2018 IEEE International Conference on Software Quality, Reliability and Security Companion

Towards a Bayesian Network Model for Predicting Flaky Automated Tests

Tariq M. King[†§], Dionny Santiago[†§], Justin Phillips[†] and Peter J. Clarke[§]

[†]*Quality and Performance Architecture*
Ultimate Software Group, Inc., Weston, Florida 33326, USA
E-mail: quality@ultimatesoftware.com

[§]*School of Computing and Information Sciences*
Florida International University, Miami Florida 33199, USA
E-mail: {tking003, dsant005, clarkep}@cis.fiu.edu

***Abstract*—Artificial intelligence and machine learning are making it possible for computers to diagnose some medical diseases more accurately than doctors. Such systems analyze millions of patient records and make generalizations to diagnose new patients. A key challenge is determining whether a patient's symptoms are attributed to a known disease or other factors. Software testers face a similar problem when troubleshooting automation failures. They investigate questions like: Is a given failure due to a defect, environmental issue, or flaky test? Flaky tests exhibit both passing and failing results although neither the code nor test has changed. Maintaining flaky tests is costly, especially in large-scale software projects. In this paper, we present an approach that leverages Bayesian networks for classifying and predicting flaky tests. Our approach views the test flakiness problem as a disease by specifying its symptoms and possible causes. Preliminary results from a case study suggest the approach is feasible.**

***Keywords*-testing, automation, machine learning, flaky, model**

I. Introduction

Developing enterprise software systems involves a large-scale testing effort where hundreds, if not thousands of automated test scripts are run continuously. Engineering teams rely on these tests to verify that code changes have not introduced new errors into previously tested components, a practice known as automated regression testing [2]. The rise of DevOps and emphasis on continuous deployment has led to an increasing reliance on automated regression testing for making decisions on whether the system is ready to ship.

Maintaining automated test scripts at scale can be costly, especially if they become slow and unstable – a problem referred to as *test flakiness* [8], [17], [25]. In an article on flaky tests, a senior manager at Google Research states [17]: "Productivity for developers at Google relies on the ability of the [automated] tests to find real problems with the code being changed or developed in a timely and reliable fashion [...]. What we find in practice is that about 84% of the transitions we observe from pass to fail involve a flaky test!" Addressing test flakiness in continuous integration pipelines therefore has the potential to significantly improve the efficiency of development teams.

Researchers in academia and industry have been investigating how advances in artificial intelligence (AI) and machine learning (ML) can be leveraged for software testing [9], [20], [22]. Work in this field can be divided into three categories: (1) *AI for Testing*: developing AI systems to test software or enhance the testing process, (2) *Testing AI*: devising methods to test systems that use AI to perform tasks, and (3) *Self-Testing*: designing autonomous systems with the intrinsic ability to test at runtime [9]. The research that will be presented in this paper fits into the category of AI for testing. It proposes a supervised ML approach that uses Bayesian networks to predict flaky automated tests.

Bayesian networks are well-suited for efficient representation and reasoning with uncertain knowledge [23]. A key factor to consider when using Bayesian networks in practice is the need to accurately define the causal relationships between variables, and provide evidence to support claims of causation. This is because correlation between two variables does not imply that one causes the other [6]. Our research hypothesis is that defining and validating a set of causal factors for test failures and flakiness, and training a Bayesian network based on those factors, will facilitate building a practically useful test flakiness classifier. If successful, the classifier could be incorporated into continuous integration and deployment pipelines to prevent new flaky automated tests from being introduced.

The major contributions of this paper are: (1) presents a novel approach that models the test flakiness problem as a *disease* with certain *symptoms* to facilitate the identification of their *causes*; (2) defines a set of metrics to support data collection when applying the approach, including a new metric for measuring the stability of web page element selectors; and (3) shares the results of a case study that investigates the feasibility of applying the approach to a cloud-based human capital management solution.

The rest of this paper is organized as follows: the next section contains a literature review. Section III presents our approach. Section IV provides the supporting test metrics. Section V defines the flaky test classifier. Section VI describes the case study and Section VII concludes the paper.

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00031

II. Literature Review

This section contains background material to aid understanding of the paper, and describes related work.

A. Bayesian Network Machine Learning

Machine learning (ML) is a subset of AI that involves the construction of algorithms that can *learn* from and make predictions on data [12]. Formally stated, if the performance ***P*** of a computer program at completing a task ***T*** improves with experience ***E***, the program is said to have learnt [19], [23]. In this definition, the experience ***E*** typically refers to data points in the form of a set of examples. ML can be supervised, meaning that the learning algorithm is trained using labeled data with known outcomes, or unsupervised where data is unlabeled and the goal is to discover new data patterns and relationships. ML approaches include decision tree learning, Bayesian networks, artificial neural networks, reinforcement learning, and more [12], [23].

A Bayesian network, also known as Bayesian belief network, is a directed acyclic graph in which each node is associated with a conditional probability distribution [12]. Conditional probability is the likelihood of an event happening, given that it has some relationship to one or more other events [6]. For example, the probability of finding a parking space is dependent on the location being visited, the day of the week, and time of day.

Formally, a Bayesian network model is defined by: (1) a set of nodes where each node corresponds to a random variable, which may be discrete or continuous; (2) a set of directed edges that connect pairs of nodes, where an edge from node *X* to node *Y* indicates that *Y* is dependent on *X*; and (3) a set of annotations in the form of conditional probability tables (CPTs), where each CPT represents a conditional probability distribution $P\left(X_i|Parents(X_i)\right)$ that quantifies the effect of the parents on node X_i.

B. Automated Software Testing

Software testing is "the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the expected behavior" [24]. Testing can occur at multiple levels that map to stages in the software development process [2]. Unit testing verifies the behavior of the basic building blocks of the software during implementation. Integration testing checks the interactions between the components that make up the architectural design. System testing validates the software as a whole to make sure that it is acceptable to the customer [2], [24]. This includes validating that the software meets functional and non-functional requirements.

Where possible, software practitioners leverage tools to automate the manual aspects of software testing. The current state of automated software testing is primarily focused on *scripting* – the process of encoding tests in a format that can be interpreted by a class or utility program called a *test driver* [10], [16]. Test drivers automatically execute the encoded actions in test scripts against the system, and produce a log of the test results. Test script automation has been heavily studied in the literature and there are a number of recommended best practices on the subject [13], [16].

C. Related Work

Test flakiness, its impacts, and possible fixes for non-deterministic tests have been studied in the literature [4], [11], [17], [25]. However, to the best of our knowledge, there are no approaches that attempt to build a test flakiness prediction model and integrate it into deployment pipelines.

The work by Gao [4] is closely related to our work. It proposes an entropy-based metric for measuring the flakiness of test outputs such as UI state, code coverage, and invariants. Gao [4] focuses on factors external to the test such as initial application state, system platform, Java version, and tool harness configurations. Our research encompasses internal and external causal factors for test flakiness, and accounts for factors related to observable outcomes.

Lou et al. [11] presents an empirical study that analyzes source code commits that likely fix flaky tests in open-source projects. Similar to our research, their approach classifies common root causes for flaky tests. They describe techniques that can be used to force flaky behavior to manifest, e.g., modifying timing parameters, object state comparison. Such checks could be used to enhance our test flakiness estimates.

Micco [17] describes techniques that range from manual inspection for preventing flakiness, to the use of monitoring tools for automatically quarantining flaky tests. However, no details are provided for how the latter could be implemented. Our approach describes how tooling like this can be implemented using machine learning.

Applying Bayesian networks to software testing is not a new concept. Rodriguez et al. [22] present a systematic literature review that describes the use of Bayesian concepts for testing effort and fault prediction, test data generation and planning, and GUI testing. They found that the main use of Bayesian networks in software testing lies within the software reliability area. However, none of the works identified investigate the reliability of the tests themselves.

III. Test Flakiness: An Automation Disease

A flaky test may be defined as a test that exhibits both a passing and failing result with the same code [4], [8], and is therefore deemed unstable and unreliable. Our approach views the test flakiness problem as a disease so that we can model its observable impacts as *symptoms*, and then determine any causal factors for those symptoms and the test flakiness disease itself. This section presents our overall approach to developing a prediction model for test flakiness, and describes its disease-symptom relationships along with the proposed causal factors.

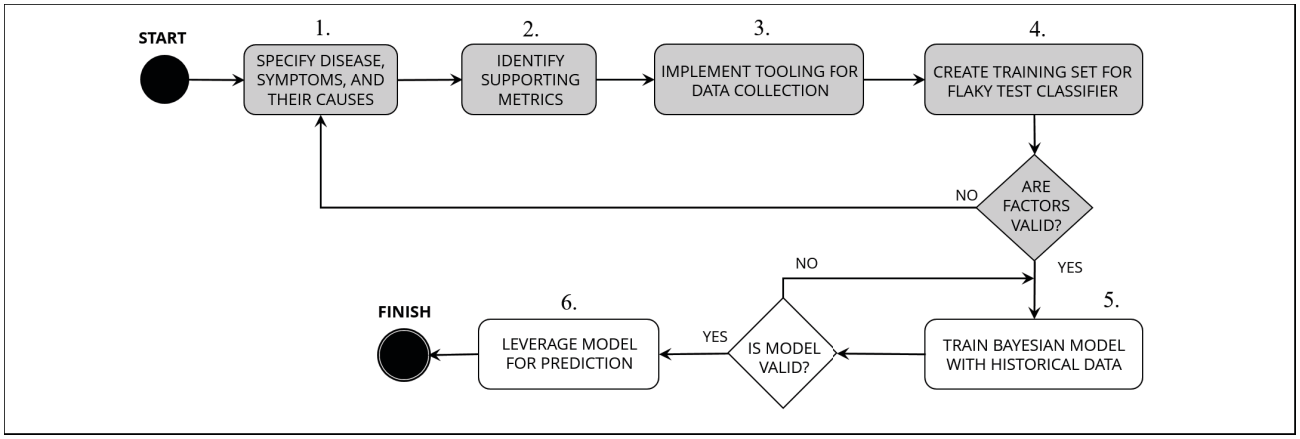

Figure 1: Workflow for Developing and Validating a Flaky Test Predictor using Bayesian Networks.

A. Approach

Our approach to developing a test flakiness predictor is illustrated in Figure 1, and can be described as follows:

1) *Specify Disease, Symptoms, and their Causes*. To be able to predict whether a given test is flaky, we must first be able to identify flaky tests via a set of characteristics. Observable symptoms of test flakiness may serve as a good starting point for these characteristics. However, since a given symptom may share other possible root causes, we must also model the probability of those shared causal factors.
2) *Identify Supporting Metrics*. After the disease, symptoms, and their causal factors are specified, a set of metrics are identified to support data collection. The goal of the metrics is to facilitate measuring the degree to which a particular characteristic holds.
3) *Implement Tooling for Data Collection*. A combination of static and dynamic analysis tools are employed to gather the data that will be used to train the model. Data collection should be completely automated to avoid human error and ensure that the data is kept in a uniform format. Tools should support sorting and filtering the data to help with validation.
4) *Create Training Set for Flaky Test Classifier*. In preparation for training, it is important to gather data and validate that the selected factors are appropriate for the classification problem. This step involves leveraging human experts to inspect the collected data, and formulate thresholds that can be used to divide the data into flaky and non-flaky examples. If they cannot achieve this goal using the selected factors, then the factors must be re-specified. Otherwise, the examples can be labeled and used to train the learning algorithm.
5) *Train Model With Historical Data*. Supervised learning is used to train the model after the historical data has been labeled. It is recommended that common techniques for validating the model during training (e.g., cross-validation) and measuring its overall performance (e.g., accuracy, precision, recall, F-score) are used at this stage to accept or reject the model [12].
6) *Leverage Model for Prediction*. With both the causal factors and trained model validated, the model can now be used to classify previously unseen flaky tests, and prevent them from being introduced into continuous integration and deployment pipelines.

B. Modeling Test Flakiness Symptoms and Common Effects

Figure 2 provides a graphical model of the test flakiness symptoms. Symptoms are represented as shaded nodes, while the unshaded nodes indicate the disease and common effect relationships. A medical example to illustrate this is the classification problem of diagnosing *lung disease*. Patients with this disease may exhibit the symptom *coughing*. However, since patients with the *cold* or *flu* are also likely to suffer from this symptom, we must consider these common effect relationships as part of the classification problem. The

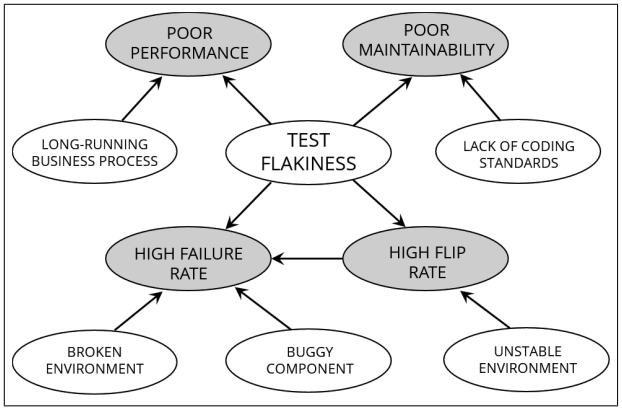

Figure 2: Test Flakiness Symptoms and Common Effects.

symptoms of test flakiness and common effect factors as they relate to Figure 2 are described as follows:

High Failure Rate: Unstable automated tests are prone to failure. Observing a high percentage of failed runs over time can be an indication of test flakiness. Too many failures can clutter the tests in the presence of false negatives. However, the failures may also be true negatives, i.e., legitimate failures due to defects in the program under test. Other non-script factors related to high failure rates include broken test environments and infrastructure issues, e.g., poor network conditions.

High Flip Rate: As previously mentioned, flaky automated tests pass sometimes and fail other times although neither the test code nor application code have changed. In other words, the test results appear to "flip" without warning or cause. When this occurs, development teams tend to adopt the practice of re-running tests until they pass. This is a waste of resources and leads to a breakdown in trust of the testing effort and results. Similar to failure rates, high flip rates may be the result of intermittent issues with the test environment and underlying infrastructure.

Poor Performance: Flaky test automation can exhibit long execution times. This is particularly true of large, complex system-level UI tests. Slowness during execution increases the probability of the test failing due to timing issues. In some cases the root cause of poor performance may have nothing to do with the test itself. Long-running business processes, memory leaks, and performance bottlenecks in the system under test all contribute to large and growing test run times.

Poor Maintainability: Flaky tests tend to be hard to read, understand and modify. As the test flakiness problem grows in an organization, teams find themselves spending more and more time maintaining tests, and less time producing new code. This symptom is one of the primary motivating factors behind this work since it may result in a significant drop in productivity. However, it is possible to find tests that are stable, yet hard to maintain. For example, the absence of coding standards may yield to poor readability, but not necessarily impact stability.

C. Modeling the Causes of Test Flakiness

Some of the factors that cause automated tests to be flaky are modeled in Figure 3. These include:

High Test Complexity: Large test scripts with many steps and assertions tend to be unstable. These types of automated tests often cover multiple scenarios, thereby violating the single responsibility principle [13]. Scripts that attempt to reuse code through inheritance and delegation may also exhibit unpredictable behavior.

Implementation Coupling: Test scripts that are tightly coupled to the program implementation are also likely to be flaky. Examples of this include referencing concrete classes rather than interfaces from unit tests, or directly referencing web element selectors from within system-level UI test scripts. As the system under test evolves, such tests tend to break frequently.

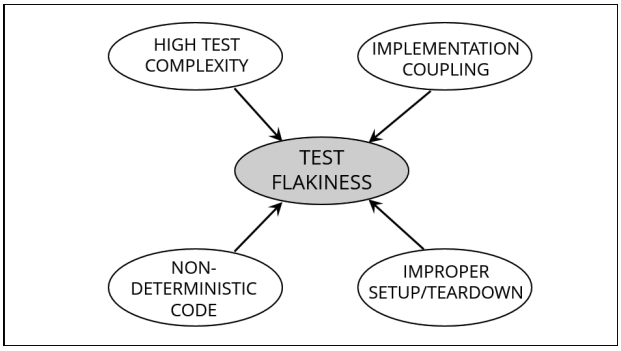

Figure 3: Causal Factors for Test Flakiness.

Non-Deterministic Code: Test scripts that contain conditional logic such as *if-then-else* and *try-catch* statements will take different paths depending on data and environmental conditions. This increases the probability that a test passes and fails even though the code has not changed. Another way that non-determinism can be introduced into test code is through explicit delays and random number generators.

Improper Setup/Teardown: Test scripts that do not teardown data and environmental conditions properly will introduce instability into the pipeline. While such tests may not impact their own execution negatively, they have side effects that may cause subsequent tests to fail. Similarly, tests that do not properly establish preconditions are candidates for being classified as flaky tests.

IV. Supporting Test Metrics

This section describes object-oriented test metrics [3] used to support data collection and discusses the rationale for collecting each metric. Subsection IV-B proposes a new test metric for measuring the stability of element selectors for web applications. Metrics have been grouped to make them easy to map to the models presented in Section III.

A. Complexity Metrics

- *Test Assertion Count*: the number of expressions that attempt to verify or confirm specific facts about the system under test. To analyze this metric at the test suite level, it is necessary to calculate the average assertion count across all test cases within each suite.
- *Test Class/Method Size*: the number of non-commented source lines of code associated with an automated test suite. To analyze this at the test suite level, the average method size across all tests should be collected.

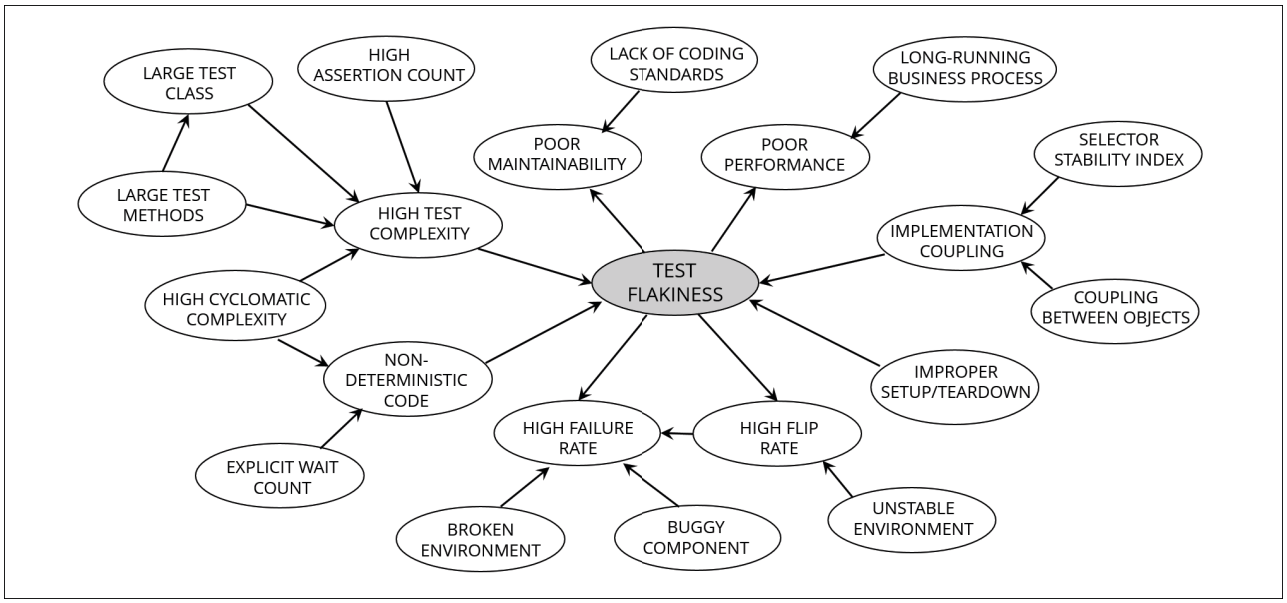

Figure 4: Bayesian Network for Test Flakiness Classification including Symptoms, Causal Factors and Supporting Metrics.

- *Depth of Inheritance Tree*: the number of parent classes along the inheritance hierarchy, starting from the leaf test suite class and leading up to the root class.

B. Implementation Coupling Metrics

- *Coupling Between Objects*: a count of the unique number of reference types that occur through method calls, parameters, return types, exceptions, and fields.
- *Selector Stability Index (SSI)*: a measure of both web element selector brittleness, as well as program implementation coupling. The SSI is defined by:

$$SSI = 1 - \frac{\sum_{n=1}^{m} \alpha P(n) + \beta H(n) + \gamma S(n)}{m}$$

$$P(n) = \begin{cases} 1, & \text{if } n \text{ is presentational} \\ 0, & \text{otherwise} \end{cases} \quad (1)(2)$$

$$H(n) = \begin{cases} 1, & \text{if } n \text{ is hierarchical} \\ 0, & \text{otherwise} \end{cases} \quad (3)(4)$$

$$S(n) = \begin{cases} 1, & \text{if } n \text{ is structural} \\ 0, & \text{otherwise} \end{cases} \quad (5)(6)$$

Three selector characteristics in the SSI definition contribute toward increased implementation coupling. Presentational selectors are defined as those containing information specific to styling and/or positioning, e.g., element class names. Hierarchical selectors represent parent and/or child information. Structural selectors have references to HTML elements that are typically used to control document block placement, e.g., div, p, table, span. The weights α, β and γ control the extent to which each characteristic affects stability.

C. Non-Determinism Metrics

- *Cyclomatic Complexity*: a measure of the number of linearly independent paths through a program [15]. Directly relates to the number of branches in the code.
- *Explicit Wait Count*: the number of statements in a test suite that cause forced delays to occur at runtime, e.g., Thread.Sleep()

D. Performance Metrics

- *Average Execution Time*: the total execution time of the test suite over a specified number of test runs.

E. General Stability Metrics

- *Failure Rate*: the number of unsuccessful test runs divided by the total number of test runs over a period of time.
- *Flip Rate*: the number of times a test fixture alternates between success and failure divided by the total number of test runs over a period of time.

V. Bayesian Network Classifier for Flaky Tests

This section presents a Bayesian network for classifying automated test scripts based on the flakiness symptoms, causal factors and supporting metrics described in Sections III and IV. Once trained on historical data, the model should be able to classify previously unseen tests as either flaky or stable (not flaky). An illustrative example of how the model computes the classification is also provided in this section.

Figure 4 shows the structure of the flaky test classifier. Recall that each node in the model represents a random

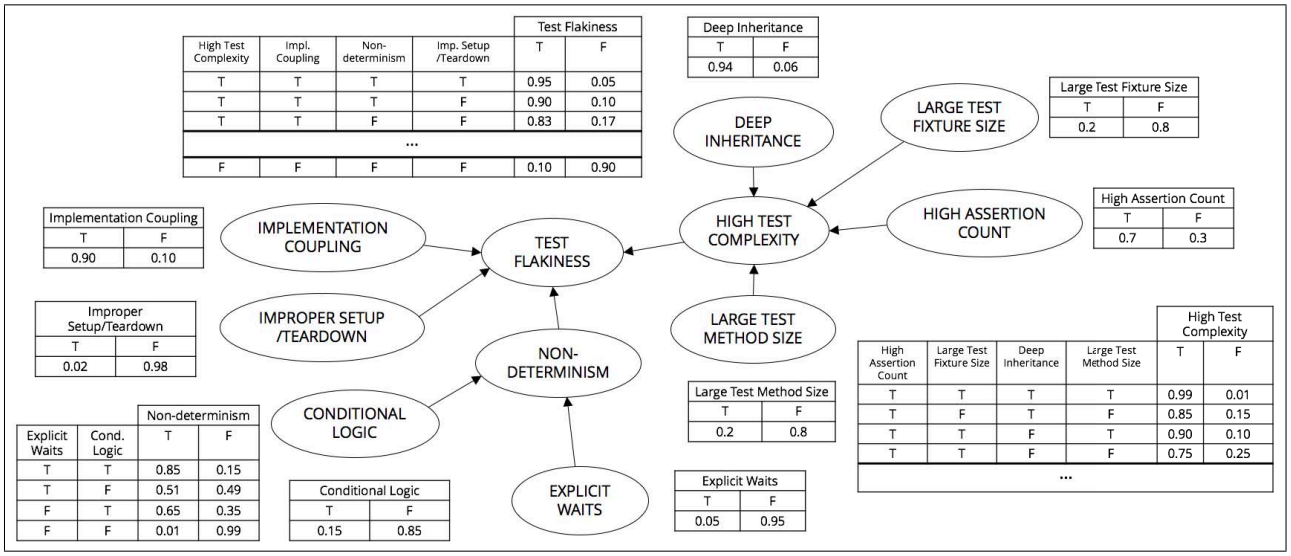

Figure 5: Illustrative Example showing a Partial Model of the Test Flakiness Bayesian Network Classifier.

$$P(TF = T \mid HTC = F, IC = F, ND = F, IST = F) \quad *$$
$$P(HTC = F \mid HAC = T, LTFS = F, DI = T, LTMS = F) \quad *$$
$$P(ND = F \mid CL = F, EW = F) \quad *$$
$$P(IC = F) * P(IST = F) * P(HAC = T) * P(LTFS = F) \quad *$$
$$P(DI = T) * P(LTMS = F) * P(CL = F) * P(EW = F)$$

$$(0.10) * (0.15) * (0.99) *$$
$$(0.10 * 0.98 * 0.7 * 0.8) *$$
$$(0.94 * 0.8 * 0.85 * 0.95) \approx 0.05\%$$

(a)

$$P(TF = T \mid HTC = T, IC = T, ND = F, IST = F)$$

$$(0.83) * (0.85) * (0.99) \quad *$$
$$(0.90 * 0.98 * 0.7 * 0.8) \quad *$$
$$(0.94 * 0.8 * 0.85 * 0.95) \approx 21\%$$

(b)

Figure 6: Example Test Flakiness Probability Calculation.

variable, and each directed edge corresponds to a conditional dependency between pairs of variables. A subset of the model along with associated conditional probability tables (CPTs) are shown in Figure 5. Applying the full joint probability formula [6] to the CPTs allows the model to be leveraged to answer classification questions.

Consider a scenario where we query the model for the probability that a test is flaky ($TF = T$) given that it *does not* have high measures for test complexity ($HTC = F$), implementation coupling ($IC = F$), non-deterministic code ($ND = F$), and improper setup/teardown statements ($IST = F$). The query produces a calculation that yields a probability of approximately 0.05% (Figure 6a). However, in the case where two of these variables are set to true, the result climbs to 21% (Figure 6b). Note that the probability values themselves need not be precise to be practically useful. Observing the rate of change and/or comparing the values of rough estimates with those in the training set can help when classifying new tests.

VI. Case Study

The approach presented in this paper evolved from an initiative to improve the efficiency of the testing process at Ultimate Software [26]. The primary goal of the initiative was to identify, quarantine, and fix flaky system-level UI tests that run within continuous integration pipelines. This section describes the case study including its context, environmental setup, objectives, procedures, and results.

A. Context

Ultimate Software is a leading provider of a human capital management (HCM) software-as-a-service solution named UltiPro [5], [26]. The core of UltiPro is composed of global human resource management, payroll, and benefits functionality. A number of applications are developed to integrate seamlessly with the core. These include but are not limited to: talent acquisition, talent management, compensation management, time and workforce management [26].

Product engineering at Ultimate Software is divided into six application domains. Each application domain consists of two or more development teams, and each team has their own continuous integration (CI) project pipeline. Team CIs run unit, integration, and system level tests against new builds of UltiPro as they become available. Across teams there is a total of approximately $280K$ automated tests.

B. Environment Setup

Ultimate Software leverages an enterprise version of TeamCity [8] for continuous integration. Build agents listen for commands from a central server to download the updated source code and start the build process. If the build process is successful, the new build is automatically applied to a test environment. The test environments and test runners are spun up on-demand using an internal cloud infrastructure.

System tests are developed using the Echo framework [14], a domain-specific middleware architecture for web UI test automation. Echo provides a layer of abstraction on top of browser and image-based interaction frameworks such as Selenium [7] and [18], respectively. NUnit [21] provides Echo with constructs for test set up, teardown, execution, and reporting.

C. Objectives and Procedures

The goal of identifying, quarantining, and fixing flaky system tests also brought with it the opportunities for fulfilling the following case study objectives: (1) validating the symptoms and causal factors identified in Section III, (2) developing and implementing the necessary tooling for collecting data based on the supporting metrics in Section IV, and (3) creating a set of examples of flaky and non-flaky tests that can be used to train a flaky test classifier. Mapping these objectives to the workflow in Section III-A, our case study primarily investigates the feasibility of the first four steps, i.e., the shaded portion of Figure 1.

Procedures for the case study involved selecting five development teams from a cross-section of HCM application sub-domains. The teams selected ranged from having between approximately 150 to 1000 test suites each, and an overall test execution pass rate ranging from 20% to 95%. The success criteria for the project was for all teams to have a 90% or higher passing rate. One team that was already above the threshold was selected to determine how effective the approach could be at maintaining the high passing rates. In terms of personnel, one test lead per team was assigned to provide guidance and track progress.

D. Preliminary Results

Some of the metrics needed for the case study were readily available via TeamCity's built-in reporting [8]. These included dynamic analysis metrics such as flip rate, failure rate, and test duration. However, there were neither mechanisms for looking at the trends of these metrics over specific time periods nor gathering any of the static analysis metrics. To overcome these drawbacks, we developed an internal tool that extended TeamCity with all of the desired metrics and trending capabilities.

Our internal test metrics tool combines the results of static and dynamic analysis for system-level test scripts into one view. The metrics and associated trends exposed by the tool include: mean pass rate, mean duration, mean flip ratio on the same build, mean flip ratio across different builds, number of lines of code, number of sleeps, cyclomatic complexity, and number of assertions. The tool facilitates searching for specific data points, sorting the results, and filtering results based on a failure rate threshold.

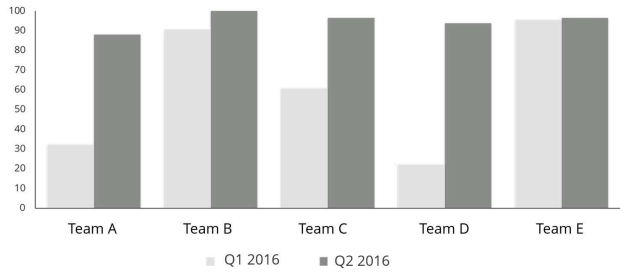

Figure 7: Test Pass Rates Pre- and Post- Clean Up Effort.

Within a three month period all five teams were able to use the tooling to successfully quarantine and fix problematic tests. Figure 7 shows the pass rate percentages of each team at the beginning and end of the quarter. Teams with highly unstable pipelines experienced significant gains, improving stability by as much as 60%. On the other hand, those with an already stable pipeline only yielded a $\sim 2\%$ increase. Quarantined tests were tagged and separated out into a local environment, where they were debugged, fixed and reintegrated into the pipeline. Previous versions of the tests were maintained in source control.

The corpus of pre- and post- clean up scripts makes up the dataset to be used for training the test flakiness classifier. A prototype consisting of a subset of the model was implemented using the Genie Modeler [1]. Data from a one month test execution period for a single team was imported into the model in comma separated value format. There were a total of 714 unique test suites with 26,298 observed test runs in the training data. Table I contains the training evaluation results which indicated an overall test flakiness prediction accuracy of 65.7%.

Characteristic	Accuracy	True	False
High Test Complexity	0.889	511/673	2275/2462
Large Test Fixture Size	0.822	404/486	2173/2649
Poor Performance	0.458	919/919	516/2216
Test Flakiness	0.657	188/583	1880/2552

Table I: Test Flakiness Prototype Training Results.

VII. Conclusion and Future Work

Conducting the case study provided preliminary evidence that the approach presented in this paper is feasible. However, the research study is limited and only sets the stage for a more in-depth training and evaluation of our test flakiness classifier. An extended case study focusing on the unshaded portion of Figure 1 is necessary to fully

evaluate the approach. Furthermore, a comparative case study across multiple organizations is needed to support or refute the stated research hypothesis, and any other claims to the generality, practicality and usefulness of the approach. Once complete we plan to share detailed training results and lessons learned. Future work will also address the degree to which each of the proposed metrics is useful, and the interoperability of the metrics in practice. Lastly, a long-term future direction involves the development of a recommendation engine for auto-correcting flaky tests, i.e., a self-healing continuous integration pipeline.

ACKNOWLEDGMENT

The authors would like to thank Philip Daye, David Skirvin, Andrew Pomerans, Jairo Pava, Yunior Betancourt, Eduardo Pena, and Gabriel Perez Clark for their contributions to this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of Ultimate Software or Florida International University.

REFERENCES

[1] Bayes Fusion LLC. GeNIe - Complete Modeling Freedom, June 2018. https://www.bayesfusion.com/genie-modeler.

[2] B. Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[3] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[4] Z. Gao. *Quantifying Flakiness and Minimizing its Effects on Software Testing*. PhD thesis, University of Maryland, 2017.

[5] Gartner, Inc. Magic Quadrant for Cloud HCM Suites for Midmarket and Large Enterprises, August 2017. https://www.gartner.com/doc/reprints?id=1-4A4OK7Q&ct=170815&st=sb.

[6] A. Gut. *Probability: A Graduate Course*. Springer-Verlag, New York, 2nd edition, 2013.

[7] J. Huggins and S. Stewart. Selenium - Web Browser Automation, April 2018. https://www.seleniumhq.org/.

[8] JetBrains. TeamCity - A Hassle-free CI and CD Server, April 2018. https://www.jetbrains.com/teamcity/.

[9] T. M. King and J. Arbon. AISTA: Artificial Intelligence for Software Testing Association, April 2018. www.aitesting.org.

[10] A. Kolawa and D. Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[11] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.

[12] S. Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[13] Martin, Robert C. Clean Coder - Getting a SOLID start, February 2009. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start.

[14] J. Martinez, T. Thomas, and T. M. King. Echo: A middleware architecture for domain-specific ui test automation. In *Proceedings of the 2014 Workshop on Joining AcadeMiA and Industry Contributions to Test Automation and Model-Based Testing*, JAMAICA 2014, pages 13–15, New York, NY, USA, 2014. ACM.

[15] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[16] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[17] J. Micco. Flaky Tests at Google and How We Mitigate Them, May 2016. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[18] MIT User Interface Design Group . Sikuli Script - Automation for anything on the screen, April 2018. https://sikuli.org/.

[19] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[20] A. S. Namin and M. Sridharan. Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 349–354, New York, NY, USA, 2010. ACM.

[21] C. Poole, R. Prouse, S. Busoli, and N. Colvin. NUnit - Unit Testing for All .NET Languages, April 2018. www.nunit.org.

[22] D. Rodriguez, J. Dolado, and J. Tuya. Bayesian concepts in software testing: An initial review. In *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, A-TEST 2015, pages 41–46, New York, NY, USA, 2015. ACM.

[23] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[24] I. C. Society, P. Bourque, and R. E. Fairley. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA, 3rd edition, 2014.

[25] The Code Gang. Flaky Tests: A War that Never Ends, December 2017. https://hackernoon.com/flaky-tests-a-war-that-never-ends-9aa32fdef359.

[26] Ultimate Software. UltiPro Enterprise: Human Capital Management Solutions, April 2018. www.ultimatesoftware.com/HRIS.