Debugging Flaky Tests on Web Applications

Jesus Morán[1][a], Cristian Augusto[1][b], Antonia Bertolino[2][c], Claudio de la Riva[1][d] and Javier Tuya[1][e]

[1]*Computer Science Department, University of Oviedo, Gijón, Spain*

[2]*ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy*

Keywords: Software Testing and Debugging, Spectrum-based Localization, Web Applications, Test Flakiness.

Abstract: Testing web applications is a challenging practice because it involves managing asynchronous requests between clients and servers, the integration of heterogeneous technologies, and concurrent accesses to the resources. Therefore, rerunning the test cases of these applications under the same conditions is difficult as one test case can be executed in many different ways according to several environmental factors like memory, screen size or network. Moreover, some of these test cases could be flaky, i.e., due to environmental factors the test outcome can vary even though the application did not change. Understanding which factors are the root cause of flakiness is very important for web developers to both prevent and fix flakiness. This paper introduces a technique to locate the root causes of flakiness based on a characterization of the different environmental factors that are not controlled during the testing of web applications. The root cause of flakiness is located by a spectrum-based localization technique that analyses the execution of the same flaky test under different environmental factors that can trigger the flakiness. The technique is illustrated on an educational web platform named FullTeaching.

1 INTRODUCTION

Software testing and debugging play an important role in the evaluation of software quality, but there are several open challenges (Bertolino 2007). The design and execution of the test cases of web applications are complex due to the distributed interoperations between heterogeneous clients and servers. These test cases can be executed each time in different ways according to environmental factors like the underlying network bandwidth, the memory or the timeouts in web server responses. The non-deterministic execution can introduce flakiness in the test cases of web applications. A test is considered flaky when the same test with the same system-under-test obtains different outcomes due to the environmental factors executed (Luo et al., 2014). Testers cannot rely on the outcome of flaky tests.

According to a recent study, the developers face flakiness frequently and they usually stop to rely on flaky tests (Eck et al., 2019). Despite debugging these tests is considered time-consuming, the majority of developers also consider that finding the root cause of flakiness is relevant in order to fix it, but it is also a very difficult challenge (Eck et al., 2019).

In this paper, we introduce an ongoing technique to locate the root cause of flakiness in test cases of web applications. This technique is based on a characterization of the environmental factors that are not controlled during testing and can cause flakiness. Based on this characterization, the test case is executed several times under different environmental factors so to get insights about flakiness. These executions are analyzed with a spectrum-based localization technique (Wong et al., 2016) considering that the factors that usually triggers the flakiness are more prone to be the root cause of flakiness.

The contributions of paper include:

[a] https://orcid.org/0000-0002-7544-3901
[b] https://orcid.org/0000-0001-6140-1375
[c] https://orcid.org/0000-0001-8749-1356
[d] https://orcid.org/0000-0001-5592-9683
[e] https://orcid.org/0000-0002-1091-934X

Morán, J., Augusto, C., Bertolino, A., de la Riva, C. and Tuya, J.
Debugging Flaky Tests on Web Applications.
DOI: 10.5220/0008559004540461
In *Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*, pages 454-461
ISBN: 978-989-758-386-5
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

1. Introduction of a technique called FlakcLoc to locate the root cause of flakiness in web applications.
2. The application of the technique to a real-world web application.

The remainder of this paper is organized as follows. The testing of web applications is introduced in Section 2. The related work about flaky tests is discussed in Section 3. The technique FlakcLoc is introduced in Section 4 and a practical working example of this technique is described in Section 5. Finally, the conclusions and future work are in Section 6.

2 FLAKINESS IN TESTING WEB APPLICATIONS

The functionality of web applications is implemented with code executed in a distributed architecture. The client-side code performs web requests that are responded by the server-side code. These interactions from the client to the server are tested performing the user actions across the web interface and checking if the server responds properly. The WebDrivers allow the automatization of the tests controlling the user actions in a browser. There are different tools to support the automatic execution of tests for web applications, such as Selenium (Selenium HQ 2019).

These tools provide several WebDrivers that support the execution of the test in different browsers. However, there are other environmental factors that can affect the execution of the tests. For example, suppose a simple test that pushes a button and awaits 2 seconds to check if the server response is right. The execution of the previous test can be affected by several environmental factors like the screen resolution, memory or network. These factors can cause flakiness in the test, so that it sometimes passes and other times fails, as in the following examples. The test passes when it is executed in large screen resolutions because it is able to find the button. In contrast, the test can fail when it is executed in small screen resolutions because the button can be hidden automatically inside of the responsive menu. The test can also fail if the button is not rendered due to lack of memory. In case the button is pushed correctly, the test awaits 2 seconds for the server response, however the test can also fail if the server employs more time due to network congestion. In the previous examples, the test is flaky because the tester cannot rely on its outcome as sometimes the test fails, and other times it passes.

The presence of a flaky test is common (Eck et al., 2019), and some researchers propose the aphorism 'Assume Test are Flaky' (ATAF) (Harman and O'Hearn 2018). In order to deal with this flakiness, the testing tools usually provide different mechanisms based on the re-execution. JUnit has the @RepeatedTest(10) tag that executes the test 10 times to avoid "failures" due to the environmental factors of the execution (Bechtold et al., 2019). In a similar way, the Spring framework has the @Repeat(10) tag (Pivotal Software 2014). For the case of progressive web applications, Android provides the @FlakyTest(tolerance=10) tag (Google 2019). Maven also support the re-execution of those tests that fail using the Surefire plugin with the option -Dsurefire.rerunFailingTestsCount=10 (Apache Software 2018). Based on the previous, Jenkins provides the Flaky Test Handler plugin (Luo and Micco 2015).

The previous tools re-execute several times the flaky test in order to check if the test passes in at least one execution. However, the tester could not rely on the test because it is still flaky, and its execution is not easy to reproduce. In order to both avoid and fix the flakiness, the developers consider very important the root cause of flakiness (Eck et al., 2019). In this paper we introduce FlakcLoc to locate the root cause of flakiness in web applications.

3 RELATED WORK

Root Causes of Flakiness:
There are several empirical studies that characterize the causes of flakiness. Luo et al., (Luo et al., 2014) characterize the following 11 causes of flakiness after analyzed 51 open-source projects: asynchronous waits, concurrency, test order dependency, resource leak, network, time, IO, randomness, unordered collections and others. The majority of flakiness is caused by asynchronous waits, as for example when the Selenium WebDriver sends an asynchronous web request and does not await enough time for the server response. Thorve et al., (Thorve, Sreshtha, and Meng 2018) analyzes 29 Android applications characterizing another three root causes of flakiness: Dependency, Program Logic, and UI. This kind of flakiness can happen also in web applications, especially the Dependency and UI. The Dependency flakiness is caused by the use of specific hardware, devices or thirty party libraries. The UI flakiness is caused by the misunderstood of the rendering process and user interface. Eck et al., (Eck et al., 2019) characterize another four root causes of flakiness

analyzing Mozilla: Too Restrictive Range, Test Case Timeout, Platform Dependency, and Test Suite Timeout. These kinds of flakiness can happen in web applications, especially Test Case Timeout and Platform Dependency. The Test Case Timeout flakiness is caused when the test does not finish in proper time and it is killed. The Platform Dependency flakiness is caused when the test passes in one platform, but it fails in another, such as for example those tests that pass in one version of the browser but they fail in another. The previous studies are the basis of our paper, that proposes a technique to locate the root causes of flakiness. Based on these studies, we characterize a series of environmental factors that are prone to trigger flakiness in web applications.

Detection of Flaky Tests:
The flaky tests are prevalent in practice (Vahabzadeh, Fard, and Mesbah 2015). The common way to detect if a test is flaky is to re-execute it. However, some researchers propose different approaches. Palomba and Zaidman (Palomba and Zaidman 2017), studied the relationship between flakiness and code smells, concluding that the flakiness of 54% of flaky test tests can be attributed to code smells. Muslu et al., (Muşlu, Soran, and Wuttke 2011) propose to isolate the execution of each test to detect problems related to dependencies. Bell et al., (Bell et al., 2018) propose to detect the flakiness when the same system-under-test code is covered by two executions of the same test, one passing and the other failing. The detection of the flaky tests is outside the scope of the present paper. In this paper, we locate the root cause of the flakiness in a given flaky test re-executing the flaky test with different environmental factors.

Root Flakiness Detection:
Lam et al., (Lam et al., 2019) propose to categorize the kind of flakiness by analysing the logs after several test executions, and locating the suspicious lines of code that trigger the flakiness. The previous technique and our paper are orthogonal because both techniques aim to improve the understanding of the flakiness, but providing complementary insights about the root cause of flakiness. Our technique (FlakcLoc) instead to provide the kind of flakiness and the line of code that triggers the flakiness, it provides the suspicious environmental factors that cause the flakiness. These environmental factors are obtained by FlakcLoc based on both the characterization and analysis of several executions through a spectrum-based localization.

4 FLAKINESS LOCALIZATION

In this section, we describe an ongoing proposal to locate the root cause of flakiness in the flaky test of web applications. A flaky test is a test that sometimes passes and other fails depending on a combination of different environmental factors that are not controlled and therefore can introduce flakiness in the test, as for example the screen size, the version of the browser, or the network traffic. We refer as "factor" to each one of the environmental characteristics that can alter the test execution, and we refer as "configuration" to one of the combinations of the previous factors.

The proposed technique, FlakcLoc, is summarized in Figure 1. This technique locates the root cause of flakiness based on the characterization of the different environmental factors that are not controlled in the flaky tests (Characterization). FlakcLoc executes the flaky test in different configurations that differ on the selection of its factors (Execution). The root cause of the flakiness is then automatically located by a spectrum-based localization technique that analyses what factors are shared by those executions that trigger the "failure" (Analysis). In the remainder of this section, we detail the main processes proposed: characterization of the factors that can cause flakiness, execution of the test in different configurations, and analysis of the root cause of flakiness.

Characterization:
We characterize the configuration that triggers the flakiness according to the potential environmental factors that can cause the flakiness. In web applications, a configuration is characterized according to a set of factors, such as those indicated below:

- Memory can cause issues in the WebDrivers,

Figure 1: Technique to locate the flakiness.

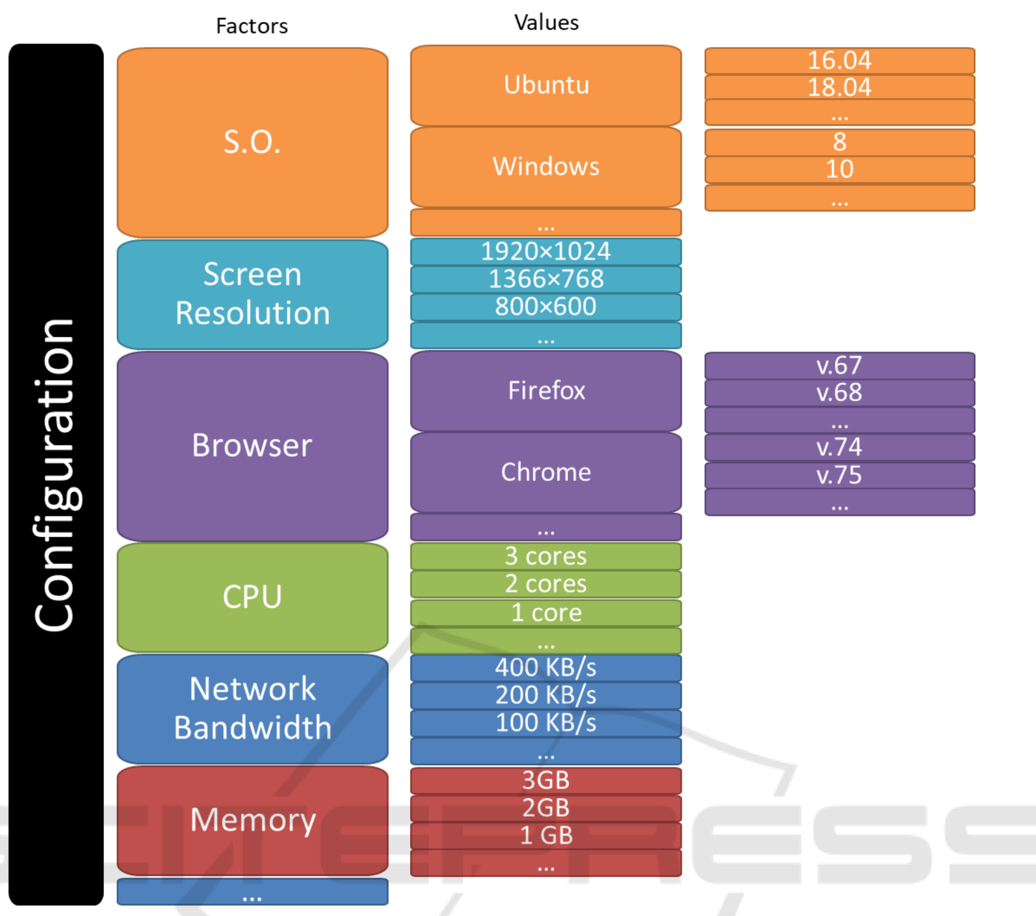

Figure 2: Model of the configurations with several characteristics.

especially when several sessions and browsers are not properly closed and consume the same memory.

- The network is one of the main causes of flakiness (Luo et al., 2014) that can produce delays and race conditions in the asynchronous web requests.
- CPU can increase or decrease the computation and the concurrency, which is one of the main issues of flakiness (Luo et al., 2014).
- Browsers and different versions of these browsers can alter the execution of the test making flakiness for different reasons such as rendering the objects in a different way.
- Screen resolution can modify the test execution, especially for those responsive applications as it can hide/expose relevant web elements during testing.
- The operating system can also produce flakiness, especially when the application uses a workspace or other environmental variables.

Each one of these factors takes one discrete value from those depicted in Figure 2. The condigurations are modelled according to the factors and the values that takes these factors. Thus, each configuration is composed of several factor-value pairs. For example, a configuration can be composed by 1GB of memory (memory-1GB pair), 100KB/s as Network bandwidth (network bandwidth - 100KB/s pair), and so on for the remaining factors.

Execution:
The same test case can be executed in different ways according to the previous characterization, some of them cause flakiness while others hide its flakiness. FlakcLoc proposes to execute the same flaky test under different configurations. For example:

- Configuration 1: Memory - 2GB, Network bandwidth - 400KB/s, CPU - 1 core, Browser - Google Chrome version 75, Screen resolution - 800×600, and Operating system - Microsoft Windows 10.
- Configuration 2: Memory - 1GB, Network bandwidth - 200KB/s, CPU - 1 core, Browser -

Google Chrome version 75, Screen resolution - 800×600, and Operating system - Microsoft Windows 10.

- Configuration 3: Memory - 1GB, Network bandwidth - 400KB/s, CPU - 1 core, Browser - Firefox version 67, Screen resolution - 800×600, and Operating system - Ubuntu 18.04.

The execution of the test in the previous configurations provides insights about the root cause of flakiness, especially those factors that usually trigger the flakiness. Suppose that the test executed with Configurations 1 and 3 passes, but the same test executed with Configuration 2 triggers a "failure" because the test cannot perform the user interactions due to the lack of the web elements required.

The factors of Configuration 2 make the test flaky whereas those factors of Configurations 1 and 3 hide the flakiness. Configuration 1 hides the flakiness with 2GB of Memory and 400Kb/s in contrast, Configuration 2 triggers the "failure" as both memory and network bandwidth are decreased. In these cases, we have some evidence that memory and network can cause flakiness. This evidence is analysed systematically with the following approach based on the fault localization techniques.

Analysis:
We analyse several executions with a ranking metric to obtain a prioritized list of the suspicious factors that cause flakiness. Whereas the ranking metrics in fault localization analyse the lines of code that cause the fault (Harrold et al., 2000, 2005), in FlakcLoc the ranking metrics analyse the factors that cause flakiness.

Suppose the previous 3 configurations described before. We analyse these executions with a ranking metric like Ochiai (Abreu, Zoeteweij, and Van Gemund 2007) obtaining that the most suspicious root cause of flakiness is 200KB/s of network bandwidth. This ranking metric analyses the similarity between the values of the factors executed and the configurations that fail/hide the flakiness. The failure is triggered with 1GB of memory in Configurations 2, but apparently is not the root cause of flakiness because 1GB also hides the flakiness in Configuration 3. A memory of 2GB is not also the root cause of flakiness because it never triggers the flakiness. In contrast, 200KB/s of network bandwidth always triggers the flakiness. After analyzing, in the same way, all factors through the localization technique, we determine that the root cause of flakiness is 200KB/s of network bandwidth. According to Ochiai ranking metric: 200KB/s (1 out of 1 of suspiciousness), 1GB of memory (0.707 out of 1 of suspiciousness), and so on.

The root cause of flakiness can improve the understanding of the flaky test in order to avoid it or fix it. The previous test case passes with 400KB/s of network bandwidth because the web requests are responded quickly just before the user interaction takes place. However, with less network bandwidth (200KB/s), the web requests are responded slowly causing that the test fails because it tries to execute the user interactions before the responses. This flakiness can be avoided in different ways like increasing the time of *sleep* or *waitFor* to wait for the web responses.

5 WORKING EXAMPLE

In this section, we illustrate how FlakcLoc is able to localize the root cause of flakiness on a web application called FullTeaching (Pérez 2017). This web application is an educational online platform on which teachers and students can perform the lessons and share their teaching materials, like calendars dashboards and forums. This project has several test cases including End-to-End tests that execute the whole system (web application, streaming server, and database). Several of these End-to-End tests are flaky because the same test sometimes passes and other fails in a non-deterministic way. In the remainder of this section, we detail the localization of the root cause of flakiness in one flaky test of FullTeaching web application.

We consider a test that checks if the user is able to log into the application, access the courses and logout. Despite the tests are executed in an isolated environment through a containerized instance, the test sometimes fails due to the configuration executed. This test was correctly executed in the tester's computer, but the same test failed in the Continuous Integration server. In both environments, the test was executed isolated inside of a container with the same resources. We checked that the system-under-test and the test case were properly deployed in the Continuous Integration server, but the flakiness remains.

In order to locate the root cause of flakiness, the technique proposed in Section 4, FlakcLoc, is applied to the previous flaky test:

Characterization:
We characterize those factors that can trigger the failure. This example is illustrated with the following factors-values pairs:

- Memory: the test execution is modelled with 90MB and 240MB to increase or decrease the WebDriver resources.
- CPU: the execution is modelled with 1 and 4 cores to increase or decrease the concurrency between the threads executed by the test case.
- Screen resolution: the execution is modelled with SVGA (800×600), HD (1366×768), and FullHD (1920×1024) resolutions. These resolutions can increase or decrease the web elements that are rendered in the navigator window.

Execution:
We execute several times the flaky test varying the previous environmental factors in the following configurations:

- Configuration 1: 90MB of memory, CPU with 4 cores, and a screen resolution of 800×600.
- Configuration 2: 90MB of memory, CPU with 1 core, and a screen resolution of 1366×768.
- Configuration 3: 90MB of memory, CPU with 1 core, and a screen resolution of 1920×1024.
- Configuration 4: 240MB of memory, CPU with 4 cores, and a screen resolution of 800×600.
- Configuration 5: 240MB of memory, CPU with 1 core, and a screen resolution of 1920×1024.
- Configuration 6: 240MB of memory, CPU with 4 cores, and a screen resolution of 1366×786.

The configurations 2, 3, 5 and 6 pass, whereas the configurations 1 and 4 fail in a flaky way. These executions provide insights about the root cause of the flakiness. We can observe that the “failure” is triggered with 1 core and a screen resolution of 800×600.

Analysis:
We analyze the previous executions with the localization technique proposed in Section 4 using the Ochiai ranking metric. According to this analysis, the most suspicious factor is the screen resolution of 800×600 (1.0 of suspiciousness), following by the CPU with 1 core (0.816 of suspiciousness).

This information is valuable to understand the flakiness in order to avoid it or fix it. The flakiness was triggered in the Continuous Integration server because it isolates the test in a container with low screen resolution. In contrast, the test is executed property in the computer of the tester because there it isolates the test in a container with more large resolution. In one part of this test, the Selenium WebDriver must push the button highlighted in Figure 3. However, the same button is hidden in low resolutions like 800x600 as in Figure 4. This test is executed inside of a container deployed by Docker, and the test passes/fails depending on the screen resolution assigned by Docker to the containers. the taxonomy of flakiness proposed by Eck et al., (Eck et al., 2019) , the flaky test described in this section is ‘Platform Dependent’.

SCIENCE AND TECHNOLOGY PUBLICATIONS

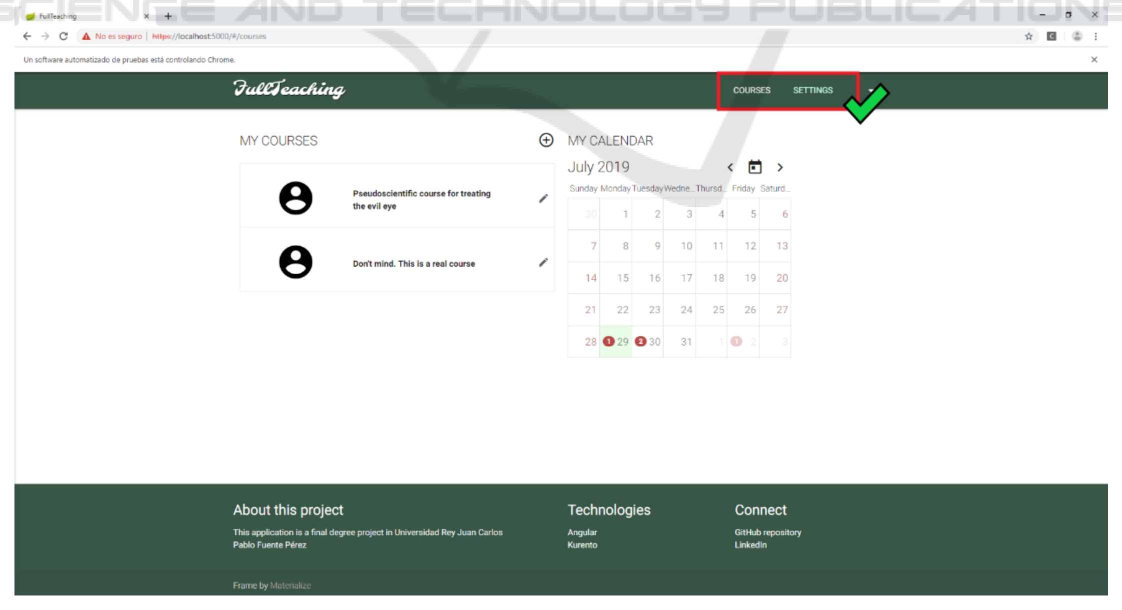

Figure 3: Web application with a 1920x1080 resolution.

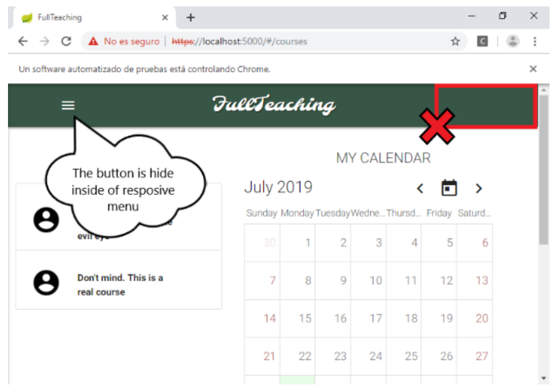

Figure 4: Web Application with A 800x600 Resolution.

6 CONCLUSIONS AND FUTURE WORK

This paper introduces a technique called FlakcLoc to locate the root cause of flakiness in the domain of web applications. FlakcLoc is based on a characterization of the environmental factors that can introduce flakiness in the test, like the screen resolution, network or memory. Varying these environmental factors, the technique executes the flaky test several times providing insights about the flakiness. Analyzing the factors executed and the times that the test fails, the root cause of flakiness of web applications is located using a spectrum-based localization technique. This paper provides a practical example on the localization of the root cause of flakiness in an educational web application.

There are a number of open questions that we can summarize in three main lines for future work. The first one is to enhance the characterization of environmental factors that cause flakiness in web applications including new factors. In this line of work, we plan to both analyze and reproduce several flaky tests in order to obtain more environmental factors of their flakiness. The second line of work is focused on the formalization of the technique with a meta-model and transformations. This meta-model could allow the localization of the root causes of flakiness in different domains. For example, the flakiness of the robotic domain can be located in a similar way but with the different environmental factors like GPU usage or sensor measurements. Last, the third line of future work is concerned with validating the proposed technique in a benchmark of web applications. In this validation, we plan to answer several research questions such as the evaluation of the best ranking metrics to locate the flakiness in web applications.

ACKNOWLEDGEMENTS

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under TestEAMoS (TIN2016-76956-C3-1-R) project and ERDF funds, and by the European Project ElasTest in the Horizon 2020 research and innovation program (GA No. 731535).

REFERENCES

Abreu, Rui, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. “On the Accuracy of Spectrum-Based Fault Localization.” Pp. 89–98 in *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*. IEEE.

Apache Software, Foundation. 2018. “Maven Surefire Plugin – Rerun Failing Tests.” Retrieved June 29, 2019 (https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html).

Bechtold, Stefan, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, and Christian Stein. 2019. “RepeatedTest (JUnit 5.2.0 API).” Retrieved June 29, 2019 (https://junit.org/junit5/docs/5.2.0/api/org/junit/jupiter/api/RepeatedTest.html).

Bell, Jonathan, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. “DeFlaker: Automatically Detecting Flaky Tests.” Pp. 433–44 in *Proceedings of the 40th International Conference on Software Engineering - ICSE ’18*. New York, New York, USA: ACM Press.

Bertolino, Antonia. 2007. “Software Testing Research: Achievements, Challenges, Dreams.” Pp. 85–103 in *2007 Future of Software Engineering*, *{FOSE} ’07*. Washington, DC, USA: IEEE Computer Society.

Eck, Moritz, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. “Understanding Flaky Tests: The Developer’s Perspective.” *To Appear FSE19/ESEC*.

Google. 2019. “FlakyTest | Android Developers.” Retrieved June 28, 2019 (https://developer.android.com/reference/android/support/test/filters/FlakyTest.html).

Harman, Mark and Peter O’Hearn. 2018. “From Start-Ups to Scale-Ups: Opportunities and Open Problems for Static and Dynamic Program Analysis.” Pp. 1–23 in *Proceedings - 18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018*. IEEE.

Harrold, Mary Jean, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. 2000. “Empirical Investigation of the Relationship between Spectra Differences and Regression Faults.” *Software Testing Verification and Reliability* 10(3):171–94.

Harrold, Mary Jean, Gregg Rothermel, Rui Wu, and Liu Yi. 2005. “An Empirical Investigation of Program Spectra.” *ACM SIGPLAN Notices* 33(7):83–90.

Lam, Wing, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. “Root Causing Flaky Tests in a Large-Scale Industrial Setting.” Pp. 101–11 in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2019*. New York, New York, USA: ACM Press.

Luo, Qingzhou, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. “An Empirical Analysis of Flaky Tests.” Pp. 643–53 in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. New York, New York, USA: ACM Press.

Luo, Qingzhou and John Micco. 2015. “Flaky Test Handler v1.04.” Retrieved June 29, 2019 (https://plugins.jenkins.io/flaky-test-handler).

Muşlu, Kivanç, Bilge Soran, and Jochen Wuttke. 2011. “Finding Bugs by Isolating Unit Tests.” P. 496 in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE ’11*. New York, New York, USA: ACM Press.

Palomba, Fabio and Andy Zaidman. 2017. “Does Refactoring of Test Smells Induce Fixing Flaky Tests?” Pp. 1–12 in *Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017*. IEEE.

Pérez, Pablo Fuente. 2017. “Fullteaching: A Web Application to Make Teaching Online Easy.”

Pivotal Software. 2014. “Repeat (Spring Framework 5.1.8.RELEASE API).” Retrieved June 28, 2019 (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/Repeat.html).

Selenium HQ. 2019. “Selenium - Web Browser Automation.” Retrieved June 29, 2019 (https://www.seleniumhq.org/).

Thorve, Swapna, Chandani Sreshtha, and Na Meng. 2018. “An Empirical Study of Flaky Tests in Android Apps.” Pp. 534–38 in *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*. IEEE.

Vahabzadeh, Arash, Amin Milani Fard, and Ali Mesbah. 2015. “An Empirical Study of Bugs in Test Code.” Pp. 101–10 in *2015 IEEE 31st International Conference on Software Maintenance and Evolution, ICSME 2015 - Proceedings*. IEEE.

Wong, W. Eric, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. “A Survey on Software Fault Localization.” *IEEE Transactions on Software Engineering* 42(8):707–40.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS