Empirically Detecting False Test Alarms Using Association Rules

Kim Herzig
Microsoft Research
Cambridge, United Kingdom
kimh@microsoft.com

Nachiappan Nagappan
Microsoft Research
Redmond, United States
nachin@microsoft.com

***Abstract*—Applying code changes to software systems and testing these code changes can be a complex task that involves many different types of software testing strategies, e.g. system and integration tests. However, not all test failures reported during code integration are hinting towards code defects. Testing large systems such as the Microsoft Windows operating system requires complex test infrastructures, which may lead to test failures caused by faulty tests and test infrastructure issues. Such *false test alarms* are particular annoying as they raise engineer attention and require manual inspection without providing any benefit. The goal of this work is to use empirical data to minimize the number of false test alarms reported during system and integration testing. To achieve this goal, we use association rule learning to identify patterns among failing test steps that are typically for false test alarms and can be used to automatically classify them. A successful classification of false test alarms is particularly valuable for product teams as manual test failure inspection is an expensive and time-consuming process that not only costs engineering time and money but also slows down product development. We evaluating our approach on system and integration tests executed during *Windows 8.1* and Microsoft *Dynamics AX* development. Performing more than 10,000 classifications for each product, our model shows a mean precision between 0.85 and 0.90 predicting between 34% and 48% of all false test alarms.**

Keywords—software testing; association rules; false test alarms; classification model; test improvement

I. Introduction

Every day engineers change software systems applying code changes to add new features, improve the product or to fix known issues. However, code changes increase the risk of introducing new issues or incompatibilities. To ensure that code changes do not lower product quality, developers typically test their code changes before merging them into the current code base. Testing all code changes applied to a code base for large software systems may be in itself a time-consuming task. While test cases on unit level might run fast, higher-level tests, such as system and integration tests, usually take more time to complete. Thus, the more changes are applied the more tests need to be executed and the more time each code change spends in verification before beig integrated into the final product. At the same time, competition between software manufacturers to gain or defend market share increases the pressure on development teams to increase efficiency while maintaining or even increasing product quality. This situation is not new and there exist many studies and reports that investigate a wide variety of code and process metrics to estimate code quality before release [1, 2, 3, 4, 5, 6]. However, the increasing pressure on development teams to deliver features faster also impacts software testing processes. Testing processes tend to reduce code integration speed (*code velocity*) and slow down product development. Therefore, optimizing test processes is likely to positively affect speed and productivity of overall software development processes.

There exist a wide range of test optimization research fields that address various aspects of testing processes, e.g. test prioritization [7, 8], test selection [9, 10], test generation [11, 12], test effectiveness measurements [13, 14, 15], etc. However, many of these techniques concentrate on unit testing, which mainly focuses on functional correctness. In this paper, we tackle a severe problem that mainly appears in the area of system and integration testing—tests that typically check for constraints such compatibility, performance, privacy, etc. In theory, test cases either pass or fail and if they fail, they hint to code defects. In practice, running system and integrations tests for systems, e.g. Microsoft Windows or the Microsoft Dynamics business software suite, require complex test setups and infrastructures, which come with their own issues. Thus, system and integration tests may also fail due to test and infrastructure issues, e.g. broken hardware prevents a test from retrieving a remote file. We call such test failures *false test alarms*. *As* any test failure, false test alarms are reported to the engineers requiring manual investigation lowering development speed. However, false test alarms provide no insights into product quality but rather harm the development process. Therefore, it is desirable to minimize or eliminate false test alarms, or at least preventing them to disrupt the development process[1]. At the same time, test failures due to code defects must remain enabled and may not be ignored as these test failures may prevent code defects to be shipped to the customer.

The goal of this work is to develop a precise *false test alarm classification model*, which identifies false test alarms automatically. Our model analyzes reported and manually classified false test alarms. Using association rule mining, we detect frequently occurring patterns between failures of

[1] False test alarms are valuable to identify problems in verification processes that should be resolved. However, false test alarms should not slow down code integration processes.

individual test steps that are unique for false test alarms. These mined association rules are then used to automatically classify newly reported test failures as false test alarms. We evaluated our approach on a large set of false test alarms reported by system and integration test cases during development periods of *Windows* and *Dynamics*.

We make the following contributions in this paper:

- We use association rule mining to analyze tens of millions of individual test steps to detect patterns between these test steps that are unique to false test alarms.
- Using these test behavior patterns, we develop a fully automatic and continuously learning model to pre-classify test case failures as false test alarms.
- We evaluate our classification model on more than 20,000 test case results executed during development periods of Windows and Dynamics.
- We further estimate the impact of false test alarms on code velocity that could have been prevented using the proposed test result classification model.
- We briefly discuss how to apply such a model in a live development process scenario.

The paper is structured as follows: In Section II and Section III we provide background on related work and a short introduction into the analyzed testing processes. In Section IV we discuss how we mined and analyzed test results and false test alarms using association rule mining, before presenting our experimental setup in Section V. In Section VI we present our evaluation results and discuss the impact of this work on the development processes in Section VII. We close with threats to validity in Section VIII and a conclude in Section IX.

II. Related Work

Classifying test results is an active research field, but most of the related studies focused on failure classification, clustering test failures with respect to failure causes, test repair, and fault localization. To the best of our knowledge, only the work presented by Hao et al. [16] and Herzig and Nagappan [15] classify test failures related to test and infrastructure issues rather than code defects.

A. *Classifying Program Failures And Behavior*

A wide range of studies classifying the type of program behavior and in particular program failures exists. In many cases, test failures are described by bug reports. Using bug reports to classifying program failures is an active research field. Guo et al. [17] presented an approach to predict which program failures get fixed. Similar to the results presented by Bettenburg et al. [18], their results suggest that the reputation of bug reporters as well as the details of test failure description determines the likelihood of bugs to get fixed. Zanetti et al. [19] used social networks to build a successful classification model to identify bug reports that refer to actual code defects and that add no duplicated bug description. Antoniol et al. [20] used text mining to separate bug reports from feature requests. More generally, approaches as presented by Sherwood et al. [21] and Bowring et al. [22] automatically classify program behavior using execution data. In contrast, the work presented in this paper, uses test step failure patterns to automatically classify whether test failures report code defects or are due to test and infrastructure issues.

B. *Fault Localization*

Studies on fault localization tackle the problem of identifying code parts that are likely to cause a test failure. Although related, studies on fault localization deal with a more complex problem of automatic fault cause analysis to enhance and speedup debugging sessions. Hildebrandt and Zeller [23] showed that a binary search on program input causing faulty program behavior can be used to minimize program input reproducing the error and thus to narrow down code areas containing the corresponding code bug. Jones and Harrold [24] compared different fault localization techniques and showed that automatic fault localization techniques can be very precise. Later, Liu and Han [25] used proposed a new type of fault proximity not only making fault localization more precise but that can also be used to cluster failing traces with respect to the fault cause. Lately, Zhou et al. [26] mined bug reports extracting additional information hinting to possible bug locations.

C. *Test Repair*

In cases in which test failures are not due to code defects but rather to test issues, researchers proposed techniques to automatically repair tests—a scenario that occurs in practice [27]. Instead of simply reordering test executions to repair partially broken test cases [28], later studies advanced and tried to actually fix the suspected broken test case automatically [29, 30, 31] also for GUI based test cases [32, 33]. The work presented in this paper does not try to automatically fix test issues nor to prevent their execution. Instead, our goal is to classify false test alarms as such, to reduce manual inspection effort, and to speed up the testing and development processes.

D. *Failure Clustering*

Similar to Liu and Han [25], there exist more approaches to classify software failures. The goal of these studies is to group test or program failures based on their suspected cause. Although closely related to this work, these approachs do not specifically target false test alarms but rather group failures often categorizing them by similarity but often without interpreting the relevance of the failures. Dickson et al. [34] used machine learning over program executions to identify faulty program executions. Later, Podgurski et al. [35] used clustering techniques to identify faulty program executions that are likely to be caused by the same code defect. DiGiuseppe and Jones used "latent-semantic-analysis techniques to categorize each failure by the semantic concepts that are expressed in the executed source code" [36] while Francis et al. [37] proposed tree-based classification techniques to cluster program failures with respect to their underlying error.

E. *Classifying Test Results*

Related work specifically targeting test results with the goal of identifying test results to filter relevant test failures is rare. Triou et al. [38] filed a patent to collect, compare, and cross-reference test failure results. Hao et al. [16] and Herzig and Nagappan [15] reported similar studies in which the authors

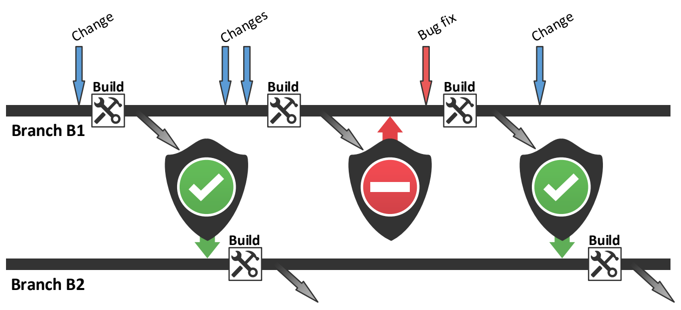


Fig. 1. Code changes have to pass system and integration tests to get integrated into lower level branches. Failing tests automatically cancel code integrations and require manual inspection and possible bug fixes to allow further code integration.

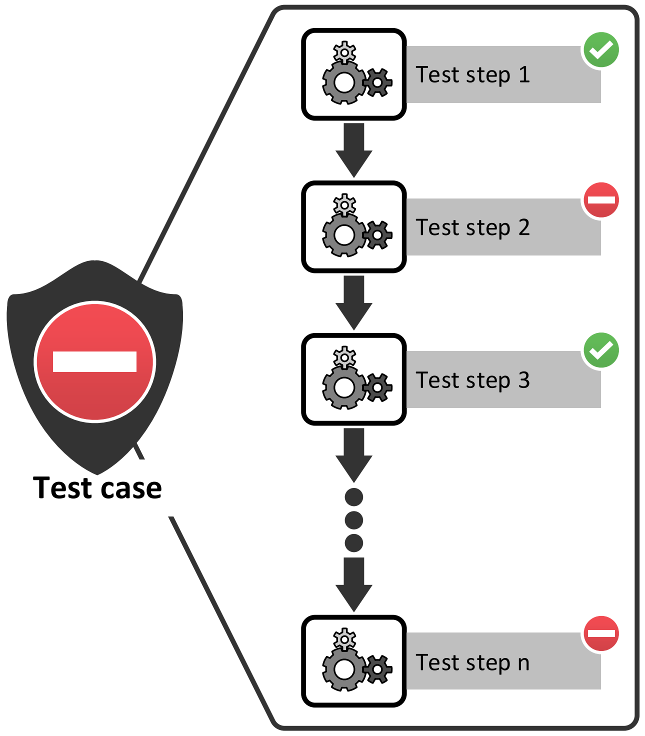


Fig. 2. Each system and integration test case as shown in Fig. 1 executes a sequence of test steps all of which are required to pass to pass the overall test case. Test failures reported to engineers contain a list of test steps that failed during execution.

classified test results to identify test failures due to test issues. However, the study by Hao et al. [16] is based on unit test level and uses test complexity and program execution measurements (e.g. code coverage) to classify test results. In comparison, the work presented in this paper does not use any static nor dynamic test measurements. Collecting such information for test executions is rather expensive in a large evolving system and slows down test speed. The presented classification model is solely based on already existing test step results captured during common test executions without the need to slow down or enhance existing test environments in any way. Closest to this approach, is the study by Herzig and Nagappan [15] who identified false test alarms in the Microsoft Windows development process and showed to show the impact of organizational structure on test reliability and effectiveness. In their study, false tests alarms are used as reliability measures for system and integration tests. In this paper, we re-use the approach presented by Herzig and Nagappan [15] to identify false test alarms.

III. Test Process

In this study, we investigate system and integration test runs continuously executed during the development of Microsoft *Windows* and Microsoft *Dynamics*. Each system and integration test case checks for one or multiple system constraints such as compatibility, performance, privacy, functional correctness etc. Where system constraints exist on products, additional test infrastructure is required to ensure all code meets those constraints. Since product constraints are system properties, they often need to be verified at system level. For example, Windows has certain backward compatibility requirements, both in terms of hardware and in terms of supported applications. To verify these constraints requires the emulation of millions of different configurations and execution setups. For the sake of brevity, we provide a high-level description of the analyzed Microsoft testing processes to make this paper self-contained. For details, we refer to Bird and Zimmermann [39] and Herzig and Nagappan [15].

The development process to develop and maintain large software systems typically involves multiple code branches—a forked copy of the code base that allows parallel modifications without interference (for more details we refer to Bird and Zimmermann [39] and Murphy et al. [40]). Typically, code changes are applied in development branches and once ready integrated into the trunk branch using integration branches. Each merge between branches is guarded by system and integration test cases ensuring basic functionality and constraints such as compatibility and performance compliances (see Fig. 1). Thus, each code change has to pass multiple layers of system and integration tests while code changes from different development branches are merged together. Once a code change reaches the trunk branch it is considered as part of the next release.

Each system and integration test case can be considered a test scenario executing a sequence of test steps to complete the scenario. To sucessfully complete the scanrio (test case), all test steps must pass (see Fig. 2). As a consequence, each failing test step causes the corresponding test case to report a test failure.

A. Test Failures

A failing test case causes a development process disruption. Scheduled code integration requests are canceled and the corresponding code branch on which the test failure occurred is excluded from code integration processes until the issue is resolved. Each failed test case requires manual inspection and resolution in order to include the branch code, and its code branch sub-tree, into the code integration process again. As a consequence, each system and integration test failure not only affects the engineers that submitted code changes to the branch before the test failure, but all engineers that will have to merge their code changes through this code branch in order to integrate into the main trunk branch. Please note that a failing test step may not cause the test case to terminate immediately. Thus, each executed test case may report more than one test step failure each of which may relate to a code defect or a false test alarm.

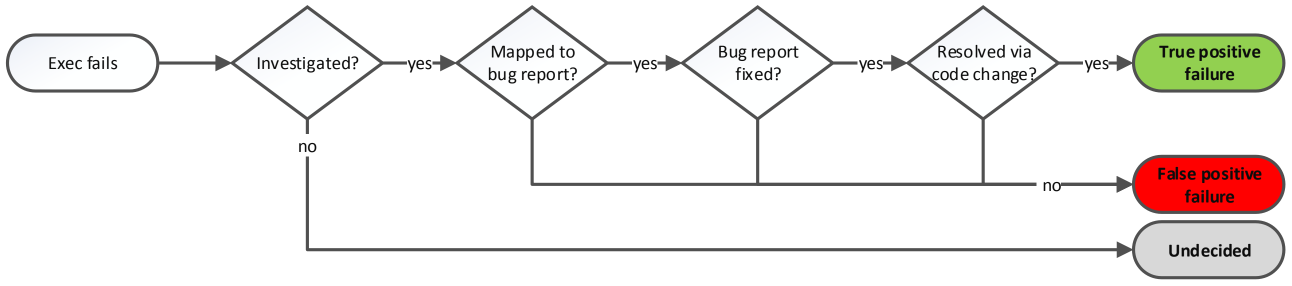


Fig. 3 Flow chart describing the process to separate test failures reporting code issues from test executions failing due to other reasons than code issues (e.g. test and infrastructure issues).

Test case failures reported to engineers contain a list of test steps that failed to help the engineer to investigate the failure cause and to resolve the underlying issue.

B. *False Test Alarms*

Test results presented to engineers are classified as passing or failing. However, it is important to further distinguish whether a test failures is caused by a code defects or whether the test failures must be considered a false test alarms. A test failure that is due to any other reason than a code defect is regarded as false alarms. In most cases, such false alarms are caused by test and infrastructure issues, e.g. a test case requires to fetch an input source from a remote server that cannot be reached at the time of the test execution. False test alarms are a common issue during system and integration tests, for example testing the installation of a Windows operating system.

False test alarms are expensive and harm the verification and development process without providing any benefit. Like other test failures, false test alarms require expensive manual inspection. However, unlike test failures due to code defects, investigating false test alarms must be considered as a waste of time and resources. The result of the investigation will be that the test failure was due to test and infrastructure issues, but allows no conclusion about the actual code quality under tests. The test suite execution must be repeated, once the test infrastructure issue is resolved. Like for any other test failure, the code branch is banned from code integrations until the tests pass again. This is likely to affect other engineers on the same branch as they are also banned from integrating changes into the main trunk branch. Thus, false test alarms not only waste the time of engineers inspecting the test failure but also slows down productivity and code velocity of entire development teams.

IV. Data Collection

The main goal of this work is to build a precise classification model to identify false test alarms without requiring expensive additional information about test runs, such as dynamic program traces or state dumps.

A. *Test Case Features (Independent Variables)*

Instead, we investigate the behavior of individual test steps to judge the outcome of the overall test case. The rational is that false test alarms show specific patterns or combinations of test step failures that rarely occur during normal test executions including test failures due to code defects. For each failing test case, we collect the following properties of all executed (failing and passing) test steps executed:

- The unique identifier of the test case execution. Each test case execution is assigned a unique identifier that can be used to reference a specific executed instance of a test case.
- The unique identifier of the test case, typically the test case name. However, this identifier does not specify the exact execution (see test case execution identifier above). A test is typically associated with many executions.
- The identifier of the executed test step. Each test case is a sequence of test steps. Test steps themselves have unique names within a test case. The full-qualified test step name is a combination of test case name and test step name. It allows us to uniquely refer to an individual test step and is unique among all test steps names.
- A simple binary field indicating whether the test step has passed or failed. This binary field contains no indication on whether the test case step failed due to a code defect or a test or infrastructure issue.

At this point in time, we made no assumption on test failures or their possible causes. In particular, we do not make any judgment on whether the test case or the individual test step failed due to code or test and infrastructure issues.

B. *False Test Alarms (Dependent Variable)*

To identify false test alarms, we trace development activities that occurred after a test failure (see Fig. 3) using CODEMINE [1]. Test failures referencing bug reports that were fixed by applying code changes must be considered test failures due to code defects. Test failures that did not lead to a bug report or that were assigned to bug reports, which never got fixed, are considered false test alarms. The only exception are test failures that were not investigated at all—we ignored these instances and removed them from our list of observed test failures. The mapping strategy was developed in cooperation with the Windows and Dynamics product teams. We estimated the number of falsely classified test failures to be below 5%.

C. *Test Step Association Rules*

To discover patterns among test step behavior unique to false test alarms, we use *association rule learning* [41] to produce rules of the form: $\{a_1, \ldots, a_n\} \Rightarrow \{c\}$ where left hand side of the implication (*antecedent*) represents one or multiple conditions that need to be satisfied to imply the right hand side (*consequent)*. In our case, the set of antecedents $a_1, \dots, a_n$ will indicate which combination of test step results is expected in order to indicate the type of test failure reported by the test case. As an example, consider the following rule:

$$\{TestStep_X = 1, TestStep_Y = 0, TestStep_Z = 1\} \Rightarrow FTA.$$

This association rule suggests that a test case execution in which test steps X and Z fail but test step Y passes should be considered a false test alarm. Typically, association rule learning returns more than a single association rule. Each rule can be treated as a separate set of conditions that if satisfied by a test case execution indicate how to interpret the corresponding test case result. Note that the antecedents of an association rules are not sufficient to let the consequence to become true. Association rules do not state implications but probabilistic relationships. As a consequence, association rules are associated with statistical measurements: *support* and *confidence*. Translated to our usage scenario, support is a value between zero and one and defined as the proportion of test case executions for which all antecedents were satisfied. A support value of 0.5 would mean that 50% of all observed test case executions satisfied all antecedents. In the example above this would mean that in 50% of all test case executions, test steps X and Z fail while test step Y passes. The confidence in a rule is defined as the relative number of observed test case executions for which all antecedents and the consequence were satisfied over the number of test case executions for which all antecedents were satisfied:

$$confidence(\{A \Rightarrow \{c\}\} = \frac{support(A)}{support(A \cup \{c\})},$$

where A represents a set of antecedent such as $\{a_1, \dots, a_n\}$. Confidence values range between zero and one. A confidence value of 1 indicates that in all cases for which the antecedents were satisfied the consequence could always be satisfied as well. Note that different association rules might contradict each other. It is important to remove contradicting rules from rule sets before using them. As result, for a series of observed test case executions, we extract a set of association rules expressing probabilistic relations between test step results and the overall test case failure categorization. Each rule is associated with a support and confidence value that allows us to filter rules based on their frequency and accuracy.

V. Experimental Setup

A. Learning Association Rules

To perform association rule learning on a given set of test case executions, we used an implementation of the *apriori algorithm* provided by Hahsler et al. [42] in their *arules* package for the statistical framework R [43]. For each given set of observations (transactions), we use a stringent selection criteria for association rules that we consider as relevant. Association rules must be associated with a minimum confidence value of 0.8 before being considered by our classification model. Additionally, we considered only rules that appeared in at least 3% of all test case failures. This minimum support value is derived by measuring the median number of occurrences per test case in the overall set of test failure observations and multiplied by ten.

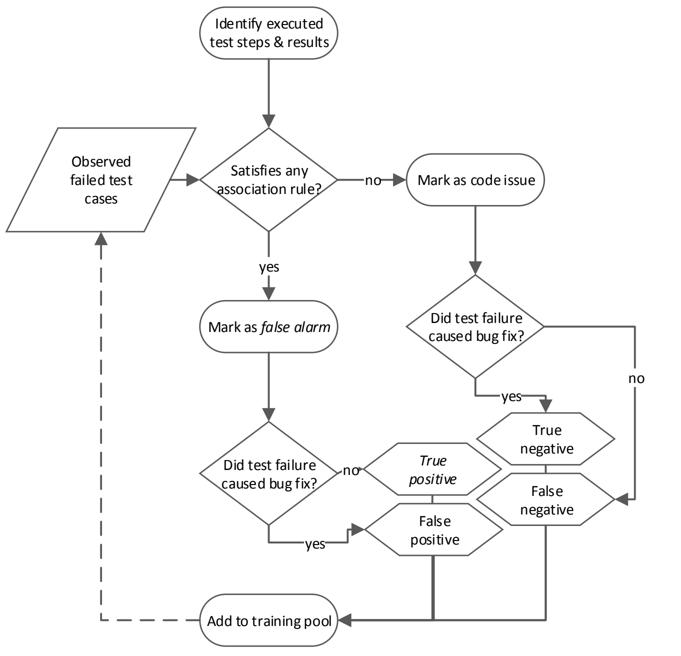

Identify executed test steps & results
Observed failed test cases
Satisfies any association rule?
no
Mark as code issue
yes
Mark as false alarm
Did test failure caused bug fix?
yes
no
True negative
Did test failure caused bug fix?
no
True positive
False negative
yes
False positive
Add to training pool

Fig. 4. Using incremental learning to evaluate false test alarm classification model. Evaluation performed on multiple million test case executions.

For each set of observations, we split the set of association rules into two subsets: one containing rules whose consequences indicate false test alarms (FP) and the other subset containing rules whose consequences indicate failures due to code defect (TP). To remove possible contradicting rules, we only use FP rules whose antecedents (left hand side) does not appear as antecedents in the set of TP rules.

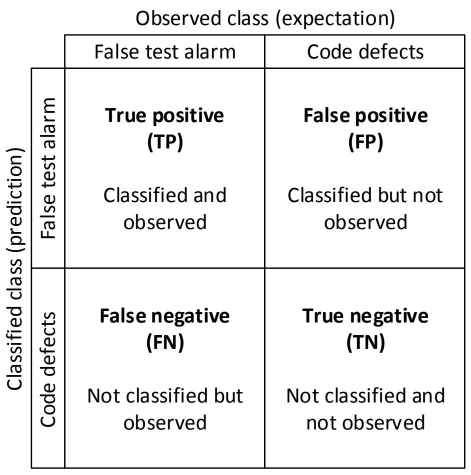
B. Predicting and Updating Classification Model

To simulate realistic scenarios, we use *incremental learning* to classify test case failures as false alarms based on previous test case execution observations (see Fig. 4). We start with an initial training set containing the first 10% of test case failures as they occurred during development (preserving temporal order). The idea is to build up a set of association rules as basis for any classification attempt. After this initial training phase, we proceed with the following steps:

Step 1: We fetch the next test case failure as it occurred during development and decompose the failure into individual test step results.

Step 2: We check if our current pool of association rules contains any rule whose left hand side (antecedent) is satisfied by the test step results observed during test case execution. If any such rule exists, we classify the test case failure as false test alarm. If no such rule exist (considering the thresholds discussed in Section V.A), we consider the test failure to be due to code defects.

Step 3: We compare the classification result with the actual ground truth by tracing development activities that occurred after a test failure (see Section IV.B).



		Observed class (expectation)	
		False test alarm	Code defects
Classified class (prediction)	False test alarm	**True positive (TP)** Classified and observed	**False positive (FP)** Classified but not observed
	Code defects	**False negative (FN)** Not classified but observed	**True negative (TN)** Not classified and not observed

Fig. 5. Comparing observed and classified test failures in a confusion matrix. Used to compute precision and recall values to measure accuracy of classification model.

Step 4: Depending on the result of this comparison, we mark the result either as *true positive* (we correctly predicted the test failure to be a false alarm), *false positive* (we predicted the test failure to be a false alarm but it was due to code defects), *false negative* (we failed to classified the test failure as false test alarm), or *true negative* (we correctly classified the test failure to be due to code defects). See Fig. 5 for schematic version of confusion matrix.

Step 5: We use the ground truth as new observation and use the updated pool of test case observations to create a new set of association rules.

Naturally, all test case executions that did not fail are treated as *true negatives*. To measure the accuracy of our classification model, we report precision and recall values over all predictions performed.

C. Study Subjects

To evaluate our classification technique, we classified integration test cases executed during development periods of *Windows* and *Dynamics*. For each product, we classified more than 10,000 test case failures executing tens of millions of test steps. TABLE I. contains details about the development periods and products our experiments were conducted on. For *Dynamics* we covered a total development period of approximately 2 years, for *Windows*, active development lasted about 1 year. Overall, we performed more than 10,000 predictions per product.

VI. Identifying False Test Alarms

As discussed in Section V, we used incremental learning to conduct our experiments.

A. Precision & Recall

The overall observed precision for *Dynamics* lies at 0.85 while for *Windows* we achieved an overall precision of 0.9 (see TABLE I.). Thus, the false positive rate—classified false test alarms that were due to real code defects—lies under 15%. This is important as false positives can have critical impact on product quality: test failures that would be falsely suppressed or ignored could lead to defects elapsing quality assurance and thus directly affect product quality.

TABLE I. Overall precision and recall values.

	Dynamics	Windows
Covered development time	~2 years	~ 1year
Overall precision	0.85	0.90
Overall recall	0.48	0.34

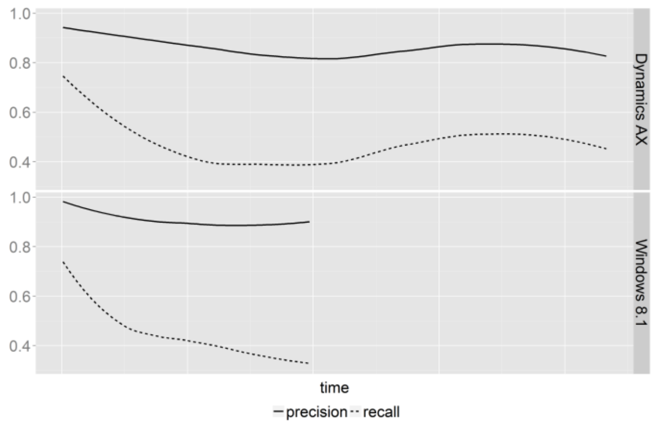


Fig. 6. Precision and recall values of Windows false test alarm classification model over time. X-axis represent test case failures ordered by time without initial training phase.

As expected, recall values for our classification models are lower. At the end of each development period, the overall observed recall value for *Dynamics* lies at 0.48 and for *Windows* at 0.34. The reason for these low recall values is that there exist a lot of false test alarms cause by infrastructure issues that occurred only a few times or even only once. Thus, low recall was expected and shows that false test alarms are more complex than simple infrastructure issues, but rather are serious issues that are hard to detect and prevent. Still, reducing the number of false test alarm reported by at least 34% is a significant improvement. We discuss implication on development processes in more detail in Section VII.

Fig. 6 shows values for cumulative precision and recall values over test case failures ordered by time. Values on the x-axis represent development time excluding the initial training phase (see Section V). As shown in Fig. 6, precision and recall values are not constant but rather follow a wave curve. In both cases, the recall value drops quite dramatically over time. For *Windows*, the cumulative recall value at the end of the one-year period drops to its lowest value of 0.34. The same trend can be observed for *Dynamics* for which the recall value recovers after a sharp drop in the first half of the development time (one year) before it starts declining again. Currently, we suspect that there could be a relationship between the decrease (and increase) of recall values and the maturity of the code base towards final release dates. However, we have no evidence to confirm this relationship. Although showing similar behavior, precision values are much more stable over time. The curve shape is less distinct but still visible. For both products, precision values do not drop under 0.8 and remain high throughout the entire experiment. This is an important factor as precision is the more important value. False positives (precision) threaten the usefulness and reliability of the presented approach, while low

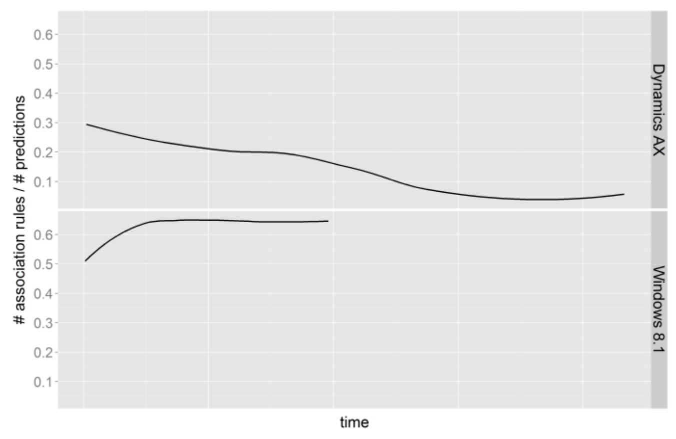


Fig. 7. Number of used association rules relative to the number of predictions performed plotted over time. X-axis values represent test case failures ordered by time without initial training phase.

recall values (false negatives) impact the ability to prevent false test alarms. However, the number of false negative classifications does not threaten product or process quality as these false test failures occurred already (not newly introduced) but could not be automatically detected.

B. *Number of Assocation Rules*

Besides precision and recall, we were interested in the number of association rules required to achieve the high precision as discussed in the previous section. In particular, we wanted to know whether the number of association rules is rather constant—no new rules must be learned over time—or whether constant learning of new rules is required. In the first case, we might be able to extract the learned rules into a static, and faster, classification model, or whether we need to keep learning new appearing rules. For this purpose, we relate the number of matching association rules per predicted test case failure with the overall number of performed predictions. This relative measure represents the number of rules to be learned relative to the number of predictions. The result is a value between 0 and 1, where 1 represents cases for which each prediction instance would require a different association rule to be learned. For the case that the number of required association rules remains constant over time, we should see a linear dropping line, when plotting this value over time.

Fig. 7 shows this relative association rule count measurement over time (predicted test case failures ordered by time). Both products show different trends. While *Dynamics* is a nearly constantly dropping function, the number of association rules required to achieve the mean precision value over 0.8 for this product seems to be limited, thus, few new rules must be learned over time. For *Dynamics*, there exist two time windows in which the number of association rules is increasing over time (curve constant or increasing). Comparing these two time windows with the curve in Fig. 7, we see that the precision and recall values for these two windows is also dropping, suggesting that new rules must be learned to gain previous precision. However, new rules are only adapted over time when these rules have proven to be reliable (support and confidence thresholds, see Section V.A). For *Windows*, the curve for the number of association rules relative to the number of performed predictions is completely different. Here, the curve is increasing in the first

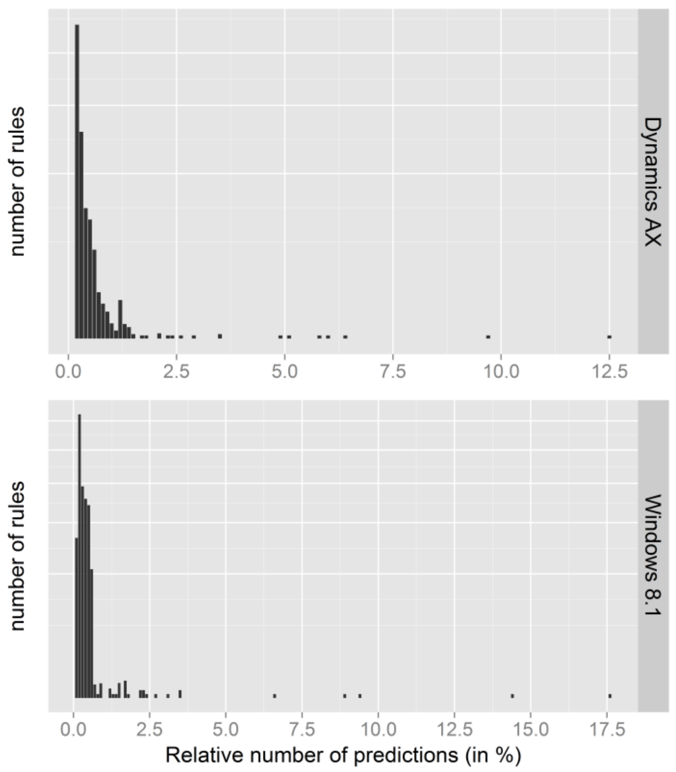


Fig. 8. Relative number of rule occurrences per development week: How often does a specific rule occurs across the entire development period? A value of one means a rule is omnipresent across all development days—having a minimum confidence value of 0.8.

couple of weeks indicating that a high number of new rules have to be learned. Then, after some time, the curve stabilizes at a level of roughly 0.65. This means, that for *Windows*, the set of association rules required to maintain a mean precision value of 0.91 is constantly changing and new rules have to be learned.

C. *Lifetime of associtation rules*

The results shown in Section VI.B suggest that individual association rules seem to have a rather short lifetime and change quickly to capture new types of test and infrastructure issues, at least for *Windows*. To further investigate this trend of association rules being valid for only short periods of time, we investigated the duration of individual association rules as relative number of occurrence across predictions—that is the number of predictions an association rule caused the test case failure to be classified as false test alarm. Fig. 8 plots the distribution of relative occurrence as a histogram. The x-axis represents the relative number of predictions a single association rule shows with a confidence value above 0.8 and a support value above 3% (see Section V). The y-axis represents the number of rules that showed this relative number of occurrences, using a square root scaling function for the y-axis.

From Fig. 8 we can see that most association rules occur contribute to less than 1% of all classifications; remember that we classified more than 10,000 test case failures per product, thus, a rule contributing to 1% of all classifications still contributed to at least 100 decisions. Only very few association rules contribute to more than the 2.5% of decisions. For

Dynamics the most prominent rule contributes to 12.5% of all decisions. Similarly, the most prominent rule for *Windows* contributes to 17.5% of decisions. In general, the data shown in Fig. 8 supports the observation that association rules have a relative short lifetime (as suspected in Section VI.B). Consequently, the number of required association rules is high, indicating that false test alarms can manifest in many different ways, but still be consistent over time. This result supports the need for an automated classification system. A high number of frequently changing patterns that can point out false tests alarms is required. In fact, the actual number of required association rules to achieve our presented precision lies well above 100. Demanding engineers to check for more than 100 patterns whenever a test case fails seems to be unpractical and too time consuming, even if the number of rules would remain constant or change little over time.

VII. Impact on Development Process

Applying an automated classification model in real development scenarios can have severe consequences and implications with respect to development processes, engineering behavior, and product quality. In this section, we discus some of these possible implications and provide some estimations on them. However, most of these improvements are hard or even impossible to measure. Nevertheless, such implications are important to understand the implications of models like the one presented in this paper.

A. *Code Velocity*

As discussed in Section III.B, false test alarms directly affect development speed. Failing tests cause code branches to be banned from code integrations until the test issue is resolved. For false test alarms, the time required to inspect the test failure the code branch is blocked unnecessary, causing delays for code changes going through this code branch. Preventing false test alarms to block the integration activity of a code branch or being raised to engineers could reduce these unnecessary delays and thus improve development speed. To estimate the benefit of our false test alarm classification model with respect to code velocity, we traced the resolution time of false test alarms for all cases that our classification model had classified correctly (true positives). Summing these time values represents the amount of time code branches were banned from integration activity, which might have caused integration delays, but which could have prevented using a classification model like the one presented in this paper.

For *Dynamics* the estimated code velocity gain is roughly 173 hours, which corresponds to little more than 7 days. Normalized over two years of development time, this corresponds to an average development speedup of 14 minutes per day. The same calculation for *Windows* results in a total gain of 611 hours or 25.5 days. Normalized over a development period of 1 year, this corresponds to an average daily code velocity gain of roughly 100 minutes per day (1.7 hours per day).

Please keep in mind that this estimation is a very rough estimation and that it assumes that the classification model would have the power to suppress classified false test alarms, a highly unlikely scenario. Note that our current classification model has a low recall value of 0.3, which implies that around 70% of false test alarms would remain undetected. Thus, the potential gain in code velocity could be significant higher.

B. *Impact on Engineers*

The main motivation for this work is to provide help in identifying false test alarms, as these test failures require manual failure inspection. Raising less false test alarms should help to increase the confidence in test results themselves, but also in decisions based on test results. Increasing code velocity, as discussed in Section VII.A, can reduce the number of merge conflicts and thus might further reduce the number of actual test failures.

C. *Impact on Product Quality*

Using the proposed classification model would imply test process changes. Like any other process change, the classification model requires monitoring the implications and possible effects on other development processes. The precision of our classification model is already high, but still produces a small fraction of false positives---test failures due to code defects classified as false test alarms---which may cause code defects to remain undetected for some time. Although we suspect this time to be short. Tests are executed in short time intervals or get triggered by code integration requests. However, using human supervision and monitoring techniques to ensure high development process quality are recommended, not only for this classification model, but for all changes to development processes.

D. *How to Use Such a Classifcation Model*

The model is an excellent tool to help engineers to prioritize test failures and to provide additional input for engineers to confirm the classification models decisions. Such a scenario has two important benefits. First, it would reduce the risk of code defects wrongly classified as false alarms to a minimum. Instead of suppressing the test failure, the failure still reaches the engineer but warns her about the possibility of being a false test alarm. It would allow human supervision of the classification system and may include a feedback loop that allows engineers to override classification results which will then help to train the classification model. At the same time, such an interactive model might help engineers to prioritize their test failure inspection. Test failures classified as false test alarms could be seen as low severity failures and ranked by their corresponding support and confidence values, similar to the eRose system suggested and implemented by Zimmermann et al. [44].

Together with the product teams within Microsoft, we believe that tackling the issue of false test alarms will strengthen confidence into testing infrastructure even if some test cases occasionally report false test alarms.

VIII. Threats To Validity

A. *Generalizability*

We investigate test executions and test results for two Microsoft products and their development processes. Even though some terminology might be unique to Microsoft, the execution of tests during software development, the impact of test execution time on development speed, and the issue of false test alarms are generalizable. We also believe that the basic assumption that test failures due to test and infrastructure issues will manifest in different test step failure patterns is general and not specific to a particular development or verification process. The data collected to build the classification model were collected using Microsoft specific tools like CODEMINE, but are not requiring the used toolset nor relies on Microsoft specific data or details. We collect test execution data, test failure statistics, as well as bug report linked to code changes used by many other studies conducted on open-source. Hence, we believe that the study and its presented model is generic and could be easily replicated on other projects in different companies.

The impact of suppressing or pre-classifying false test alarms on the overall development process might differ and are of course product and company specific. The estimated code velocity measurements presented in this paper are specific to Microsoft. Replicating the experiments on different projects, releases, or development processes requires detailed reviews and applied heuristics and might yield different results.

This paper focuses on system and integration tests that test various constraints of a system, e.g. performance, compatibility, and functionality. However, different test types, such as unit tests, might require different approaches or might not even require false test alarm detection.

B. Construct Validity

In this paper, we identify false test alarms mapping various data sources kept in different databases. Although, we discussed our heuristic, data mappings, and data interpretations with all involved product teams, it is possible that some data mappings might be missing or wrong. The product teams and we consider all made approximations as fair and realistic and that assumptions made in this study reflect the development processes accurately. Time factors used to estimate code velocity delays are based on average Microsoft figures and numbers. These numbers vary over time and might not consider all possible aspects.

The classification models presented in this paper depends on data extracted using CODEMINE [1], the approach to identify false test alarms by Herzig and Nagappan [15], and on the apriori algorithm implemented in the *arules* [42] package for the R statistical framework. All threads to validity to these tools also apply for this study.

IX. Conclusion

In this paper, we presented a novel approach to automatically classify false test alarms, test case failures due to test and infrastructure issues. Our classification model uses association rule mining to identify patterns across failing test steps that correlate with false test alarms. We evaluated our classification model on two development processes and periods covering the development of Microsoft Windows and a 2-year development period of Microsoft Dynamics. Our classification model shows a mean precision above 0.85 for both products. This means that our classification model produces a false positive rate below 15%. Recall values for both products are low and range between 0.3 and 0.5. Having a low recall does not imply negative consequences for the development process as these test failures already occurred in reality but simply could not be prevented. We estimate the achieved code velocity increase by preventing classified false test alarms to block code integration activity to 14 minutes per day for Dynamics and 100 minutes per day for Windows. The product teams are confident that the presented approach can help to positively impact the development processes by increasing the confidence into test executions and test results, but also by speeding up development processes.

The technique and results described in this paper have convinced the Microsoft Windows product team to explore ways to integrate the presented classification model into their production test environments. The goal is to rank and mark classified test failures accordingly to raise awareness of potential false test failures.

Acknowledgment

We thank the Windows and Dynamics product teams for their tremendous support and feedback. This work is based on data extracted from varies development repositories provided by the Microsoft TSE group. Our special thanks go to Brendan Murphy, Michaela Greiler, Jacek Czerwonka, Jason Means, Rama Durairajan, Josh Swedberg, and Adrian Marius Marin for their great support and invaluable insights into development processes and datasets.

X. References

[1] J. Czerwonka, N. Nagappan, W. Schulte and B. Murphy, "CODEMINE: Building a Software Development Data Analytics Platform at Microsoft," *Software, IEEE,* vol. 30, no. 4, pp. 64--71, 2013.

[2] K. Herzig and A. Zeller, "Mining cause-effect-chains from version histories," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, 2011.

[3] K. S. Herzig, "Capturing the long-term impact of changes," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010.

[4] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 2010.

[5] N. Nagappan, B. Murphy and V. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study," in *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[6] S. McIntosh, Y. Kamei, B. Adams and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects," in *In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*, Hyderabad (India), 2014.

[7] G. Rothermel, R. Untch, C. Chu and M. Harrold, "Test case prioritization: an empirical study," in *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, 1999.

[8] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, 2002.

[9] S. Fujiwara, G. v.Bochmann, F. Khendek, M. Amalou and A. Ghedamsi, "Test selection based on finite state models," *Software Engineering, IEEE Transactions on,* vol. 17, pp. 591-603, Jun 1991.

[10] G. Rothermel and M. J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Softw. Eng. Methodol.,* vol. 6, pp. 173--210, apr 1997.

[11] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[12] V. Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller, "Generating Test Cases for Specification Mining," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.

[13] M. Perscheid, D. Cassou and R. Hirschfeld, "Test Quality Feedback Improving Effectivity and Efficiency of Unit Testing," in *Creating, Connecting and Collaborating through Computing (C5), 2012 10th International Conference on*, 2012.

[14] S. Zeltyn, P. Tarr, M. Cantor, R. Delmonico, S. Kannegala, M. Keren, A. P. Kumar and S. Wasserkrug, "Improving Efficiency in Software Maintenance," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011.

[15] K. Herzig and N. Nagappan, "The Impact of Test Ownership and Team Structure on the Reliability and Effectiveness of Quality Test Runs," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014.

[16] D. Hao, T. Lan, H. Zhang, C. Guo and L. Zhang, "Is This a Bug or an Obsolete Test?," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013.

[17] P. J. Guo, T. Zimmermann, N. Nagappan and B. Murphy, "Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, 2010.

[18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj and T. Zimmermann, "What Makes a Good Bug Report?," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[19] M. S. Zanetti, I. Scholtes, C. J. Tessone and F. Schweitzer, "Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[20] G. Antoniol, K. Ayari, M. Penta, F. Khomh and G. Yann-Gaël, "Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, 2008.

[21] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *SIGOPS Oper. Syst. Rev.,* vol. 36, pp. 45--57, oct 2002.

[22] J. F. Bowring, J. M. Rehg and M. J. Harrold, "Active Learning for Automatic Classification of Software Behavior," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2004.

[23] R. Hildebrandt and A. Zeller, "Simplifying Failure-inducing Input," *SIGSOFT Softw. Eng. Notes,* vol. 25, pp. 135--145, aug 2000.

[24] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[25] C. Liu and J. Han, "Failure Proximity: A Fault Localization-based Approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2006.

[26] J. Zhou, H. Zhang and D. Lo, "Where Should the Bugs Be Fixed? - More Accurate Information Retrieval-based Bug Localization Based on Bug Reports," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[27] L. S. Pinto, S. Sinha and A. Orso, "Understanding Myths and Realities of Test-suite Evolution," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[28] M. Galli, M. Lanza, O. Nierstrasz and R. Wuyts, "Ordering broken unit tests for focused debugging," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004.

[29] B. Daniel, V. Jagannath, D. Dig and D. Marinov, "ReAssert: Suggesting Repairs for Broken Unit Tests," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, 2009.

[30] B. Daniel, T. Gvero and D. Marinov, "On Test Repair Using Symbolic Execution," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.

[31] G. Yang, S. Khurshid and M. Kim, "Specification-Based Test Repair Using a Lightweight Formal Method," vol. 7436, D. Giannakopoulou and D. Méry, Eds., Springer Berlin Heidelberg, pp. 455-470.

[32] M. Harman and N. Alshahwan, "Automated Session Data Repair for Web Application Regression Testing," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*, 2008.

[33] A. M. Memon, "Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing," *ACM Trans. Softw. Eng. Methodol.,* vol. 18, pp. 4:1--4:36, nov 2008.

[34] W. Dickinson, D. Leon and A. Podgurski, "Pursuing Failure: The Distribution of Program Failures in a Profile Space," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2001.

[35] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003.

[36] N. DiGiuseppe and J. A. Jones, "Concept-based Failure Clustering," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[37] P. Francis, D. Leon, M. Minch and A. Podgurski, "Tree-based methods for classifying software failures," in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, 2004.

[38] E. Triou, A. Milbradt, O. Agbonile and A. Dar, "Systems and methods for automated classification and analysis of large volumes of test result data," Google Patents, 2009.

[39] C. Bird and T. Zimmermann, "Assessing the Value of Branches with What-if Analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, Cary, North Carolina, 2012.

[40] B. Murphy, J. Czerwonka and L. Williams, "Branching Taxonomy," Microsoft Research, Cambridge, 2014.

[41] R. Agrawal, T. Imieli\'nski and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases," *SIGMOD Rec.,* vol. 22, pp. 207--216, jun 1993.

[42] M. Hahsler, S. Chelluboina, K. Hornik and C. Buchta, "The Arules R-Package Ecosystem: Analyzing Interesting Patterns from Large Transaction Data Sets," *J. Mach. Learn. Res.,* vol. 12, pp. 2021--2025, jul 2011.

[43] R. D. C. Team, "R: A Language and Environment for Statistical Computing," 2010.

[44] T. Zimmermann, V. Dallmeier, K. Halachev and A. Zeller, "eROSE: Guiding Programmers in Eclipse," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2005.