Why Did the Test Execution Fail? Failure Classification Using Association Rules

1st Claudius Jordan
Technical University of Munich
Munich, Germany
0000-0002-4341-8701

2nd Philipp Foth
Technical University of Munich
Munich, Germany
0000-0003-4753-1979

3rd Matthias Fruth
TraceTronic GmbH
Dresden, Germany
matthias.fruth@tracetronic.de

4th Alexander Pretschner
Technical University of Munich
Munich, Germany
0000-0002-5573-1201

***Abstract*—Testing automotive electronic control units and their software, the system-under-test (SUT) in our context, requires complex test infrastructure setups. As those setups are developed in parallel to the SUT, often a problem in the test infrastructure instead of the SUT causes a failed test case execution (TCE). We call such unexpectedly failing TCEs *invalid*. As there are several reasons that lead to such invalid test failure, failed TCEs are manually analyzed and categorized. This failure classification is a time-consuming task. Thus, automatic classification could significantly reduce overall development time and cost. A previous study suggests using association rule learning (ARL) to classify failed TCEs as valid or invalid based solely on test step information. In this work, we extend this ARL-based approach to our multi-class setting and evaluate its application on data from five running verification & validation projects in the automotive industry. In total, we predict the defect classes of more than 85k TCEs and achieve an overall precision up to 86.7% with an overall recall up to 57.4%. With this work, we offer evidence that the application of said approach, originally presented in the context of information systems, can be fruitful in automotive integration- and system-level testing contexts as well.**

***Index Terms*—hardware-in-the-loop, system testing, integration testing, failure classification, association rule mining, test infrastructure, automotive**

I. Introduction

The increasing share of embedded software in modern automotive systems magnifies the ever growing need for systematic and automated support throughout all phases of the development cycle, especially the testing activities. Due to the complex functionality and the variant diversity of modern vehicles, the required test infrastructures, which are used for integration- and system-level testing, necessarily also grow in their complexity. Such test infrastructures are built up and continuously evolve in parallel to the system-under-test (SUT) development. We consider every component involved in the testing process – hence, everything apart from the SUT – to be part of the test infrastructure: In particular, there are the hardware-in-the-loop test benches (HiLs) that are composed of hardware as well as software components, e.g. for simulation purposes, and there are several different software tools needed to enable the test automation. In practice, not only hardware components suffer from sporadic malfunctioning but also involved software tools [1]–[5]. The fact that test infrastructures also need to be considered error-prone only recently gained attention in the literature [2], [4], [5]. This leads to tests failing because of a test infrastructure issue rather than a defect in the SUT.

The resulting problem is that produced test results cannot be regarded reliable. Therefore, practitioners manually investigate the causes for failed test case executions (TCEs) and, most importantly, determine whether the test results are trustworthy. In other words, they decide whether a failed TCE is *valid*, i.e., a flaw in the SUT has been discovered, or *invalid*, i.e., the test case fails for any other reason. As domain experts report, a substantial portion of the daily work load is associated with this investigation and, thus, is an important cost factor for validation & verification (V&V) projects in the studied context. In fact, in a recent study [5] from the automotive domain, it is reported that up to 91% of failed TCEs are invalid. There, the authors distinguish between sporadic test infrastructure induced failures and other causes, which include test case implementation errors and incorrect test specifications. Given the magnitude of the problem, it becomes apparent that reducing the burden for the experts to manually analyze the TCEs leads to a significant productivity gain. Hence, the technical challenge addressed in this work is the automatic classification of failed TCEs in automotive integration- and system-level black-box regression testing setups.

In the context of business information systems, Herzig and Nagappan [6] suggest to use association rule learning (ARL) [7] to detect patterns of failing test steps that allow to automatically distinguish valid from invalid test results. For our context, however, it is not sufficient to perform such a binary classification because there are distinct reasons why invalid test failures occur, and in our context it is critical to discriminate between them: Depending on the underlying failure cause, the appropriate action is to be performed, i.e., the respective stakeholder is identified who is responsible to take care of the revealed problem. If, e.g., a test fails due to misconfiguration, a tester can react and correct the mistake. If, however, the simulation environment is flawed, the respective expert needs to be informed. Likewise, if an actual flaw in the SUT is identified the developers are informed. While other approaches that distinguish failure categories depend on context-specific test log processing, e.g. [8], we rely solely on test step data, i.e., verdicts of separable steps a test case is composed of, which in our context are readily available.

For this practical experience report, we hence borrow the idea of ARL based on test step information for our multi-class failure classification problem. Concretely, we shed light on the transferability of said approach to our context and answer the following question: *How well can failed TCEs in automotive testing setups be classified according to their failure cause using test step-based ARL?*

In summary, we make the following contributions:

- We extend the failed TCE classification approach by Herzig and Nagappan [6] to a multi-class setting for multiple failure categories. Additionally, we apply rule pruning strategies [9].
- We report our practical experiences and lessons learned with the ARL-based failure classification in a new domain.
- We collected data from five real-life projects in the automotive domain containing more than 85k labeled failed TCEs. We achieve an overall precision up to 86.7% with an overall recall up to 57.4%, which are similar to the results reported in [6].
- In addition, we analyze the impact of the *minimum confidence* parameter as well as limiting the training data. We find that time windows between 30 to 90 days deliver the best results for the studied projects. This indicates that already data from around one to three months is sufficient to fruitfully apply the ARL-based approach.

II. Association Rule Learning and Classification

ARL [7] is a machine-learning method that learns statistical connections in the form of implications between items i within a set of items $\mathcal{I}$ in a dataset D. Such statistical connections are encoded as so-called *association rules* – hence the name *association rule learning*. These association rules r can be used for classification tasks as we will see in the following.

A. Preliminaries

In general, ARL can be applied to a dataset D that can be represented as a list of N itemsets: $D = [I_1, ..., I_N]$. For any j, *itemset* $I_j \subseteq \mathcal{I}$ consists of any number of all available items $i \in \mathcal{I}$. A popular example for ARL deals with grocery shopping, where the set of items $\mathcal{I}$ are the products that can be bought in a store, e.g., bread, milk and eggs, and the dataset D is represented as a list of shopping carts I_j each containing a subset of such items.

As mentioned before, we are interested in statistical connections between items i in the dataset that can be encoded as implications. Such an implication, an association rule r, consists of a set of antecedent items $A \subseteq \mathcal{I}$, and a single consequent item $c \in \mathcal{I} \setminus A$. A rule can then be interpreted and written as follows: For a given itemset I_j, if it contains a certain combination of antecedent items A, this implies that it is likely to also contain the consequent item c:

$$r \ = \ (\ A \Rightarrow \{c\}\ )$$

In order to measure the quality of such rules, there are two important parameters that specify the minimum requirements for a rule to be learned: *minimum support* and *minimum confidence*. The *support* of an association rule r quantifies the relative frequency of the rule itemset R in the entire dataset, where $R = A \cup \{c\}$:

$$\text{supp}(r) = \text{supp}(R) = \frac{|\{I_j \in D | R \subseteq I_j\}|}{|D|}$$

Here, $\{I_j \in D | R \subseteq I_j\}$ are those itemsets $I_j \in D$ that contain the rule itemset R. It acts as a sort of statistical significance measure, and by setting a minimum value, it limits which itemsets are to be considered for rule learning. Besides, a limit for the *maximum antecedent set size* can also be introduced. Both can drastically reduce the number of sets to consider, and as a result reduce the computational cost.

The *confidence* of an association rule quantifies how often the antecedent set appears together with the consequent, relative to with or without the consequent:

$$\text{conf}(r) = \frac{\text{supp}(R)}{\text{supp}(A)} = \frac{|\{I_j \in D | R \subseteq I_j\}|}{|\{I_j \in D | A \subseteq I_j\}|}$$

This is a measure of how strongly the antecedent is connected to the consequent. By setting a minimum value only rules with a larger statistical confidence in the connection they represent are learned. Additionally, it is possible to restrict learning rules to those with a consequent from a given set of consequent items. This is necessary in our use case, because we use the association rules for classification. If there is a learned association rule r that applies to a set of antecedents $\tilde{A}$ the corresponding consequent c can be assigned as the label to $\tilde{A}$.

B. Association Rules based on Test Steps – An Example

For our purposes, we use ARL to find statistical connections between test step verdicts of failed TCEs and failure causes. Hence, the dataset D is the history of TCEs. Each TCE consists of test steps and a so-called *defect class*. For the itemset I_j, we use as *step items* the test step denominations joined with their respective verdict and the defect class is used as the consequent c. Hence, for a any labeled failed TCE its itemset I_j in our dataset D consists of step items and the defect class label.

Note, in our context, a test step is an operation, e.g., reading or writing a value or carrying out calculations or comparisons. Consequently, each test step has a corresponding verdict depending on the test case implementation, which can be one of $Passed$ (P), $Inconclusive$ (I), $Failed$ (F), $Error$ (E), or $None$ (N). In general, $Passed$ indicates that the expectations of the test step are met, respectively $Failed$ indicates that they are not. The verdict $Inconclusive$ is assigned when the expectation cannot be evaluated, e.g., if a precondition for a check is not met. A test step is assigned the verdict $Error$, e.g., when the test step as such cannot be performed correctly. $None$ is used as the default verdict, if no particular other verdict is assigned, e.g., if there is no expectation associated with a test step.

Have a look at an exemplary dataset D consisting of five failed TCEs ($D = [I_1, ..., I_5]$) from the same test case:

[TCE_id 1 : { $step_1{=}E$, $step_2{=}P$, $step_3{=}E$, $label{=}Hardware$ } ,
TCE_id 2 : { $step_1{=}E$, $step_2{=}P$, $step_3{=}F$, $label{=}Hardware$ } ,
TCE_id 3 : { $step_1{=}E$, $step_2{=}F$, $step_3{=}P$, $label{=}Hardware$ } ,
TCE_id 4 : { $step_1{=}E$, $step_2{=}F$, $step_3{=}E$, $label{=}CodeDefect$ } ,
TCE_id 5 : { $step_1{=}F$, $step_2{=}F$, $step_3{=}P$, $label{=}CodeDefect$ }]

The exemplary test case consists of three generic steps and, depending on the TCE, these steps are assigned respective verdicts. In addition, each TCE is also assigned a label. In the example the labels are *Hardware* or *Code Defect*, meaning TCEs with Ids 1-3 failed due to some problem with the hardware and the ones with Ids 4-5 revealed an actual code defect in the SUT. Thus, for this example $\mathcal{I}{=}\{step_1{=}E,\ step_1{=}F,\ step_2{=}P,\ step_2{=}F,\ step_3{=}E,\ step_3{=}P,\ label{=}Hardware,\ label{=}CodeDefect\}$. Now, given an exemplary minimum support of 0.4 and a minimum confidence of 0.66, the following four rules are learned:

Id	conf	supp	Rule
1	1.0	0.4	$\{step_2{=}P\} \Rightarrow \{label{=}Hardware\}$
2	1.0	0.4	$\{step_1{=}E,\ step_2{=}P\} \Rightarrow \{label{=}Hardware\}$
3	0.75	0.6	$\{step_1{=}E\} \Rightarrow \{label{=}Hardware\}$
4	0.67	0.4	$\{step_2{=}F\} \Rightarrow \{label{=}CodeDefect\}$

Note, the learned rules can be interpreted by humans, however they do not necessarily represent causal relationships. Rule 1, for instance, has a confidence of 100% and implies that any TCE including step item $step_2{=}P$ (meaning $step_2$ is assigned the verdict $Passed$) fails because of a test infrastructure hardware problem. Intuitively, a passing step is not expected to be an indicator for any type of failure. Yet, for the given dataset of only failed TCEs, the $Passed$ verdict for this step happens to be sufficient (statistically speaking) to distinguish hardware problems from other failure causes.

Given a TCE with $\tilde{A} = \{step_1{=}E,\ step_2{=}E,\ step_3{=}P\}$. As rule 3 applies, the ARL-based classifier assigns label $Hardware$.

C. Pruning Strategies

To construct a classifier based on ARL, we follow the main ideas of Liu et al. [9]. ARL often generates an extremely large number of rules for classification data due to combinatorial explosion. So after rule generation, we prune the rule set as a post-processing step with what we refer to as *data coverage pruning*. As in [9], we sort the rules by highest confidence, highest support, and smallest antecedent rule set size. Then we iterate through the rules and apply them to all samples in the training dataset. Each training sample that gets correctly classified by a rule is flagged. If a rule (further down the list) fails to correctly classify at least one unflagged training sample, it is removed from the rule set. In the example above, only Rule 2 is removed because Rule 1 already correctly classifies all of the TCEs that Rule 2 also does. In practice this drastically reduces the size of the rule set while often even slightly improving the classification performance [9]. We also observe this performance improvement in our case study as we report in Sec. V-B.

In general, a classifier is expected to predict a label for each given input sample. Therefore, Liu et al. [9] suggest to add a *default rule* to the very end of the rule set. A default rule consists of an empty antecedent set, such that it applies to every possible input. The consequent is set to the majority class of the unflagged training samples, i.e., the samples for which none of the learned rules applies. Therefore, even if no learned rule applies to an input sample, a class can still be predicted. This also makes it possible to use *default rule pruning* which substitutes learned rules by a default rule, if this leads to less misclassifications on the training data.

III. Case Study

In the case study, we investigate how well the ARL-based classification approach motivated in [6] performs in automotive integration- and system-level black-box regression testing setups. Therefore, we first introduce the study objects (Sec. III-A), then we discuss how we collected the data (Sec. III-B), and finally, we present the datasets (Sec. III-C).

A. Subject Systems

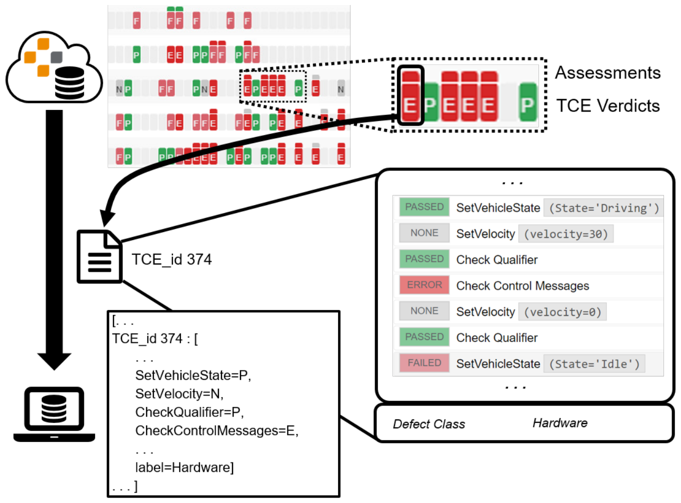
Data used throughout this work is provided by our industry partner. In the running V&V projects they are involved in, the SUTs are system-level customer-functions that deal with various car sub-systems such as the power train or the on-board supply system, e.g., automatic motor stop at traffic lights.

In such black-box settings, testers do not have access to any source code or models. On the contrary, they only have access to testing code and machinery, which is composed of various different software tools and components, i.e., real hardware as well as other software components. Hence, the only source of information are test cases and their history of execution, which is stored in a test report database.

In those projects the test experts' workflow envisages that test executions with a result other than *passed* need to be reviewed and manually assessed. This manual assessment (also referred to as *reassessment*) consists of diagnosing the failure cause and documenting it in a review comment plus assigning a so-called *defect class*, which is a project-specific failure category. Hence, we can directly use those defect classes as the labels for our classification task. That said, all necessary information for the application of the ARL-based failure classification approach are readily available in projects in this context.

Each test case can be considered as a test scenario, in which a sequence of test steps is executed to complete the scenario. In many cases, after the actual test scenario additional analysis steps are appended, which are also treated as test steps for our purposes. Consequently, the failure of any test step results in a failed TCE. In the projects (except D) on average there are between 35 and 45 test steps in each test case; for D there are on average even around 110 test steps.

In Fig. 1, a snippet from an exemplary TCE report is displayed consisting of the following actions encoded in test steps: Set the vehicle state to 'driving' and accelerate to 30 km/h. Then check a qualifier as well as some control messages. Afterwards come to a stop, again check the qualifier and set the vehicle state to 'idle'.



```
. . .
PASSED  SetVehicleState  (State='Driving')
NONE    SetVelocity  (velocity=30)
PASSED  Check Qualifier
ERROR   Check Control Messages
NONE    SetVelocity  (velocity=0)
PASSED  Check Qualifier
FAILED  SetVehicleState  (State='Idle')
. . .
```

```
[. . .
TCE_id 374 : [
    . . .
    SetVehicleState=P,
    SetVelocity=N,
    CheckQualifier=P,
    CheckControlMessages=E,
    . . .
    label=Hardware]
. . . ]
```

Fig. 1: Data collection and test step encoding

From the example, one can see that in our context a failed[1] test step does not necessarily terminate the running test execution. On the contrary, only in specific cases the execution is aborted. Thus, test cases regularly report multiple test step failures that may indicate the underlying problem. In fact, depending on the project between 30% up to 65% of all test steps are assigned verdicts other than $None$. Compared to other contexts, where only the TCE verdicts are available, we have more fine-grained information readily available.

B. Data Collection

The projects included in this study were not specifically selected. We only excluded projects from the study that did not offer the required information, namely the failure categories.The used test report database provides unique identifiers of the TCEs, test steps including their verdicts, and the manual assessments including the assigned defect classes.

The data collection is depicted in Fig. 1. In the test report database the history of the TCEs including their verdicts and assessments (if available) are stored. In each row the TCEs of the same test case are represented as colored tiles. The assessment (if available) is depicted as a bar above the corresponding TCE tile. As can be seen, not all TCEs have received a reassessment. For each TCE the individual test steps and their respective verdicts are available. For example, in Fig. 1 there are steps with verdicts $Passed$ (first time 'Set Vehicle State' and two times step 'Check Qualifier'), $None$ (the 'Set Velocity' steps), $Error$ ('Check Control Messages') and $Failed$ (second time 'Set Vehicle State'). Accordingly, this information is stored in a local database, where the test steps including their verdicts are encoded as test step items, complemented with the defect class label (cf. Sec. II-B).

C. Datasets

At the core of this study are five ongoing industry projects from our industry partner, where we collected data. As can be found in the upper part of Tab. I the datasets vary in terms of the number of test cases (from 0.2k to 2k) and TCEs. Also the portions of failed TCEs and the ones that we use in our study, the so-called *labeled TCEs*, vary significantly. Recall that the labeled TCEs are the failed TCEs, which the testers have reassessed and assigned a (meaningful)[2] defect class. In our datasets they range from 2k up to 40k over time spans of nine to 28 months. Despite the comparable time duration, the amount of data in terms of TCEs and labeled TCEs vary drastically. E.g., projects A and C have a very low ratio of labeled TCEs compared to the other projects.

As can be seen in the lower part of Tab. I, we group the defect classes into three categories: *code defects* (CD), *test infrastructure issues* (TI) and *other causes for failures* (OC). TI includes hardware and software failures, e.g., defective components, incorrect simulation models, and unstable testing software tools. OC includes configuration mistakes, test case implementation problems, incorrect or outdated test specifications, issues with the configuration of either the HiL or the test case, and other human-dependent failures.

The number of defect classes varies among the projects and range from four up to eight. As can be seen from the label distribution in Tab. I, the dataset is highly unbalanced and there are some dominant defect classes, e.g., the class 'Testcase' in projects B and D, while others are very rarely assigned, e.g., 'Config' in project B or 'Hardware-2' in project E. In summary, the portion of invalid test failures ranges from 77% up to 91% of all labeled failed TCEs.

IV. Experiment

As the approach used in this study is based on [6], we generally follow their experimental setup. For learning the association rules we use the *apriori algorithm*.In addition, we implemented rule pruning concepts for the ARL-based classification as described in Sec. II-C. Moreover, we adopt the default values for the minimum confidence and minimum support of 0.8 and 0.03 from [6]. We justify the choice of parameters and its influence on the classification performance in Sec. V-C.

A. Incremental Learning

We follow the incremental learning paradigm for backtesting [6], [10] with an initial training set with 10% of the available data. This means we simulate how the ARL-based classification approach would have performed if it had been in use during the time for which we have collected labeled data. Of course, it is important that at each simulated prediction in the past, the classifier only has access to data that would have actually been available at that time. To simulate this accurately, we use two types of timestamps: the execution-timestamps and the review-timestamps. We predict the defect class to aid the tester with failure diagnosis as soon as the test has been executed. Hence, a prediction is made at each

[1]Recall, we use the term *failed* and also mean verdict $Error$ here.

[2]Through discussions with experts we found that some defect classes are not meaningful to predict as they do not actually distinguish different failure categories but are rather used as catch-all oddments tray.

TABLE I: Summary of datasets and average classification performance in total and of each label, for each project.

	Project A				Project B				Project C				Project D				Project E			
Time span (months):				10				28				10				12				9
# Test Cases:				200				2.000				1.000				1.000				500
Avg. # test steps:				35				45				45				110				35
# TCEs:				136.000				1.500.000				3.000.000				30.000				100.000
# failed TCEs:				18.000				100.000				70.000				8.000				50.000
# labeled TCEs:				2.000				40.000				8.000				5.000				20.000
Performance:			Prec.	Rec.			Prec.	Rec.			Prec.	Rec.			Prec.	Rec.			Prec.	Rec.
	Total		.792	.360	Total		.867	.574	Total		.675	.321	Total		.653	.289	Total		.544	.320
Labels:	Label	freq. (%)			Label	freq. (%)			Label	freq. (%)			Label	freq. (%)			Label	freq. (%)		
CD:	CD	22.9	.925	.723	CD	13.4	.854	.488	CD-1	1.2	.623	.413	CD	21.1	.831	.453	CD	10.9	.359	.143
									CD-2	7.3	.591	.432								
TI:	Hardware	9.9	.434	.162	Hardware	3.5	.790	.671	Hardware	3.3	.213	.050	Hardware	18.5	.452	.162	Hardware-1	0.6	.000	.000
	Model-1	32.1	.788	.480	Model	23.9	.829	.589	Model	21.2	.611	.254	Tool	11.2	.510	.175	Hardware-2	1.4	.067	.012
	Model-2	13.7	.571	.084	Tool	5.8	.768	.132									Tool	2.1	.353	.111
	Tool	2.4	.000	.000													Model	43.5	.695	.297
																	Memory	1.2	.617	.407
OC:	Testcase	19.0	.574	.063	Testcase	46.3	.912	.636	Testcase	36.8	.701	.321	Testcase	49.2	.648	.293	Misc.	14.4	.377	.258
					Spec.	6.1	.734	.631	Config	30.2	.747	.371					Config	26.0	.523	.524
					Config.	0.9	.742	.190												

execution-timestamp, based on the data available up to that point.[3] Yet, the label (ground truth) of an execution only becomes available when its review-timestamp is reached. As a result, it can only be included in the training data when a tester has already manually checked the failed execution and submitted a review containing the defect class. The rationale behind this procedure is, that in the context of the studied V&V projects, generally, tests are executed over night and, if they fail, they are classified. However, only later, they are added to the labeled training data during the following day(s) once they have been reviewed.

Lastly, we can limit the training data at each given timestamp with a *time window*. In the backtesting simulation, instead of using all reviewed executions of the past up to the current point in time for training, only the most recent ones are used. The rationale behind the time window is that more data is not necessarily better as identified problems eventually get fixed over time and, hence, might lead to outdated training data. Since the test environment and test objects are both under development and constantly updated, old data may become useless or even detrimental for current predictions. We further elaborate on this rationale when reflecting on the observations made regarding the lifetime of association rules in Sec. V-B. In our experiments, a time window length of 60 | 90 | 120 | 60 | 30 days is used for project A | B | C | D | E respectively. In particular, in Sec. V-C, we discuss the influence of this parameter on the classification performance.

B. Evaluation Metrics

With the classifier we predict the defect class of failed TCEs. In order to evaluate how good the predictions made by the ARL-based classifier are, we calculate precision and recall, which allow for an intuitive interpretation. As such, precision indicates which proportion of performed predictions provide the correct defect class, and recall indicates the proportion of the actual defect classes correctly classified.

[3]Note that the classifier is simply retrained before each prediction, i.e., the association rules are learned on the current simulated training dataset.

An important difference to other classification techniques is that for our ARL-based classifier it is a valid output to not predict a defect class for a given TCE. For our purpose and context, we decide not to use default rules (cf. Sec. II-C), as in our context the classification is intended as a support tool and does not have the requirement that each input gets a prediction. It is considered more favorable to sometimes not get a defect class prediction rather than having a higher chance of getting an incorrect prediction. This is because we prioritize precision over recall. In the case of our studied projects, by not using default rules, on average, the precision is 10 percentage points (pp) higher and the recall 11 pp lower.

As we deal with multi-class classification, we calculate the performance per label following a one-vs-rest approach: We consider *true positives* (TP) as the predictions that have correctly assigned the ground truth label, *false positives* (FP) for falsely assigned labels, and *false negatives* (FN) for labels that are not predicted. The precision can then be calculated as $TP/(TP+FP)$ and the recall as $TP/(TP+FN)$ respectively. The overall performance is then the result of the total amount of predictions, i.e., we use the sum of TP, FP and FN over all labels to calculate the total precision and recall. As a consequence, the performance of more frequent labels have a higher impact on the overall performance.

V. Results

As discussed in Sec. IV, we run the backtest experiment for the five projects from our industry partner. In the following, we discuss how well the ARL-based classification performs and put the results into context. Therefore, in Sec. V-A we present our findings regarding the overall classification performance on all projects and have a look at the prediction performances for the individual class labels. In Sec. V-B, we discuss our observations regarding the lifetime of association rules. Afterwards, we report the influence of the parameters *minimum confidence* and *time window* in Sec. V-C, before we talk about the limitations of our study in Sec. V-D. Finally we compare the results to baseline classifiers in Sec. V-E.

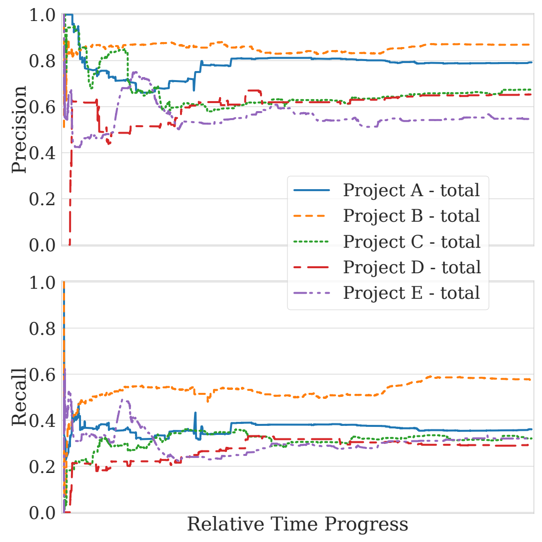


Fig. 2: Overall precision and recall values over the relative time progress for the different projects.

A. Overall Performance

The overall performance over time of the ARL-based classification is depicted in Fig. 2. Despite all the differences, the results appear similar to what has been reported in [6]. The exact values for the total precision and recall of all predictions for each project are listed in the lower part of Tab. I. Clearly, the results vary a lot between the projects. This seems to indicate that the performance of the approach depends on the project characteristics represented in the data, e.g., the structure of the implemented test cases or the reassessment practices including the choice of defect classes to differentiate.

The best precision (.867) and recall (.574) are achieved for project B. This is the project that uses the reassessment and in particular the defect class assignment for the longest time and, hence, the most data from the longest time span are available. However, since we use a similar time window to limit the training data in all projects, the larger dataset should not be the only reason for the better results. This is further shown by the other projects, which have varying amounts of data, yet a larger dataset size does not lead to better results: E.g., project A has the smallest dataset and the second best overall results.

The most noticeable trend for the results per defect class is that the performance seems to be correlated with the class label frequency in the datasets, shown in Tab. I. The predictions are (usually) better for the more frequent labels, and worse for the more uncommon labels. For the very infrequent defect classes 'Tool' in project A, and 'Hardware-1' in project E not even a single correct prediction was made.

Regarding the broader defect class categories, the CD predictions perform relatively good compared to the others considering the low relative frequency of occurrence in the datasets. One hypothesis for why that may be, is that the test cases are specifically constructed to test for SUT defects, and presumably fail at the expected test step(s). Failures caused by, e.g., sporadic test infrastructure errors, however, may be reflected unexpectedly in various different test steps.

Regarding the defect classes of the other two categories, TI and OC, the performance is mixed. The performance varies even within the projects. More specifically, model related failures seem to be reasonably well classified. One reason might be that problems with simulation models need to be fixed by the corresponding supplier and are only updated once in a while. If such problems only occur sporadically, they might be even tolerated without getting fixed, and can therefore be recognized reasonably well. In project E, e.g., the defect class 'Memory' captures such a frequently reoccurring sporadic failure that is tolerated and the classifier performs comparably well (prec. 0.617, rec. 0.407) despite the relatively low label frequency of 1.2%. In projects B, C and D this also seems to hold for issues related with the test cases themselves. In summary, in these cases the same problems occur frequently in various TCEs and are, thus, often predicted correctly.

B. Lifetime of association rules

For the different projects A | B | C | D | E a total of around 0.2k | 4k | 1.2k | 1k | 0.8k distinct rules have been learned over the course of the simulation experiments. Note, those are the numbers *after* pruning (cf. Sec. II-C). When not applying the pruning strategies the numbers are higher by a factor of roughly 250. At the same time for all projects except E the precision increased by around 2 pp whereas the recall decreased by between 1 to 3 pp. For project E the precision decreased by 2 pp and the recall dropped by 7 pp.

In fact, new rules are applied for the classification during the entire time. This means the set of association rules constantly changes in order to adapt to current developments in the testing infrastructure and SUT. On average across the projects, a single rule is used for only around 10 predictions, i.e., for about every 10th failed TCE a completely new rule is used.

For each prediction only one rule is used, however, multiple rules can be learned before each prediction. Yet, as can be seen in Fig. 3, only very few rules are even learned in more than 20% of the development weeks.

Another way to look at the transience of rules is depicted in Fig. 4, where the maximal difference between the first time and last time the same rule has been used for a prediction is shown for the different projects. Only very few rules surpass half the experiments' duration in that sense. For the rest, the usage is confined to a rather limited time period, already 91% | 88% | 89% | 95% | 94% of the rules are applied within a time-span of less than 10% of the experiments' duration.

In summary, the described transience of rules reflects what was to be expected: Throughout the progress of the V&V projects not only the SUT but also the test infrastructure evolves. Understandably the failure patterns change, which results in changing rule sets. Hence, we affirm the consideration that it is not advisable to come up with a set of static, handcrafted rules as they would most likely be valid for a short period of time only. However, as we did not specifically look

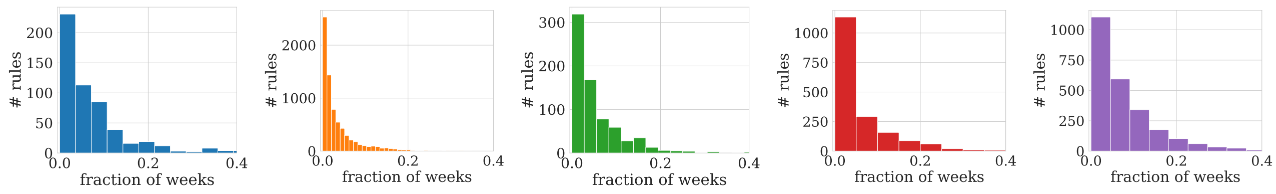


Fig. 3: Histogram showing the number of rules that are *learned* in different numbers of development weeks, as fractions of the total amount of weeks in the simulation time (for the different projects from left to right, A to E).

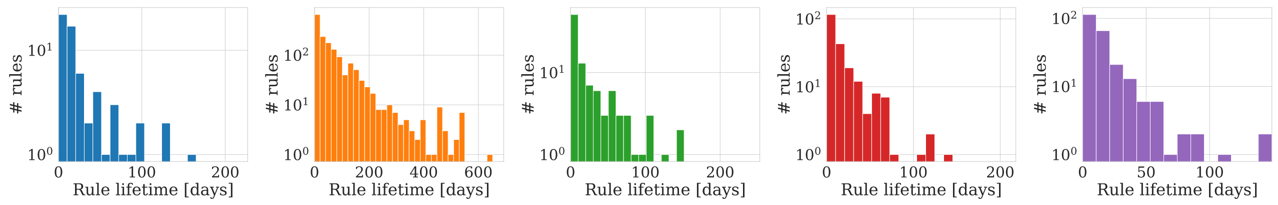


Fig. 4: Histogram showing the number of rules for different lifetimes, which is the maximum number of days between the first and last classification *usage* of the same rule (for the different projects from left to right, A to E).

into the rules or try to come up with the ‘best possible static rule set’, we cannot cogently reject the idea.

C. *Choice of Parameters*

In the experiments, we used fixed values for the minimum confidence, minimum support and time window. Regarding the minimum support, investigations showed that changing the minimum support parameter lead to very little change in the performance. By increasing the minimum support, the recall decreases slightly and for just two of the five projects, the precision increases insignificantly. Only when setting high values, e.g. 0.5 and higher, the recall drops substantially. For the minimum confidence and the time window, we performed a grid search to investigate their influence on the classification performance. The results for each project are compiled in Fig. 5, where for each confidence value a line is plotted showing the overall classification performance varying with the time window size.

As can be seen by the line color variation from the different minimum confidence values, this parameter shows a positive correlation with the precision and a negative correlation with the recall. This is not too surprising: As introduced in Sec. II, the minimum confidence specifies how often a rule has to correctly apply in the training data. Setting a high value leads to fewer included rules, increasing the odds of none applying to a new TCE. When no prediction can be made the recall decreases. With a lower minimum confidence value, more rules are considered, which might apply to the previously unclassified TCEs, increasing the recall. Yet, based on the training data, these new rules are also more likely to be incorrect, resulting in a lower precision. A minimum confidence of 0.5 may seem too low, as it includes rules that are incorrect half of the time in the training data. However, since the rules are sorted by highest confidence, the previous predictions made by high confidence rules remain unchanged. Thus, the average confidence of the applied rules is higher than 0.5. This also explains why changing the minimum confidence value causes a larger variation in the resulting recall than in the precision. Interestingly, the caused variation in precision and recall become larger with increasing time window size.

Regarding the time window parameter, the recall first improves with increasing window size and at around 60-90 days approaches saturation. This is also a good indication of how much initial data collection is necessary or appropriate before using the classification in practice, at least for the studied projects. For further increasing time window sizes, the recall stagnates or even declines. This effect is best visible for project B in Fig. 5. It is harder to make out a trend across the projects for the precision. In projects A and B, the precision decreases for increasing time window sizes. However, for projects C and D the precision is especially low for the smallest window sizes of 7 and 30 days. Similar to the recall, the time window effect on the precision mostly stagnates for values higher than 90 days. Except for project B, which may be explained by the fact that Bs dataset spans by far the longest time period. Consequently, a greater portion of the TCEs in the experiment were affected by the large time windows.

In summary, we argue that there is no general best minimum confidence value and the choice depends more on the dataset and the desired trade-off between higher recall or precision. It is up to the practitioners in the concrete setting to adjust the confidence parameter in order to emphasize higher precision (less wrong predictions) or recall (more predictions in general). Limiting the training data by a time window of 30-90 days works best in our experiments indicating that too much outdated training data deteriorates the classifiers performance. However, this observation may depend on the specific datasets and does not necessarily generalize.

D. *Limitations*

Our study shows that the ARL-based approach can be used to classify failed TCEs in running V&V industry projects from

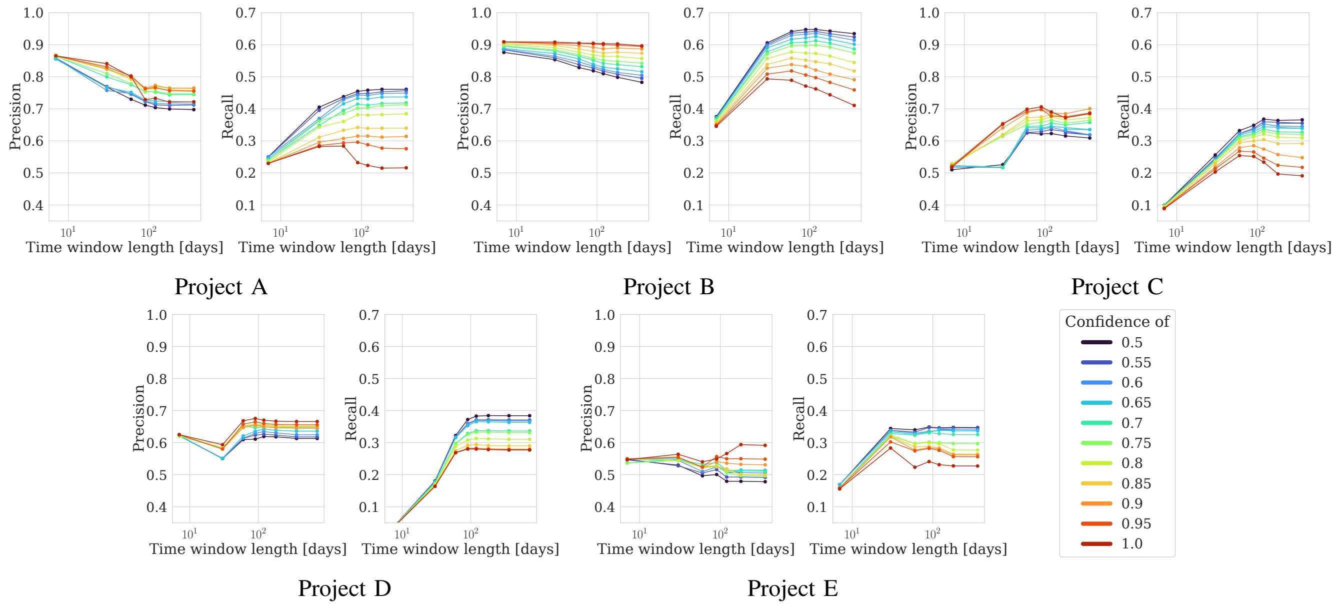


Fig. 5: Precision and recall plotted over the time window size in days for different minimum confidence values.

the automotive domain. Below, we point out some limitations and threats to validity of our study.

The research presented in this paper is based solely on data from a limited number of projects of one company. Nevertheless, as this company operates as a supplier in the automotive industry, the experts confirm that it is reasonable to expect similar data to be available in other settings as well.[4] Given our practical experience, we encourage practitioners to replicate this study in their projects.

We take the review comments and more specifically the assigned defect classes as the ground truth for our experiments, even though – for some cases – we have anecdotal evidence that even the experts do not always identify the *true* failure cause. Opposed to other studies, e.g., [10] that included information about incorrectly triaged bugs (*bug tossing*), unfortunately, no such information about potentially falsely assigned defect classes is available for the presented projects. According to the workflow, the failure category is associated manually to a TCE during inspection. Different experts might come to different conclusions and, hence, assign different labels. Another reason for such conflicting label assignments can be that in the same test execution multiple different failures manifest. Hence, it is possible that contradicting statements deteriorate the performance of the presented failure classification results.

In the datasets, we found TCEs that are identical in terms of the test step verdicts, however, have different defect class labels assigned to them. To assess the extent of this problem, we analyzed the datasets and identified all such groups of TCEs with identical step verdicts with more than one sample. In fact, we found for the projects A | B | C | D | E, for 65% | 76% | 75% | 35% | 79% of the respective TCEs at least one identical TCE has occurred before. Further, we found that out of those groups, around 50% | 30% | 21% | 41% | 18% have conflicting labels, meaning there are (at least two) identically appearing TCEs which are assigned different labels. However, less than 14% | 14% | 13% | 17% | 5% of all TCEs in such groups are assigned a label which is different from the majority label of their respective identical groups. Therefore, given that the confidence for association rules to be learned need not be 1.0[5], we argue that a classification approach, which is based on the test step verdicts, is still reasonable.

Nevertheless, it remains an open question to be discussed with the experts why different labels have been assigned to identically appearing test executions, whether those are justified and what additional information would be necessary to distinguish those cases. Some experts state that the specific test steps' *return values*, which may be different even for the same verdict, are determining factors for differentiation. However, these are not yet available to our classifier.

E. Baseline experiment

In fact, the findings regarding identically appearing TCEs with different class labels (Sec. V-D), motivated the implementation of two baseline experiments with naive classifiers *BL1* and *BL2* that we briefly discuss in the following: For the naive classifiers a prediction is made on the basis of an exact match of the test step verdicts. In cases where different labels have been assigned to identical executions – as discussed above – either the most recent label (BL1) or the majority label (BL2) is assigned. In fact, 22% | 14% | 15% | 50% | 13% of TCEs are unique, meaning their test step verdicts do not match any other execution in the corresponding projects A | B | C | D | E. Hence, the naive classifiers cannot predict a class for them.

[4]Recall that the approach was first applied to two Microsoft products [6] with a totally different tool chain.

[5]Recall, in our experiments we set the minimum confidence to 0.8.

TABLE II: Performance comparison with naive classifiers.

	Project A		Project B		Project C		Project D		Project E	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
ARL	.792	.360	.867	.574	.675	.321	.653	.289	.544	.320
BL1	.441	.155	.637	.406	.630	.211	.515	.115	.557	.083
BL2	.661	.233	.796	.508	.627	.210	.577	.129	.573	.086

The overall performance of the naive classifiers BL1 and BL2 is compiled in Tab. II.

As can be seen BL2 outperforms BL1 for all cases. BL2's precision is even competitive with the ARL classifier for the datasets of projects C and E, whereas the recall is not. As such, the experiments with those naive classifiers provide us an additional perspective on the dataset quality or, alternatively, the information richness encoded in the test step verdicts. If the different labels for two identically encoded TCEs were actually correct, additional information would be necessary to distinguish the two.

We have also performed experiments with many other classifiers including nearest neighbors, decision trees and random forests. Most perform comparably to the ARL-based classifier, meaning – not too surprisingly – the dataset quality is of much more importance than the employed classifier. For this work we confine ourselves to report only the results of the ARL classifier along the baseline classifiers because it triggered our investigations reported here.

VI. Lessons Learned

The practical implications of the gained insights are plenty. First, we learned that a high proportion of invalid test failures is encountered in the studied context. The practitioners assigned so-called defect classes to failed TCEs to distinguish the different types of failures regularly encountered. Second, as mentioned in Sec. V-D, we performed an analysis of the dataset quality. This quality assessment was a byproduct of our actual investigations that, nonetheless, lead to discussions with and among the test experts about how the failure analysis is performed and how the experts come to their defect class assignments. Especially, for the cases of identical test step verdicts with differing defect class assignments some differences regarding the interpretation could be revealed. However, for a portion of such cases, the different class labels are justified and the experts use information currently not encoded in the datasets to distinguish between the cases. Third, with our motivation to reduce the manual effort for the failure analysis, we got feedback from the practitioners that they would be motivated to adapt the test cases such that a test step-based classifier would potentially improve its performance.

Moreover, we can now answer the initially raised question: *How well can failed TCEs in automotive testing setups be classified according to their failure cause using test step-based ARL?* Depending on the project an overall precision between 54% and 87% can be achieved in the studied projects with recalls ranging from 29% up to 57%. The results achieved for projects A and B are comparable to the ones reported in [6] despite all differences between the two contexts. For some defect classes (e.g., (CD) in Project A, all classes in Project B) the approach works compellingly well, while for others (e.g., Tool or Model-2 in Project A, Tool or Testcase in Project E) additional investigations are necessary to determine whether there is a common cause for the unsatisfactory performance.

Further, we learned that the practical benefits of automatically classifying TCEs depends on the implications the class assignment can have: In some projects the defect class assignments were merely used for reporting while in other cases a subsequent process is triggered such as filing change request or bug tickets. In one of the projects the most commonly assigned defect class is not meaningful to predict, as this label is used when the reason for the failure is still unknown and needs further investigation. That is why we excluded it from the respective dataset for the experiments presented in this work.

Finally, in particular projects the infrastructure allows to automatically restart some components of the test infrastructure. Such a restart can be considered as a basic healing action that can be triggered during the nightly unsupervised automatic test case execution in case the classifier identified said component caused an invalid failed TCE. This leads us to the discussion of our planned future work.

A. Future work

Based on the limitations (Sec. V-D) and discussed lessons learned, we motivate future work to extend the ARL-based approach: First, we plan to incorporate the return values as additional information on test step level, as we have anecdotal evidence (cf. Sec. VI) that in some cases the return values are the determining factor for distinguishing defect classes. This information, however, is not readily available in our context. Thus, a technical challenge is to obtain the return values for all TCEs. In addition, it is not straight forward how to treat the return values in terms of items in our itemsets for ARL.

Based on the partially promising results, we motivate further investigations regarding how good a classifier needs to perform to be considered acceptable for the application in practice. Considering the achieved performance in projects B and A, we are confident that the presented ARL-based classification can be practically beneficial.

In some projects we came in touch with, reassessments are made but no defect classes are assigned. In fact, even in the studied projects, many failed TCEs have been reassessed without a defect class assignment, which made them unusable for this study. In such cases, one could assign defect classes based on the review comments, e.g. based on regular expressions. In fact, we experimented with that in one project but we do not report the results here, because we are aware of the bias of our own hand-crafted regular expressions-based label extraction, which also has implications on the comparability to the other projects where defect classes are inherently used.

VII. Related Work

A. Flaky tests

In the software engineering domain, tests are considered *flaky* if different results are obtained even though no changes have been made to the SUT or the test environment. Several works have investigated categories of causes for flaky tests in the software domain, e.g., [11]–[13]. One recent work extends this view by investigating intermittently failing tests in the embedded systems domain [14]. In our setting, the experts are aware of the unreliability of the test setup. However, due to the project circumstances the focus lies rather on coping with the unreliability than fixing it. That is why opposed to recent attempts to automatically root causing and even fixing flaky tests [15]–[17], we focus on classifying test executions according to their failure category.

B. Issue Assignment

Poulos and Veneris state that issue assignment is an "emerging need to appropriately categorize, prioritize and distribute [...] failures to the engineer(s) best-suited for detailed debugging of each failure" [18]. In fact, there are various recent works in the direction of automated issue assignment [10], [19], [20]. Note, we use *issue assignment* as a representative term, knowing that in the literature different terms are used, e.g., bug triaging [10], failure triaging [18], or, more generally, incident management [21]. This translates to the context of this work as to deciding which responsible person or team should analyze the prevalent failure situation brought up by a failing test execution. Those approaches predominantly use human written reports and employ natural language processing (NLP) techniques [10], whereas in our context we directly use the test results from the automated test execution.

Another line of research tries to make use of bug reports to predict root causes [22] or categorize the defects [23]. Those two works are conceptually closer to what we do in the sense that we also categorize reports according to (predefined) failure causes as we will discuss in the next subsection. However, we do not use human written reports or comments but only the test step verdicts and the human-assigned defect class.

C. Test Report Analysis

Another stream of papers deals with information obtained during testing. Liu et al., e.g., compare different so-called *fingerprinting* functions [24] including execution profiles. The central underlying rationale is that the same fault manifests in the same (or at least similar) behavior. This lead to approaches that try to cluster tests or rather their execution profiles to sort out groups of failures [25]–[27]. Also in [28], that idea is taken up to cluster failures in automotive HiL settings based on time-series. In contrast, in this work we do not cluster test executions but classify them. For that we do not use execution profiles but only readily available test execution information, namely, the test step verdicts.

Under the term *behavior learning* Bowring et al. propose an algorithm to label a program execution as either "pass" or "fail" [29]. In contrast, we do not need to learn to distinguish between passing and failing executions because we have test oracles for that. However, this idea of labeling program executions was further developed by Xie et al., who propose an approach to assign multiple labels to a program execution for cases where one failure should be attributed to one or more fault types [30]. The idea of multi-label classification was also empirically investigated under different application settings by Feng et al. [31]. In fact, there is ongoing discussion in the scientific community whether or not the independence assumption of component failures is appropriate and, hence, whether or not the consideration of multi-fault situations is necessary [32]. We report only on single-label classification in this work, because in our setting only a single defect class label is assigned to each TCEs.

Another type of classification is proposed by Hao et al. who distinguish the case where the cause of a test failure lies in the source code (bug) or in the test code (obsolete test) for Java programs, for which they use tailored features [33]. A similar question was also raised by Herzig and Nagappan, who distinguish bugs in the SUT from defects in the test infrastructure, which they call *false test alarms* [6]. In other words, both perform a binary classification task. We, however, need to distinguish between multiple failure causes, including the SUT, the test infrastructure and even the operator. In contrast to [8], who use NLP, we employ ARL inspired by [6]. In contrast to [6] however, we distinguish multiple different failure causes. To the best of our knowledge, we are the first to apply ARL to the complex and versatile setting of system-level testing of automotive embedded systems to categorize various different failure causes.

VIII. Conclusion

We identify invalid test case execution (TCE) failures and the necessary manual diagnosis and failure category attribution thereof as a major challenge that is particularly impacting integration- and system-level testing that involve hardware-in-the-loop test benches (HiLs). In the studied projects the portion of invalid test failures ranges from 77% up to 96% of all labeled failed TCEs. We found that little attention has been paid to this aspect in the literature so far, and methods to automatically distinguish between different test failure categories have considerable potential for time and cost reduction. We report the application of one approach taken from the software engineering domain, namely association rule learning (ARL)-based classification, and evaluate its performance in our context with five ongoing industry projects. Overall, we predict the failure category of more than 85k TCEs and achieve an overall precision up to 86.7% with an overall recall up to 57.4%.

Acknowledgment

This work was funded by the German Federal Ministry for Economic Affairs and Energy (BMWi), grant no. ZF4086014SO9. The responsibility for this article lies with the authors.

References

[1] J. Kasurinen, O. Taipale, and K. Smolander, "Analysis of Problems in Testing Practices," in *2009 16th Asia-Pacific Softw. Eng. Conf.* Batu Ferringhi, Malaysia: IEEE, dec 2009, pp. 309–315. [Online]. Available: http://ieeexplore.ieee.org/document/5358706/

[2] K. Wiklund, S. Eldh, D. Sundmark, and K. Lundqvist, "Impediments for software test automation: A systematic literature review," *Softw. Test. Verif. Reliab.*, vol. 27, no. 8, pp. 1–20, 2017.

[3] C. V. Jordan, F. Maurer, S. Lowenberg, and J. Provost, "Framework for Flexible, Adaptive Support of Test Management by Means of Software Agents," *IEEE Robot. Autom. Lett.*, vol. 4, no. 3, pp. 2754–2761, jul 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8719978/

[4] I. Evans, C. Porter, M. Micallef, and J. Harty, "Stuck in Limbo with Magical Solutions: The Testers' Lived Experiences of Tools and Automation," in *Proc. 15th Int. Jt. Conf. Comput. Vision, Imaging Comput. Graph. Theory Appl.* Valletta, Malta: SCITEPRESS - Science and Technology Publications, 2020, pp. 195–202. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009091801950202

[5] C. Jordan, P. Foth, A. Pretschner, and M. Fruth, "Unreliable Test Infrastructures in Automotive Testing Setups," in *2022 IEEE/ACM 44th Int. Conf. Softw. Eng. Softw. Eng. Pract.* IEEE, may 2022, pp. 307–308. [Online]. Available: https://ieeexplore.ieee.org/document/9793982/

[6] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms Using Association Rules," in *2015 IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2. Florence, Italy: IEEE, may 2015, pp. 39–48. [Online]. Available: http://ieeexplore.ieee.org/document/7202948/

[7] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. 1993 ACM SIGMOD Int. Conf. Manag. data - SIGMOD '93*. New York, New York, USA: ACM Press, 1993, pp. 207–216. [Online]. Available: http://portal.acm.org/citation.cfm?doid=170035.170072

[8] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing," in *2017 IEEE/ACM 39th Int. Conf. Softw. Eng.* Buenos Aires, Argentina: IEEE, may 2017, pp. 712–723. [Online]. Available: http://ieeexplore.ieee.org/document/7985707/

[9] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *KDD'98 Proc. Fourth Int. Conf. Knowl. Discov. Data Min.*, R. Agrawal and P. Stolorz, Eds., vol. 98. New York, NY, USA: AAAI Press, 1998, pp. 80–86.

[10] A. Sarkar, P. C. Rigby, and B. Bartalos, "Improving Bug Triaging with High Confidence Predictions at Ericsson," in *2019 IEEE Int. Conf. Softw. Maint. Evol.* Cleveland, OH, USA: IEEE, sep 2019, pp. 81–91. [Online]. Available: https://ieeexplore.ieee.org/document/8919115/

[11] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," *Proc. ACM SIGSOFT Symp. Found. Softw. Eng.*, vol. 16-21-Nove, pp. 643–653, 2014.

[12] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in *Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.* New York, NY, USA: ACM, aug 2019, pp. 830–840. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338945

[13] A. Ahmad, O. Leifler, and K. Sandahl, "Empirical analysis of practitioners' perceptions of test flakiness factors," *Softw. Testing, Verif. Reliab.*, vol. 31, no. 8, pp. 1–24, dec 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/stvr.1791

[14] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* New York, NY, USA: ACM, jul 2020, pp. 337–348. [Online]. Available: https://dl.acm.org/doi/10.1145/3395363.3397359

[15] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA 2019 - Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.* Beijing, China: Association for Computing Machinery, 2019, pp. 101–111.

[16] J. Morán, C. Augusto, A. Bertolino, C. D. L. Riva, and J. Tuya, "FlakyLoc: Flakiness Localization for Reliable Test Suites in Web Applications," *J. Web Eng.*, vol. 19, no. 2, pp. 267–296, jun 2020. [Online]. Available: https://journals.riverpublishers.com/index.php/JWE/article/view/3361

[17] C. Ziftci and D. Cavalcanti, "De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google," in *2020 IEEE Int. Conf. Softw. Maint. Evol.* Adelaide, SA, Australia: IEEE, sep 2020, pp. 736–745. [Online]. Available: https://ieeexplore.ieee.org/document/9240685/

[18] Z. Poulos and A. Veneris, "Clustering-based failure triage for RTL regression debugging," in *2014 Int. Test Conf.* Seattle, WA, USA: IEEE, oct 2014, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7035339/

[19] C. Bansal, S. Renganathan, A. Asudani, O. Midy, and M. Janakiraman, "DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng. Softw. Eng. Pract.* New York, NY, USA: ACM, jun 2020, pp. 201–210. [Online]. Available: https://dl.acm.org/doi/10.1145/3377813.3381353

[20] E. U. Aktas and C. Yilmaz, "Automated issue assignment: results and insights from an industrial case," *Empir. Softw. Eng.*, vol. 25, no. 5, pp. 3544–3589, sep 2020. [Online]. Available: https://link.springer.com/10.1007/s10664-020-09846-3

[21] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "An Empirical Investigation of Incident Triage for Online Service Systems," in *2019 IEEE/ACM 41st Int. Conf. Softw. Eng. Softw. Eng. Pract.* Montreal, QC, Canada: IEEE, may 2019, pp. 111–120. [Online]. Available: https://ieeexplore.ieee.org/document/8804464/

[22] T. Hirsch and B. Hofer, "Root cause prediction based on bug reports," in *2020 IEEE Int. Symp. Softw. Reliab. Eng. Work.* Coimbra, Portugal: IEEE, oct 2020, pp. 171–176. [Online]. Available: https://ieeexplore.ieee.org/document/9307647/

[23] F. Thung, X.-B. D. Le, and D. Lo, "Active Semi-supervised Defect Categorization," in *2015 IEEE 23rd Int. Conf. Progr. Compr.*, vol. 2015-Augus. Florence, Italy: IEEE, may 2015, pp. 60–70. [Online]. Available: http://ieeexplore.ieee.org/document/7181433/

[24] C. Liu, X. Zhang, and J. Han, "A Systematic Study of Failure Proximity," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 826–843, nov 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4589219/

[25] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun, and Bin Wang, "Automated support for classifying software failure reports," in *25th Int. Conf. Softw. Eng. 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, pp. 465–475. [Online]. Available: http://ieeexplore.ieee.org/document/1201224/

[26] N. DiGiuseppe and J. A. Jones, "Concept-based failure clustering," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng. - FSE '12*. New York, New York, USA: ACM Press, 2012, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2393596.2393629

[27] M. Golagha, A. Pretschner, D. Fisch, and R. Nagy, "Reducing failure analysis time: an industrial evaluation," in *2017 IEEE/ACM 39th Int. Conf. Softw. Eng. Softw. Eng. Pract. Track*. Buenos Aires, Argentina: IEEE, may 2017, pp. 293–302. [Online]. Available: http://ieeexplore.ieee.org/document/7965453/

[28] C. V. Jordan, F. Hauer, P. Foth, and A. Pretschner, "Time-Series-Based Clustering for Failure Analysis in Hardware-in-the-Loop Setups: An Automotive Case Study," in *2020 IEEE Int. Symp. Softw. Reliab. Eng. Work.* Coimbra, Portugal: IEEE, oct 2020, pp. 67–72. [Online]. Available: https://ieeexplore.ieee.org/document/9307664/

[29] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Proc. 2004 ACM SIGSOFT Int. Symp. Softw. Test. Anal. - ISSTA '04*. New York, New York, USA: ACM Press, 2004, p. 195. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1007512.1007539

[30] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *2014 Softw. Evol. Week - IEEE Conf. Softw. Maintenance, Reengineering, Reverse Eng.* Antwerp, Belgium: IEEE, feb 2014, pp. 134–143. [Online]. Available: http://ieeexplore.ieee.org/document/6747163/

[31] Y. Feng, J. Jones, Z. Chen, and C. Fang, "An Empirical Study on Software Failure Classification with Multi-label and Problem-Transformation Techniques," in *2018 IEEE 11th Int. Conf. Softw. Testing, Verif. Valid.* Västerås, Sweden: IEEE, apr 2018, pp. 320–330. [Online]. Available: https://ieeexplore.ieee.org/document/8367059/

[32] J. Li, X. Yan, B. Liu, and S. Wang, "An insight of double-faults interactions in program: An empirical study," in *2017 Second Int. Conf. Reliab. Syst. Eng.*, no. Icrse. Beijing, China: IEEE, jul 2017, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/8030752/

[33] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, "Is this a bug or an obsolete test?" *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7920 LNCS, pp. 602–628, 2013.