size"; then, the rules are pointing out the same "size" violations. As the previous modifications, Mod4\_pes and Mod7\_es (coloured in brown) are pointing out the modification "isEmpty" and "size" which is Mod7\_es but not the modification peek in Mod4\_pes. The set of rules from FS4 and FS-10 can find violations in Mod1\_p, the interesting fact is they find the same number of rules, and the same rules. FS-10 has 1450 rules, while FS-9 has 679 only, some of the rules from 679 may also be in the set of rules of FS-10. That is why the rules that are violated in Mod1\_p in FS-9 are the same in FS-10. Both FS-9 and FS-10, the number of violations, does not have the same number of violation like FS-3, FS-4 and FS-8.

Table 12 Report on the number of totals of rules matched and violated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **DS** | **# of Rules** | **Total # of rules Matched (LHS and RHS)** | **Total # of rules Violated (LHS match but NOT RHS)** | **Match-intersection-Violate** | **Total # of rules used (MUV)** |
| **FS-3\_original** | **14** | **14** | **0** | **0** | **14** |
| **FS-3\_Mod1\_p** | **14** | **14** | **0** | **0** | **14** |
| **FS-3\_Mod2\_e** | **14** | **14** | **3** | **3** | **14** |
| **FS-3\_Mod3\_s** | **14** | **14** | **0** | **0** | **14** |
| **FS-3\_Mod4\_pes** | **14** | **14** | **3** | **3** | **14** |
| **FS-3\_Mod5\_pe** | **14** | **14** | **3** | **3** | **14** |
| **FS-3\_Mod6\_ps** | **14** | **14** | **0** | **0** | **14** |
| **FS-3\_Mod7\_es** | **14** | **14** | **3** | **3** | **14** |
| **FS-4\_original** | **36** | **36** | **0** | **0** | **36** |
| **FS-4\_Mod1\_p** | **36** | **36** | **0** | **0** | **36** |
| **FS-4\_Mod2\_e** | **36** | **36** | **4** | **4** | **36** |
| **FS-4\_Mod3\_s** | **36** | **36** | **0** | **0** | **36** |
| **FS-4\_Mod4\_pes** | **36** | **36** | **4** | **4** | **36** |
| **FS-4\_Mod5\_pe** | **36** | **36** | **4** | **4** | **36** |
| **FS-4\_Mod6\_ps** | **36** | **36** | **0** | **0** | **36** |
| **FS-4\_Mod7\_es** | **36** | **36** | **4** | **4** | **36** |
| **FS-8\_original** | **439** | **439** | **0** | **0** | **439** |
| **FS-8\_Mod1\_p** | **439** | **439** | **0** | **0** | **439** |
| **FS-8\_Mod2\_e** | **439** | **439** | **73** | **73** | **439** |
| **FS-8\_Mod3\_s** | **439** | **439** | **95** | **95** | **439** |
| **FS-8\_Mod4\_pes** | **439** | **439** | **113** | **113** | **439** |
| **FS-8\_Mod5\_pe** | **439** | **439** | **73** | **73** | **439** |
| **FS-8\_Mod6\_ps** | **439** | **439** | **95** | **95** | **439** |
| **FS-8\_Mod7\_es** | **439** | **439** | **113** | **113** | **439** |
| **FS9\_original** | **676** | **676** | **0** | **0** | **676** |
| **FS9\_Mod1\_p** | **676** | **676** | **12** | **12** | **676** |
| **FS9\_Mod2\_e** | **676** | **676** | **142** | **142** | **676** |
| **FS9\_Mod3\_s** | **676** | **676** | **95** | **95** | **676** |
| **FS9\_Mod4\_pes** | **676** | **676** | **170** | **170** | **676** |
| FS9\_Mod5\_**pe** | 676 | 676 | 154 | 154 | 676 |
| FS9\_Mod6\_**ps** | 676 | 676 | 107 | 107 | 676 |
| FS9\_Mod7\_**es** | 676 | 676 | 167 | 167 | 676 |
| **FS10\_original** | **1450** | **1450** | **0** | **0** | **1450** |
| **FS10\_Mod1\_p** | **1450** | **1450** | **12** | **12** | **1450** |
| **FS10\_Mod2\_e** | **1450** | **1450** | **224** | **224** | **1450** |
| **FS10\_Mod3\_s** | **1450** | **1450** | **285** | **285** | **1450** |
| **FS9\_Mod4\_pes** | **1450** | **1450** | **246** | **246** | **1450** |
| FS9\_Mod5\_**pe** | 1450 | 1450 | 236 | 236 | 1450 |
| FS9\_Mod6\_**ps** | 1450 | 1450 | 297 | 297 | 1450 |
| FS9\_Mod7\_**es** | 1450 | 1450 | 314 | 314 | 1450 |

* 1. **Q3: What faults can the rule mining approach detect?**

We apply rules to the new rows and observe what data rows were violated; then, we check whether the rules that were violated pointed out the mutants introduced. The previous analysis shows that the rules can detect faults. We can consider each rule violation as a fault which has been detected. Now, we must to localised such fault.

We used performance measures derived from the Confusion Matrix. Figure 16 provides better visualisation of the confusion matrix in our context. Let A and A' denote a classification (e.g., an incorrect value found); then each reference standard measure is expressed as a function of the Confusion Matrix defined as follows:

* True Positive (TP): The actual class of a case was A, and the predicted class was A. This represents a successful prediction. The row with the incorrect value violates at least one rule
* True Negative (TN): The actual class of a case was A' and the predicted class was A'. This represents a successful prediction. The row with a correct value does match with at least one rule.
* False Positive (FP): The actual class of a case was A' and the predicted class was A. This represents an unsuccessful prediction. The row with the correct value violates with at least one rule
* False Negative (FN): The actual class of a case was A, and the predicted class was A'. This represents an unsuccessful prediction. The row with the correct value violates at least one a rule

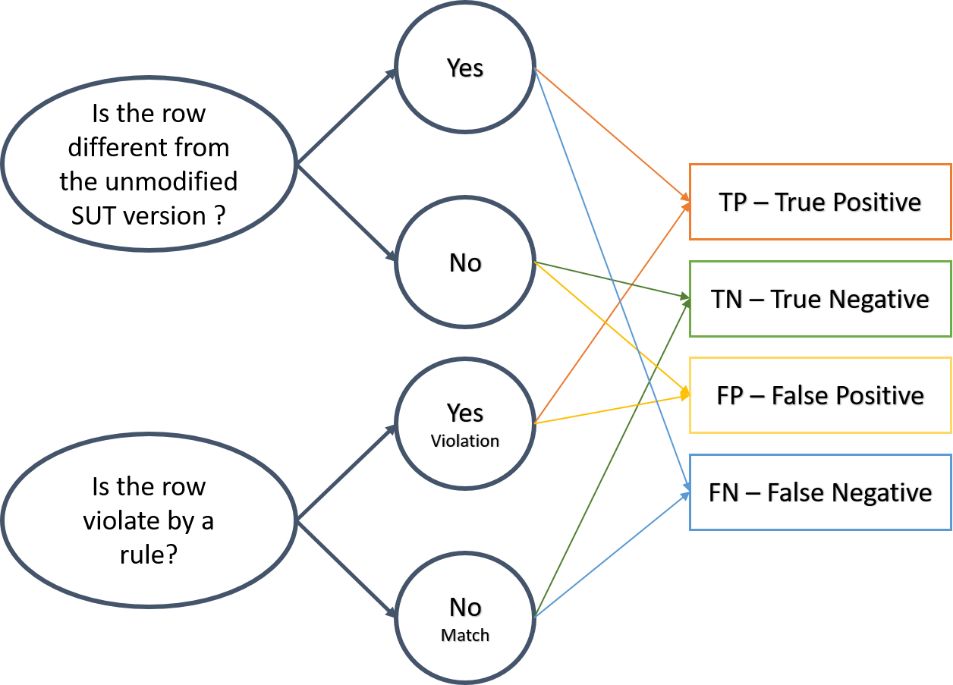
Table 13 provides the confusion matrix for all dataset analysed. The performance measures in this paper are given by five metrics derived from the Confusion Matrix, viz., True Positive Rate (TPR), False Positive Rate (FPR), True Negative Rate (TNR), False Negative Rate (FNR), Accuracy, Precision, Missclasification Rate, and f-measure.

Figure Relations between TP, TN, FP, and FN in the context of the ARM approach

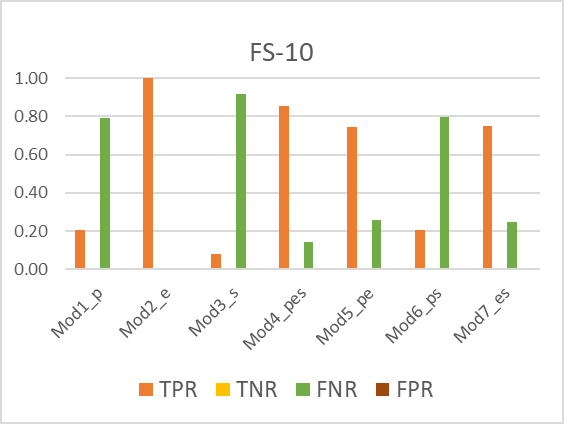
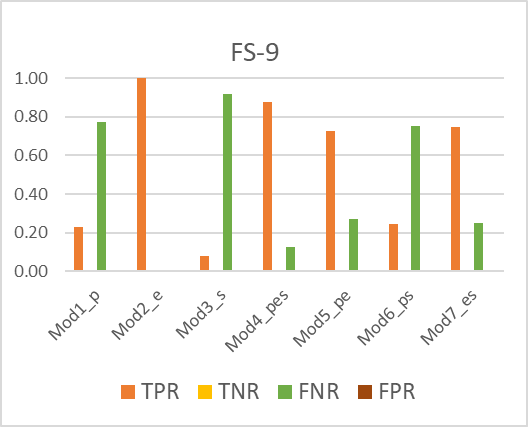
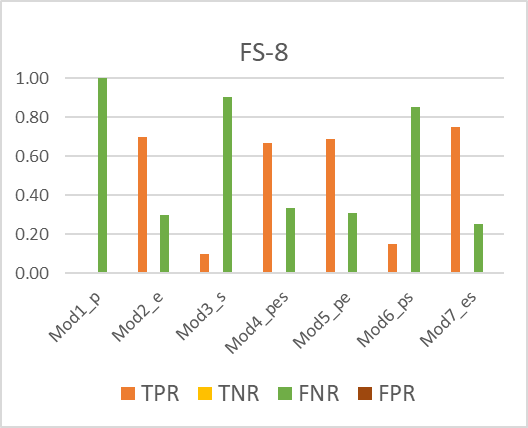
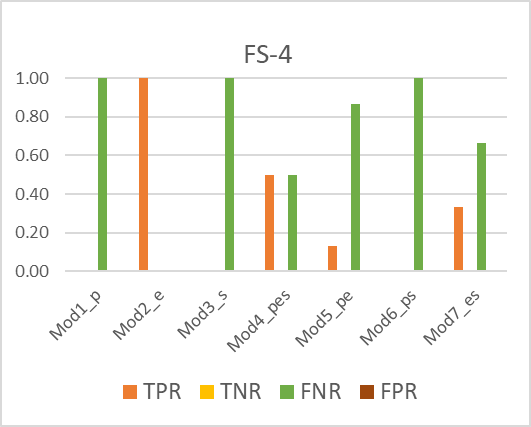
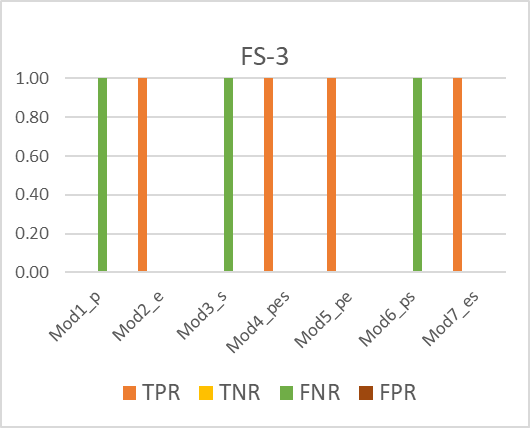
* **TPR** measures the proportion of positives that are correctly identified (e.g., the percentage of different rows from the unmodified SUT version, which are violating at least one rule). The TPR is expressed as:
* **FPR** (also known as **fall-out** or **false alarm ratio)** is the probability of falsely rejecting the null hypothesis for a particular test. The FPR is calculated as the ratio between the number of negative events wrongly categorised as positive (FP) and the total number of actual negative events (FN and TN). The FPR is expressed as:
* **TNR** measures the proportion of negatives that are correctly identified (e.g., the percentage of equals rows from the unmodified SUT version, which is not violating rules). In our case, only the rows that are exclusively different from the first version of the SUT are examining, i.e., this measure is 0. The TNR is expressed as:
* **FNR** also know as ***Miss rate*** is a verification measure of categorical forecast performance equal to the number of false alarms or FN divided by the total number of event forecasts (FN and TN)
* **Accuracy** is the ratio of successful predictions made to both classes and expressed as:
* **Precision** (or positive predictive value) is the ratio of correct predictions made for class A and expressed as:
* **F-measure:** The f-measure statistic (or F1 score) considers both the TPR and precision of a classifier to measure its quality

Since we only consider the rows that are different from the unmodified version of the SUT, we assume that those are the wrong values. The confusion matrix reduces TP and FN only.

Table 13 Confusion matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DS | Rows NOT matched and NOT violated | TP | FN | FP | TN | Total of rows analised |
| FS-3\_Mod1\_p | **28** | 0 | 14 | 0 | 0 | 28 |
| FS-3\_Mod2\_e | **0** | 1 | 0 | 0 | 0 | 1 |
| FS-3\_Mod3\_s | **46** | 0 | 1 | 0 | 0 | 46 |
| FS-3\_Mod4\_pes | **2** | 1 | 0 | 0 | 0 | 3 |
| FS-3\_Mod5\_pe | **13** | 1 | 0 | 0 | 0 | 14 |
| FS-3\_Mod6\_ps | **44** | 0 | 0 | 0 | 0 | 44 |
| FS-3\_Mod7\_es | **4** | 1 | 0 | 0 | 0 | 5 |
|  |  |  |  |  |  |  |
| DS | **Rows NOT matched and NOT violated** | **TP** | **FN** | **FP** | **TN** | **Total of rows analised** |
| FS-4\_Mod1\_p | **15** | 0 | 28 | 0 | 0 | 43 |
| FS-4\_Mod2\_e |  | 2 | 0 | 0 | 0 | 2 |
| FS-4\_Mod3\_s | **28** | 0 | 46 | 0 | 0 | 74 |
| FS-4\_Mod4\_pes | **1** | 2 | 2 | 0 | 0 | 5 |
| FS-4\_Mod5\_pe | **5** | 2 | 13 | 0 | 0 | 20 |
| FS-4\_Mod6\_ps | **27** | 0 | 44 | 0 | 0 | 71 |
| FS-4\_Mod7\_es | **1** | 2 | 4 | 0 | 0 | 7 |
|  |  |  |  |  |  |  |
| DS | **Rows NOT matched and NOT violated** | **TP** | **FN** | **FP** | **TN** | **Total of rows analised** |
| FS-8\_Mod1\_p | **9** | 0 | 88 | 0 | 0 | 97 |
| FS-8\_Mod2\_e |  | 14 | 6 | 0 | 0 | 20 |
| FS-8\_Mod3\_s | **24** | 21 | 192 | 0 | 0 | 237 |
| FS-8\_Mod4\_pes |  | 4 | 2 | 0 | 0 | 6 |
| FS-8\_Mod5\_pe | **1** | 31 | 14 | 0 | 0 | 46 |
| FS-8\_Mod6\_ps | **16** | 21 | 119 | 0 | 0 | 156 |
| FS-8\_Mod7\_es |  | 6 | 2 | 0 | 0 | 8 |
|  |  |  |  |  |  |  |
| DS | **Rows NOT matched and NOT violated** | **TP** | **FN** | **FP** | **TN** | **Total of rows analised** |
| FS-9\_Mod1\_p |  | 38 | 128 | 0 | 0 | 166 |
| FS-9\_Mod2\_e |  | 20 | 0 | 0 | 0 | 20 |
| FS-9\_Mod3\_s |  | 19 | 218 | 0 | 0 | 237 |
| FS-9\_Mod4\_pes |  | 7 | 1 | 0 | 0 | 8 |
| FS-9\_Mod5\_pe |  | 43 | 16 | 0 | 0 | 59 |
| FS-9\_Mod6\_ps |  | 57 | 175 | 0 | 0 | 232 |
| FS-9\_Mod7\_es |  | 6 | 2 | 0 | 0 | 8 |
|  |  |  |  |  |  |  |
| DS | **Rows NOT matched and NOT violated** | **TP** | **FN** | **FP** | **TN** | **Total of rows analised** |
| FS-10\_Mod1\_p |  | 39 | 150 | 0 | 0 | 189 |
| FS-10\_Mod2\_e |  | 20 | 0 | 0 | 0 | 20 |
| FS-10\_Mod3\_s |  | 19 | 218 | 0 | 0 | 237 |
| FS-10\_Mod4\_pes |  | 6 | 1 | 0 | 0 | 7 |
| FS-10\_Mod5\_pe |  | 49 | 17 | 0 | 0 | 66 |
| FS-10\_Mod6\_ps |  | 57 | 222 | 0 | 0 | 279 |
| FS-10\_Mod7\_es |  | 6 | 2 | 0 | 0 | 8 |

Figure 17 TPR, TNR, FNR, and FPR per dataset



In our case, TPR means the rate of the number of rows with incorrect values, and at the same time, they violate at least one rule. The desired value for TPR is 1. TNR refers to the number of rows with correct values, and at the same time, they match with at least one rule. Similar to TPR, the desired value for TNR is 1. In our case, we only consider the rows that are different from the unmodified version of the SUT; we assume that those are the wrong values. Then, we don't have TNR

FNR means the rate of the number of rows that are different from the unmodified SUT but match at least one rule. An indication of good performance is when this metric remains close to zero, the same for FPR since FPR refers to the number of rows with correct values and at the same time violates at least the rule.

In Figure 17, we can see that the TPR is very low and at some point zero for Mod1\_p in almost all data sets. Although some rules are violated for the data sets, FS-9 and FS-10, the set of generated rules do not fully manage to find this modification. The best performance against the TPR metric is when it is used to find Mod2\_e. The TPR in FS-8 is less than one for Mod2\_e. However, it is above 0.6, which is a good measurement. In general, the worst TPR measure is related to the size modification, for example, Mod3\_s, Mod6\_ps. In these modifications, the TPR is less than 0.2

It seems that the modifications with the best performance regarding TPR and FNR, i.e., TPR close to one and FNR close to zero, are the metrics that are related to the Empty modification, for example, Mod2\_e, Mod4\_pes, Mod5\_ep, and Mod7\_es. Overall, the results show high FNR, it means that the ARM is not an the optimal approach. Since for each FN, it is an analysis that will lead to a misinterpretation.

Several performance metrics can be calculated from the confusion matrix such as accuracy (Figure 18), precision (Figure 19), and f-measure (Figure 20). Below we provide each of them. The accuracy of the proposed approach is affected by the small amount of TP, and a large amount of FN. The contrast between precision and accuracy is significant. This contrast occurs since in the precision the amount of TN is positive in almost all the data sets.

Figure Accuracy per dataset used

Figure Precision per data set

Figure f-measure per data set used

1. **Discussion/Conclusion**

The results obtained allow us to address the last research question*: What information regarding faults detection and location can the methodology offer?* From our approach, we can get three different results: *i)* Tests Report, *ii)* State Report, and *iii)* Rule validation Report. *i)* The Test Report refers to the regression test reports (pass/fail), Table 11. If we manually check the tests that failed, perhaps we can find the errors. Of seven modifications made, regression tests found only those that were related to isEmpty. This is the case of Mod1-p, Mod3-s and Mod6-ps. As we explained earlier, we can see that the number of failed tests is the same for the Mod2\_e and Mod5\_pe data sets. Also, the number of failed tests is the same for Mod\_4 and Mod\_7. The test log also shows the same behaviour, but with the total number of rows, Mod2\_e and Mod5\_pe have the same number of rows. All those data sets share the modification of the isEmpty method. In particular, Mod2\_e and Mod5\_pe have 1452 failed tests. The Mod5\_pe modification is the combination of the modified peek and isEmpty methods. However, it seems that the regression test detects the failure related to "isEmplty" only. Also, as Table 11 shows, Mod1\_p, which is Peek's modification, did not fail any regression tests. This fact confirms that the failed regression tests of Mod5\_pe pertain only to the modification "isEmpty. "

*ii)* The SUT State Report is a CSV file generated by the Test Driver. Assuming the first version of the SUT is correct (the SUT has no flaws), we can compare that CSV file with the new versions, that is, the new CSV files. By comparing the two files, we can easily distinguish which rows are different from the first version. Our approach is based on the fact that the relationships in the state should not change significantly when new versions of the SUT are tested. Thus, we can consider different rows in the new SUT as potential faults. Although the comparison between two files seems to be simple, it is complex. A row-by-row comparison is possible when the new SUT version passes all the regression test. For instance, Mod1\_p, Mod3\_s, and Mod6\_ps since the CSV file generated are the same size than the No-mod version. However, if the new SUT version fails son regression test, the CSV file will be generated with different size, e.g., in Mod2\_e, Mod4\_pes, Mod5\_pe, and Mod7\_es. The comparison between those tells about the number of missing rows only.

The rule validation report is a report provided by the rule mining approach. This approach can detect that something is wrong when a rule is violated. Then, by analysing the violated rules, it is possible to locate the fault. However, in terms of time, the rule mining approach is time-consuming, particularly when the number of rules is high, for example in the rules set of FS-8, FS-9, and FS-10, which has 439, 767 and 1450 rules. Checking all the rules in each data set takes more than 1 hour for unique new rows. Also, the downside of the ARM approach is its FN rate. The FNR in most of the datasets studied is high. This represents a great disadvantage because each rule that is violated must be analysed to find the fault if you have a lot of rules violated, but they are lying, we wasted time analysing them rules.

**References**

[1] J. Caldeira, F. B. e Abreu, J. Cardoso, and T. Oliveira, "Software Development Analytics in Practice: A Systematic Literature Review," 2020.

[2] K. Patel and R. M. Hierons, "A partial oracle for uniformity statistics," *Softw. Qual. J.*, vol. 27, no. 4, pp. 1419–1447, 2019, doi: 10.1007/s11219-019-09459-0.

[3] S. K. Solanki and J. T. Patel, "A survey on association rule mining," *Int. Conf. Adv. Comput. Commun. Technol. ACCT*, vol. 2015-April, pp. 212–216, 2015, doi: 10.1109/ACCT.2015.69.

[4] N. Parashar, "Classification using association rule mining over mammogram images: A review," *Int. J. Emerg. Technol.*, vol. 11, no. 2, pp. 954–957, 2020.

[5] L. Zhou and S. Yau, "Efficient association rule mining among both frequent and infrequent items," *Comput. Math. with Appl.*, vol. 54, no. 6, pp. 737–749, 2007, doi: 10.1016/j.camwa.2007.02.010.

[6] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," *Proc. - Int. Conf. Softw. Eng.*, pp. 75–84, 2007, doi: 10.1109/ICSE.2007.37.

[7] "Efficient-Apriori — Efficient-Apriori 1.1.1 documentation." [Online]. Available: https://efficient-apriori.readthedocs.io/en/latest/. [Accessed: 10-Jun-2020].