FlakyLoc: Flakiness Localization for Reliable Test Suites in Web Applications

Jesús Morán[1,*], Cristian Augusto[1], Antonia Bertolino[2], Claudio De La Riva[1] and Javier Tuya[1]

[1]*Computer Science Department, University of Oviedo, Gijón, Spain*
[2]*ISTI-CNR, Consiglio Nazionale delle Ricerche, Pisa, Italy*
E-mail: moranjesus@uniovi.es; augustocristian@uniovi.es; antonia.bertolino@isti.cnr.it; claudio@uniovi.es; tuya@uniovi.es
[*]*Corresponding Author*

Received 19 December 2019; Accepted 14 April 2020;
Publication 03 June 2020

Abstract

Web application testing is a great challenge due to the management of complex asynchronous communications, the concurrency between the clients-servers, and the heterogeneity of resources employed. It is difficult to ensure that a test case is re-running in the same conditions because it can be executed in undesirable ways according to several environmental factors that are not easy to fine-grain control such as network bottlenecks, memory issues or screen resolution. These environmental factors can cause flakiness, which occurs when the same test case sometimes obtains one test outcome and other times another outcome in the same application due to the execution of environmental factors. The tester usually stops relying on flaky test cases because their outcome varies during the re-executions. To fix and reduce the flakiness it is very important to locate and understand which environmental factors cause the flakiness. This paper is focused on the localization of the root cause of flakiness in web applications based on the characterization of the different environmental factors that are not controlled during testing. The root cause of flakiness is located by means of spectrum-based localization techniques that analyse the test execution under different combinations of

Journal of Web Engineering, Vol. 19_2, 267–296.
doi: 10.13052/jwe1540-9589.1927

the environmental factors that can trigger the flakiness. This technique is evaluated with an educational web platform called FullTeaching. As a result, our technique was able to locate automatically the root cause of flakiness and provide enough information to both understand it and fix it.

Keywords: Software testing and debugging, spectrum-based localization, web applications, test flakiness.

1 Introduction

Software testing and debugging play an important role in the evaluation of software quality, but they pose several challenges [1]. 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 varying 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 case is considered flaky when the same test case run on the same system-under-test obtains different outcomes due to the environmental factors [2]. Testers cannot rely on the outcome of flaky test cases.

Avoiding flakiness is difficult because it does not always depend on the application, but on the behavior of these environments. The developer designs the test cases in one environment, but they can be executed in another. Even if the test cases are executed in the same environment they are prone to flakiness because the environment is not usually predictable or coarse-grained controllable. It is common that developers design the test cases to be executed several times to avoid non-predicted environmental issues. However, according to a recent study, developers face flakiness frequently and they usually stop relying on potentially flaky tests [3]. Despite the fact that debugging these tests is considered time-consuming, the majority of developers consider that finding the root cause of flakiness is relevant in order to fix it, but it is also a very difficult challenge [3].

In this paper, we introduce a technique to locate the root cause of flakiness in test cases for 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, a test case is executed several times under different environmental factors to obtain insights about flakiness. These executions are analysed with a spectrum-based localization

technique [4] considering that the factors that usually trigger the flakiness are more prone to be the root cause of flakiness.

This article extends our earlier work [5], improving the technique proposed and its evaluation, as well as expanding the survey of related work. The localization technique is enhanced to analyze combinatorial test executions with different ranking metrics like *Ochiai* and *Tarantula*. The evaluation of the technique is also extended providing insights not only of the localization but also about the fixing of flakiness in a real-world application. The related work is extended introducing a thorough analysis of the state-of-the-art.

The contributions of this article thus include:

1. A technique called FlakyLoc to locate the root cause of flakiness in web applications. This technique characterizes the test environment, executes the test case in different ways based on combinatorial testing, and analyzes the test execution using different ranking metrics.
2. The localization of the root cause of flakiness in a real-world web application using the technique proposed.
3. The mitigation and fixing of the flakiness in a real-world web application based on the information provided by the technique proposed.

The remainder of the paper is organized as follows. The testing of web applications is introduced in Section 2. The related work on flaky tests is discussed in Section 3. The technique FlakyLoc is introduced in Section 4 and a practical working example of this technique is described in Section 5. Finally, conclusions and future work are set out 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 to 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. There are several frameworks to support testing in web applications, among others NDT [6, 7]. The WebDrivers allow the automatization of the test cases controlling the user actions in a browser. Different tools exist to support the automatic execution of test cases for web applications, such as Selenium [8].

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 test cases. For example, suppose a simple test

case that pushes a button and waits 2 seconds to check if the server response is correct. The execution of the previous test case can be affected by several environmental factors such as the screen resolution, memory or network. These factors can cause flakiness in the test case, so that sometimes it passes and other times fails, as in the following examples. The test case passes when it is executed in large screen resolutions because it is able to find the button. In contrast, the test case can fail when it is executed in small screen resolutions because the button can be hidden automatically inside the response menu. The test case can also fail if the button is not rendered due to lack of memory. If the button is pushed correctly, the test case waits 2 seconds for the server response, however, the test case can also fail if the server employs more time due to network congestion. In the previous examples, the test case is flaky because the tester cannot rely on its outcome as sometimes the test case fails, and other times it passes.

The presence of a flaky test case is common [3], and some researchers propose the aphorism 'Assume Tests are Flaky' (ATAF) [9]. In order to deal with this flakiness, the testing tools usually provide different mechanisms based on re-execution. JUnit has the @RepeatedTest(10) tag that executes the test case 10 times to avoid "failures" due to the environmental factors of the execution [10]. In a similar way, the Spring framework has the @Repeat(10) tag [11]. For the case of progressive web applications, Android provides the @FlakyTest(tolerance=10) tag [12]. Maven also supports the re-execution of those test cases that fail using the Surefire plugin with the option -Dsurefire.rerunFailingTestsCount=10 [13]. Based on the abovementioned, Jenkins provides the Flaky Test Handler plugin [14].

The previous tools re-execute the flaky test case several times in order to check if the test case passes in at least one execution. However, the tester cannot rely on the test case as it is still flaky, and its execution is not easy to reproduce. In order to both avoid and fix the flakiness, the developers consider it very important to identify the root cause of flakiness [3]. In this paper, we introduce FlakyLoc to locate the root cause of flakiness in web applications.

3 Related Work

This section discusses different works that are related to FlakyLoc classified in the subsections below. Subsection 3.1 is focused on the studies that classify different kinds of flakiness instead of debugging the flakiness. The studies

on detecting the different kinds of flakiness are detailed in Subsection 3.2. The techniques to locate the root cause of this flakiness are described in Subsection 3.3. Finally, Subsection 3.4 introduces the studies on fixing the flakiness. The following subsections discuss the related work and both their similarities and differences with the FlakyLoc technique.

3.1 Classification of flakiness

The root causes of faults have been widely studied and several authors propose different taxonomies to classify them [15, 16]. Although flakiness in software testing is not a new problem [17], its interest has increased in the recent years. The first classification of flakiness analyzed 51 open source projects [2], classifying the flakiness into 11 different categories: asynchronous waits, concurrency, test order dependency, resource leak, network, time, IO, randomness, floating-point operations, unordered collections and others. Most flakiness is caused by asynchronous waits [2], as for example when the Selenium WebDriver sends an asynchronous web request and does not wait enough time for the server response. Thorve et al. [18], after analyzing 29 Android Applications, extend the classification of flakiness with the following three categories: Dependency, Program Logic, and UI. All of these kinds of flakiness can happen also in web applications, especially the Dependency and UI. Dependency flakiness is caused by the use of specific hardware, devices or third party libraries. UI flakiness is caused by the misunderstanding of the rendering process and user interface. Eck et al. [3] also extend the flakiness classification with the following four new categories after analyzing the Bugzilla reports: Restrictive Range, Test Case Timeout, Platform Dependency, and Test Suite Timeout. These kinds of flakiness can also happen in web applications, especially Test Case Timeout and Platform Dependency. The Test Case Timeout flakiness is caused when the test case does not finish in the correct time and it is killed. Platform Dependency flakiness is caused when the test case passes in one platform, but it fails in another, such as for example those test cases that pass in one version of the browser, yet fail in another.

The previous studies are focused on classifying the kinds of flakiness instead of debugging them. In FlakyLoc, we propose to debug the flaky test cases with the aim of locating the root cause of flakiness of these kinds of flakiness. Based on these studies of this subsection, we characterize a series of environmental factors that are prone to trigger flakiness in web applications.

3.2 Detection of a flaky test

The interest in the literature about test flakiness begins with the so-called "false alarm" tests. The false alarm tests are those that indicate failure but there is no fault in the code. Several works have addressed the detection of these false alarm tests. Herzing and Nagappan [19] aimed to detect them using association rules that classify the test case in real faults or false alarms based on a series of test execution parameters. This work has been put into production in the Microsoft continuous integration system, achieving savings of 1.7 hours per day in development velocity [19].

Jiang et al. [20] also classify the test cases at Huawei proposing to analyze the tests logs with an NLP technique that classifies the cause of flakiness between product code, configuration error, test script defect, among others.

Flaky test cases and false alarm tests are sometimes referred to interchangeably in the literature. Several authors argue that flaky tests are prevalent in practice [21]. The common way to detect whether a test case is flaky is to re-execute it several times until different outcomes are detected when the test case is executed under similar conditions. However, some researchers propose different approaches. Palomba and Zaidman [22] studied the relationship between flakiness and code smells, concluding that the flakiness of 54% of flaky test cases can be attributed to code smells. Muslu et al. [23] proposed isolating the execution of each test case to detect problems related to dependencies. Bell et al. [24] proposed detecting the flakiness when the same test case in two executions covers the same code of the system-under-test but in one execution passes and in the other execution fails.

The technique proposed in our paper, FlakyLoc, is not focused on the detection of flaky test cases, but on the localization of the root cause of flakiness once the flakiness is detected. Notwithstanding, FlakyLoc executes the test case in different environmental characteristics to analyze the pattern of characteristics in which the flakiness is detected.

3.3 Localization of the root cause of flakiness

Lam et al. [25] propose to classify the category of flakiness by analyzing the logs after several test executions and locating the suspicious lines of code that trigger flakiness. The previous technique and our paper are orthogonal because both techniques aim to improve the understanding of the flakiness but provide complementary insights about the root cause of flakiness. Some authors [26] have also proposed to detect Order and Non-Order dependent flakiness with a tool called iDFlakies. These tools aim to change the execution

order of the test suite in order to discover underlying dependencies between the test cases. The technique proposed in our paper is not only focused on the flakiness caused by order dependencies but also on localizing more types of root causes of flakiness such as those presented in the previous sections.

Our technique, FlakyLoc, instead of providing the category of flakiness, the line of code or the order that triggers the flakiness, provides the suspicious environmental factors that cause the flakiness. These environmental factors are obtained by FlakyLoc based on both the characterization and analysis of several executions through a spectrum-based localization and combinatorial testing.

3.4 Fixing the flakiness

Several authors have proposed to fix or decrease the undesirable effects of the test flakiness in the test suites. Some approaches [27][28] isolate the flaky test cases into a quarantine subset that is executed after the whole test suite execution to provide extra insights without stopping the continuous production cycle. Lam et al. also shed light on how these test cases interact with each other and provide the correct way to fix the flakiness [29]. In his PhD dissertation, Gao [30] proposes a test flakiness filter that reaches a tradeoff between the minimization of the flakiness effects and real failure detection.

The technique proposed in or our paper, FlakyLoc, does not fix flakiness. However, FlakyLoc is aimed to help the developer understand the root cause of flakiness, giving statistical insights that can also provide valuable information to fix or decrease the flakiness.

4 FlakyLoc: Localization of the Root Cause of Flakiness

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

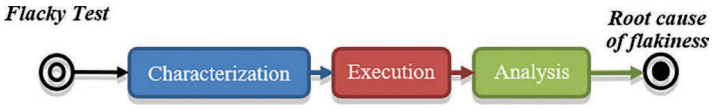

Figure 1 Technique to locate the flakiness.

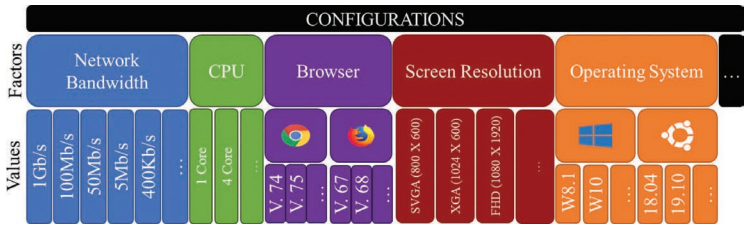

Figure 2 Model of the configurations with several characteristics.

The proposed technique, FlakyLoc, 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). FlakyLoc executes the flaky test case in different configurations selected with a combinatorial approach (Execution). The root cause of the flakiness is then automatically located by a spectrum-based localization technique that analyses which 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 combinations of configurations, and analysis of the root cause of flakiness.

4.1 Characterization

We characterize the configuration that triggers the flakiness according to the potential environmental factors that can cause the flakiness. A catalog of these environmental factors can be collected from the state-of-the-art studies about the classification of flakiness and considering the specific-domain of the application (Subsection 3.1). For example, in web applications, a configuration can be characterized according to the following factors (also depicted in Figure 2):

- Memory can cause issues in the WebDrivers, especially when several sessions and browsers are not properly closed and they consume the same memory.

- The network is one of the main causes of flakiness [2] 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 [2].
- Browsers and different versions of these browsers can alter the execution of the test cases, causing flakiness for different reasons such as rendering the objects in a different way.
- Screen resolution can modify the test execution, especially for those interactive 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 configurations are modelled according to the factors and the values that these factors take. Thus, each configuration is composed of several factor-value pairs. For example, a configuration can be composed by 400KB/s as network bandwidth (Network bandwidth – 400KB/s pair), 1 core CPU (CPU – 1 core pair), Chrome v75 (Browser – Chrome v75 pair), SVGA screen resolution (Screen resolution – SVGA pair), and Windows 10 (Operating system. – Windows 10 pair).

4.2 Execution

The same test case can be executed in different ways according to the combination of the previous characterization, some of which cause flakiness while others hide flakiness. FlakyLoc proposes to execute the same flaky test case under different representative configurations using a combinatorial approach [31, 32]. Testing all combinations of the environmental factors may be inefficient because the number of configurations grows exponentially according to the number of factors-values. All combinations of the environmental factors represented in Figure 2 require to execute the test case in at least 64 configurations.

However, combinatorial approaches such as t-wise (also known as t-way) can be used to obtain a representative subset of combinations. T-wise proposes to test only all combinations of each t environmental factors [33]. Based on this approach, 1-Wise (also known as each use) [34] proposes that all values of each environmental factor appear in at least one configuration, whereas 2-Wise (also known as pairwise) proposes that the combination of all values per pair of environmental factors appears in at least one configuration.

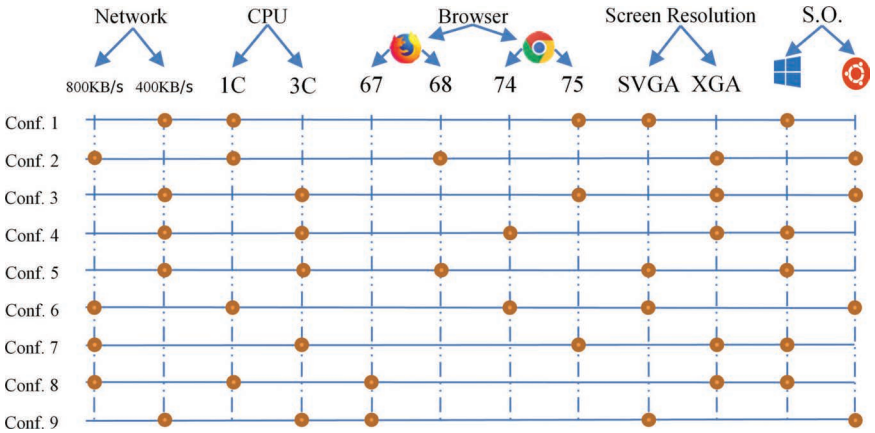

Figure 3 2-Wise configurations of network, CPU, browser, Screen resolution and S.O. factors.

The 2-Wise approach is almost as effective as all combinations in software testing [35], but employs much fewer resources in terms of time and cost [36]. Therefore, the FlakyLoc proposes to execute the test case in 2-Wise combinations of the environmental factors. For the environmental factors represented in Figure 2, FlakyLoc proposes to execute the test case in the configurations represented in Figure 3. These 9 configurations cover 2-Wise because all combinations of each pair of environmental factors (Network, CPU, Browser, Screen resolution and Operating system) are executed in at least one configuration.

In the example of Figure 3, the executions of the test case in the 9 configurations with 2-Wise combinatorial approach provides insights into the root cause of flakiness, especially those factors that usually trigger the flakiness. The test case executed in Configurations 2, 6, 7 and 9 succeeds, but on the other hand, the same test case executed in Configurations 1, 3, 4 and 5 triggers a "failure" because the test case cannot perform the user interactions due to the lack of the web elements required.

The environmental factors of Configurations 1, 3, 4 and 5 cause flakiness whereas the factors of Configurations 2, 6, 7 and 9 hide the flakiness. Some configurations trigger the "failure" with 400KB/s (Configuration 1, 3, 4 and 5), they do not provide enough insights into the root cause of flakiness because other configurations with 400KB/s mask the flakiness (Configuration 9). The same happens with the remainder environmental factors because the test executions trigger the flakiness in a non-deterministic way without

an apparent clear pattern. However, the test executions provide evidence about the most suspicious environmental factor that causes the flakiness. This evidence is analysed systematically with the following approach based on the fault localization techniques and statistical rankings of suspiciousness.

4.3 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 [37, 38], in FlakyLoc the ranking metrics analyse the factors that cause flakiness.

The ranking metrics analyse the similarity between the values of the factors executed and the configurations that fail/hide the flakiness. The environmental factors that are executed in the configurations that trigger the flakiness are more suspicious than those executed in the configurations that do not trigger the flakiness. In contrast, the environmental factors not executed in the configurations that trigger the flakiness are less suspicious than those not executed in the configurations that do not trigger the flakiness. There are different ways to obtain the suspiciousness based on the above, and the ranking metrics use different weights to obtain the suspiciousness per each environmental factor based on the following:

- N_{CF} is the number of configurations that execute the environmental factor and trigger the flakiness.
- N_{CS} is the number of configurations that execute the environmental factor but do not trigger the flakiness.
- N_F is the number of configurations that trigger the flakiness.
- N_S is the number of configurations that do not trigger the flakiness.

FlackyLoc uses the *Ochiai* [39] and *Tarantula* [40] ranking metrics that calculate the suspiciousness per each environmental factor in the following way:

- *Ochiai:* $\frac{N_{CF}}{\sqrt{N_F \cdot (N_{CF}+N_{CS})}}$
- *Tarantula:* $\frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F}+\frac{N_{CS}}{N_S}}$

In the example of Figure 3, the 9 configurations are analyzed with the *Ochiai* and *Tarantula* ranking metrics (Table 1). Both ranking metrics obtain automatically that the most suspicious root cause of flakiness is 400KB/s of network bandwidth.

Table 1 Localization of the root cause of flakiness with Ochiai and Tarantula ranking metrics

Environmental factors		Configurations									Ochiai		Tarantula	
		1	2	3	4	5	6	7	8	9	Suspiciousness	Ranking	Suspiciousness	Ranking
Network bandwidth	5 MB/s		X				X	X	X		0	11	0	11
	400 KB/s	**X**		**X**	**X**	**X**				**X**	**0.894**	**1**	**0.833**	**1**
CPU	1 core	X	X				X		X		0.25	9	0.294	9
	4 cores			X	X	X		X		X	0.671	2	0.652	3
Browser	Firefox v67								X	X	0	11	0	11
	Firefox v68		X			X					0.354	7	0.556	5
	Chrome v74				X		X				0.354	7	0.556	5
	Chrome v75	X		X				X			0.577	4	0.714	2
Screen resolution	SVGA	X				X	X			X	0.5	5	0.556	5
	XGA		X	X	X			X	X		0.447	6	0.455	8
Operating sytem	Windows 10	X			X	X		X	X		0.671	2	0.652	3
	Ubuntu 18.04		X	X			X			X	0.25	9	0.294	9
	Failures	X		X	X	X								

Despite the fact that the failure is triggered with the Windows 10 operating system in Configurations 1, 4 and 5, apparently this is not the root cause of flakiness because Windows 10 also hides the flakiness in Configurations 7 and 8. The same happens with the remainder of the environmental factors. The Firefox v67 browser is not the root cause of flakiness because it never triggers the flakiness. In contrast, 400KB/s of network bandwidth triggers the flakiness most of the times. After analyzing automatically all factors through the localization technique, both Ochiai and Tarantula determine statistically that the most suspicious root cause of flakiness is 400KB/s of network bandwidth. According to *Ochiai* ranking metric, the top of the rank of suspiciousness is 400KB/s (0.894 out of 1 of suspiciousness), followed by both 4 cores and Windows 10 (0.714 out of 1 of suspiciousness). The *Tarantula* ranking metric also determines 400Kb/s of bandwidth network as the most suspicious (0.833 of 1 of suspiciousness), followed by the Chrome v75 browser with 0.714 of suspiciousness.

The localization of 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 succeeds with 5Mb/s of network bandwidth because the web requests are responded to quickly just before the user interaction takes place. However, with less network bandwidth (400 KB/s), the web requests are responded to slowly, causing that the test case fails because it tries to execute the user interactions before the responses. This flakiness can be avoided in different ways, for example increasing the time of sleep or wait for to wait for the web responses.

5 Evaluation

In this section, we evaluate how FlakyLoc is able to locate the root cause of flakiness in a case study about an educational web platform called Full-Teaching [41]. This platform allows teachers and students to perform the lessons and share their teaching materials, such as calendars, dashboards or forums. The Fullteaching 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 case sometimes passes and others fail in a non-deterministic way. In the remainder of this section, we evaluate if FlakyLoc is able to both locate the root cause of flakiness and provide valuable information to fix a flaky test case. The evaluation considers the first environmental factors provided by the Ochiai and Tarantula ranking metrics.

We consider a test case that checks if the user is able to log into the application, get into the settings menu and logout. Although the test cases are executed in an isolated environment through a containerized instance, the test case sometimes crashes due to the configuration in which the test case is executed. This test case was correctly executed in the tester's computer, but the same test case failed in the Continuous Integration server. In both environments, the test case was executed isolated inside a container with the same resources. We have 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, FlakyLoc, is applied to the previous flaky test.

5.1 Characterization

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

1. Memory: the test execution is modelled with 90MB and 240MB to increase or decrease the WebDriver resources.
2. CPU: the execution is modelled with 1 and 4 cores to increase or decrease the concurrency between the threads executed by the test case.
3. Browser: the execution is modelled with Mozilla Firefox and Google Chrome that can render the web elements in different ways.
4. Screen resolution: the execution is modelled with SVGA (800×600), XGA (1024×768), and WFHD (2560×1024) resolutions. These resolutions can increase or decrease the web elements that are rendered in the navigator window.

5.2 Execution

A combination of the previous factor-values characterizes the execution of the test case. We execute the test case with the 6 configurations of Figure 4 obtained by the 2-wise combination of the aforementioned environmental factors. These 6 configurations guarantee that all combinations of each pair of environmental factors are executed at least once.

After the execution of the test case in the previous 6 configurations, the test case fails 50% of times (Configurations 1, 2 and 4) and masks the "failure" in another 50% of times (Configurations 3, 5 and 6) without an apparent clear pattern. The test case executed with 90MB of memory sometimes fails

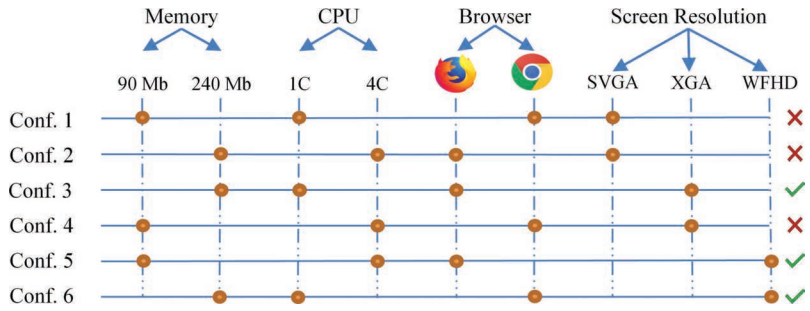

Figure 4 Configurations executed in FullTeaching application.

(Configurations 1 and 4) whereas other times it succeeds (Configuration 5). Increasing the memory to 240MB still makes the test case sometimes fail (Configuration 2) and other times succeed (Configurations 3 and 6). The test case fails more times with 90MB than with 240MB. However, there is no clear evidence that memory size causes flakiness, and the same happens with the remaining environmental factors. The test case fails with 1 core sometimes (Configuration 1), but the test executions that increase the CPU to 4 cores still fail sometimes (Configurations 2 and 4). The same happens with the browser, the test case fails sometimes with Firefox (Configuration 2), but apparently, the browser is not the root cause of flakiness because the test case also fails with Chrome browser (Configurations 1 and 4). The same happens with the screen resolution because the test case sometimes fails in 800x600 (Configurations 1 and 2) and other times fails in 1024x768 (Configuration 4). Apparently, the screen resolution is not the root cause of flakiness because the test case executed with the same screen resolution sometimes fails and other times succeeds: the test case fails with 1024x768 in Configuration 4 and succeeds in Configuration 3. Therefore, it is difficult to obtain strong clues about the root cause of flakiness analyzing the test executions by hand.

All of the failures produce the following trace:

```
Expected condition failed: waiting for visibility
of element located by By.id: settings-button
(tried for 10 second(s) with 500 MILLISECONDS
interval)
```

This error trace can be caused by timeouts (“Test case timeouts” according to the Eck et al. [3] classification) because of issues in the network or processing, among others. However, this is misleading, and in fact the root cause of flakiness is not related to timeouts as the following subsection details.

5.3 Analysis

We apply the FlakyLoc technique to locate automatically the root cause of the flakiness analysing the test executions. FlakyLoc employs a ranking metric to analyse the previous 6 test executions considering statistically those factors both covered (marked with an "X" in Table 2) and non-covered when a test case fails (marked with an "X" in the bottom row of Table 2), and also when it succeeds. In this subsection, we analyse the test executions with FlakyLoc using the Ochiai and Tarantula ranking metrics that are often used in the localization of root causes.

Table 2 details the most suspicious environmental factors obtained by the FlakyLoc technique. Regardless of the ranking metric used (*Ochiai* or *Tarantula*), the most suspicious environmental factor ranked in the first position is the screen resolution of 800×600 (0.816 of suspiciousness in *Ochiai* and 1 in *Tarantula*), followed by the Chrome browser, 4 cores of CPU and 90MB of memory that are ranked in the second position (0.667 of suspiciousness in both *Ochiai* and *Tarantula*).

The localization of the root cause of flakiness (800x600 screen resolution) is valuable to understand the flakiness in order to avoid it or fix it. In the FullTeaching application, the flaky failure was triggered sometimes in the Continuous Integration server but masked in the tester computer. Once the root cause of flakiness is located, we are able to understand that the tester computer masked the flakiness because it has a widescreen resolution, whereas the Continuous Integration server sometimes triggers the flaky failure because it isolates the test case in a container with low screen resolutions.

The analysed test case aims to check the setting configuration of the FullTeaching application. During the test execution, the Selenium WebDriver pushes a "SETTING" button to enter the setting configuration and finally checks that the settings are fine. Once the root cause of flakiness is located, we can understand that in computers with wide resolutions like the tester computer, the SETTING button is visible and the test case checks the settings properly as in Figure 5 (2560x1080 screen resolution). However, we can understand that in computers with low screen resolution as sometimes happens in Continuous Integration deployment, the SETTING button is not visible because it is hidden inside the response menu as in Figure 6 (800x600 screen resolution).

Once FlakyLoc determines automatically that the low screen resolution causes the flakiness, we have enough clues to understand the flakiness.

Table 2 Localization of the root cause of flakiness in FullTeaching application

Environmental factors		Configurations						Ochiai		Tarantula	
		1	2	3	4	5	6	Suspiciousness	Ranking	Suspiciousness	Ranking
Memory	90MB	X			X	X		0.667	2	0.667	2
	240MB		X	X			X	0.333	6	0.333	6
CPU	1 core	X		X			X	0.333	6	0.333	6
	4 cores		X		X	X		0.667	2	0.667	2
Browser	Firefox		X	X		X		0.333	6	0.333	6
	Chrome	X			X		X	0.667	2	0.667	2
Screen resolution	**800x600**	**X**	**X**					**0.816**	**1**	**1**	**1**
	1024x768			X	X			0.408	5	0.5	5
	2560x1024					X	X	0	9	0	9
	Failures	X	X		X						

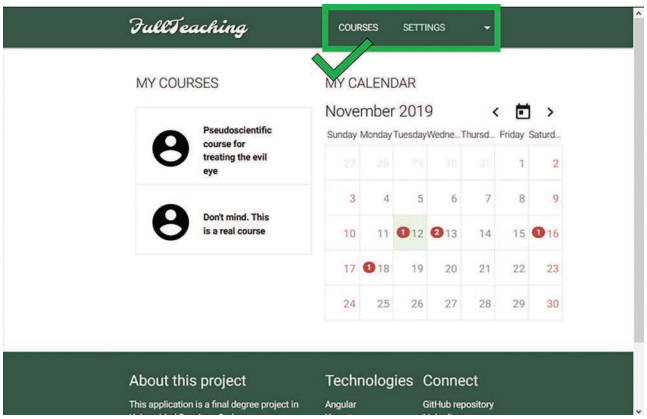

Figure 5 Test case executed in 2560 × 1080 screen resolution.

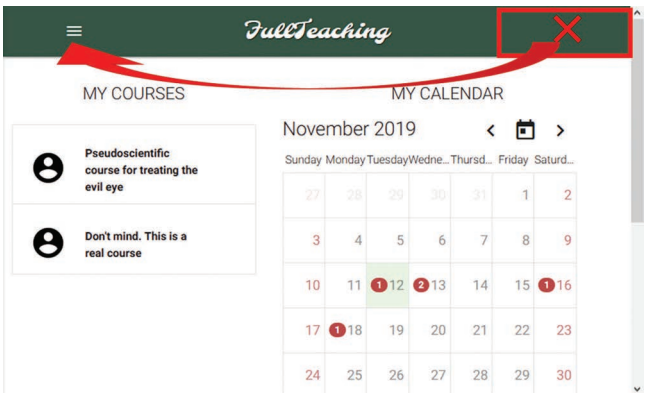

Figure 6 Test case executed in 800 × 600 screen resolution.

However, the test case fails sometimes in 1024x768 screen resolution (Configuration 4) and other times succeeds (Configuration 3) depending on the browser. The test case succeeds in 1024x768 screen resolution with Firefox browser because the SETTING button is visible (Figure 7), but fails in Chrome browser because the web elements are rendered in a different way and the button is again hidden inside the response menu (Figure 8). Although the test case aims to be executed inside a container deployed on Docker during Continuous Integration, the test case succeeds or fails depending mainly on screen resolution and also on the browser. According to the taxonomy of flakiness proposed by Eck et al. [3], this flaky test case is considered 'Platform Dependent' on both browser and screen resolution.

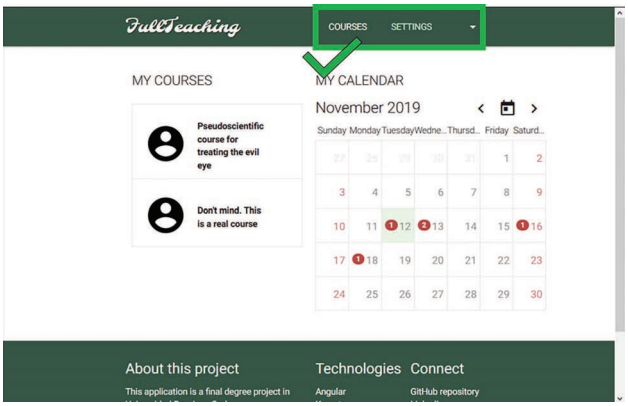

Figure 7 Test case executed in 1024×768 screen resolution with Firefox browser

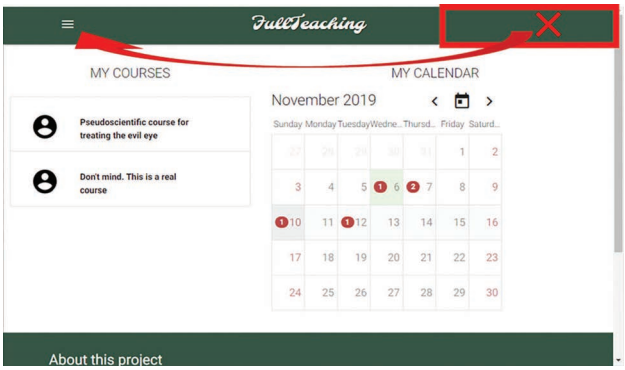

Figure 8 Test case executed in 1024×768 screen resolution with Chrome browser.

We have analysed how the web elements are rendered in both the screen and browser, observing that the browser window is not maximized. Therefore, the test case does not take advantage of the whole screen to display the buttons properly and sometimes places the buttons inside the response menu.

5.4 Fixing flakiness

In order to fix/decrease the flakiness, we modify the test case maximizing the window of the browser programmatically to avoid the platform dependency. After several executions with maximized windows, we observed that the test case has reduced the flakiness, but it is still flaky. We re-execute the 6 configurations obtained with the 2-Wise combinatorial approach (Figure 9).

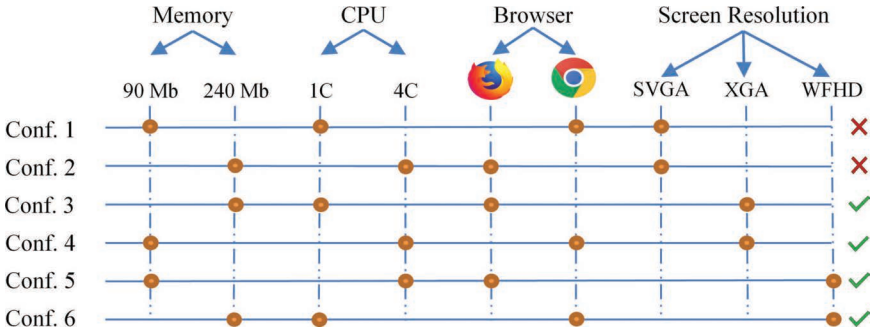

Figure 9 Configurations executed in the FullTeaching application using a browser with a maximized window.

The test case fails 33.33% of times (Configurations 1 and 2) and masks the "failure" in the other 66.66% of times (Configurations 3, 4, 5 and 6). Note that the programmatic maximization of the window decreases the flakiness from 50% (3 failures out of 6 in Figure 9) to 33.33% (2 failures out of 6 in Figure 9).

We can observe that the test case stops failing with 1024×768 screen resolution using the Chrome browser because the maximization of the browser window provides more space to place the web elements. However, the test case is still flaky because it fails in 800×600 screen resolution regardless of the browser or other environmental factors. We use again the FlakyLoc technique as Table 3 details. As we expected, FlakyLoc still pinpoints automatically that the root cause of flakiness is the 800×600 screen resolution. The test case fails in 800×600 screen resolution because the SETTING button is hidden inside the response menu and the test case does not find it.

The programmatic maximization of the browser windows removes the browser dependency (platform dependency [3]) of the test case because the test case stops failing in Chrome and succeeds in Firefox for 1024x768. However, the test case is still platform dependent on the screen resolution because the FlakyLoc indicates that the test case is just a little flaky and the root cause of flakiness is the 800x600 screen resolution.

In order to avoid the flakiness, we again modify the test case to force it programmatically to be deployed in a container with 2560x1080 screen resolution modifying the capabilities of the Docker deployment during Continuous Integration. We execute the test case several times and the flakiness disappears because the test case stops failing due to the browser or screen resolution issues. The test case is executed every time as Figure 10 depicts

Table 3 Localization of the root cause of flakiness in FullTeaching application

Environmental factors		Configurations						Ochiai		Tarantula	
		1	2	3	4	5	6	Suspiciousness	Ranking	Suspiciousness	Ranking
Memory	90MB	X			X	X		0.408	2	0.5	2
	240MB		X	X			X	0.408	2	0.5	2
CPU	1 core	X		X			X	0.408	2	0.5	2
	4 cores		X		X	X		0.408	2	0.5	2
Browser	Firefox		X	X		X		0.408	2	0.5	2
	Chrome	X			X		X	0.408	2	0.5	2
Screen resolution	**800x600**	**X**	**X**					**1**	**1**	**1**	**1**
	1024x768			X	X			0	8	0	8
	2560x1024					X	X	0	8	0	8
	Failures	X	X								

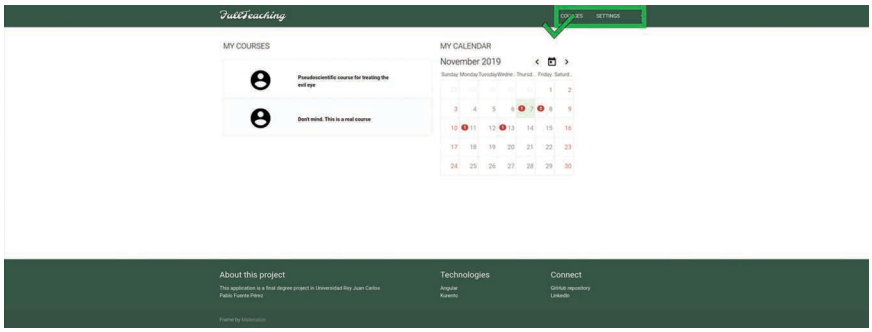

Figure 10 Test case executed using a Docker instance with a fixed 2560Œ1024 screen resolution.

with the SETTING button always visible. Therefore, the test case is able to push the SETTING button and finally it checks that the settings are fine.

6 Discussion and Threats to Validity

Before we used the FlakyLoc technique, we thought that flakiness was caused by timeouts (Test case timeouts [3]). After using FlakyLoc, we located the root cause of flakiness and understood that the flakiness is triggered due to a dependency on both browser and screen resolution (Platform dependency [3]) because sometimes a button is rendered in the main windows and other times inside the response menu. We fixed the platform dependency of the browser through the programmatic maximization of the browser window aimed to provide enough space to the browser to place the web elements. Once the platform dependency of the browser was fixed, the test case decreased the flakiness, but it was still failing in some screen resolutions (platform dependency of screen resolution). According to Figure 11 both the browser and the screen resolution are suspicious before maximizing the browser window, but after fixing the browser dependency only the screen resolution is suspicious. Finally, the platform dependency of the screen resolution is avoided modifying the test case to force its execution inside a Docker container with a fixed resolution. The FlakyLoc technique locates the root cause of flakiness of the test case, and it provides valuable information to fix/decrease the flakiness. However, fixing flakiness automatically is challenging because it depends on the environment executed.

There are several threats to validity. Firstly, regarding the ***internal threats*** to validity, the technique could yield different results because the execution

Figure 11 Suspiciouness of flakiness before fixing the test case.

of the same configuration could sometimes trigger flakiness and other times it could remain hidden due to non-determinism. To mitigate this problem, the technique guarantees that each characteristic is executed several times due to a combinatorial approach.

Secondly, regarding the ***external threats*** to validity, the first threat is that the technique was evaluated with only one case study. However, this case study is a real-world web application and in future work we plan to evaluate the technique with more kinds of flakiness. The second *external threat* is related to the generation of configurations because the technique could yield different outputs depending on the configurations executed. To mitigate this problem, the technique uses a combinatorial approach to select the configurations.

Finally, regarding the ***construct threats*** to validity, the technique can obtain different results depending on the ranking metric used. To mitigate this problem, the tehcnique obtains the results with two ranking metrics: Ochiai and Tarantula.

7 Conclusions and Future Work

The test cases of web applications can be executed differently depending on the environmental factors i.e. network bandwidth, memory or screen resolution. If the test case sometimes obtains one outcome and other times obtains another different outcome due to the environmental factors executed, then this test case is considered flaky. Flaky test cases reduce the reliability of the test suite because the tester stops relying on test outcomes. In spite of the fact that developers face frequently flaky test cases, it is difficult to both locate the root cause of flakiness and fix them. In this paper, we propose a technique called FlakyLoc to locate automatically the root cause of flakiness in web applications based on the characterization of the environmental

factors that make the test case more prone to be flaky. FlakyLoc executes the test case in different environmental factors through combinatorial testing and analyzes statistically each environmental factor with a spectrum-based approach obtaining a ranking of the suspicious root cause of flakiness.

We performed an evaluation of FlakyLoc in a web platform with a real flaky test case. FlakyLoc allowed the automatic detection of the root cause of flakiness and provided the appropriate insights to fix the flakiness. In conclusion, the characterization of the environmental factors together with the spectrum-based analysis of several test executions can locate automatically the root cause of flakiness of web applications and could provide valuable information to fix the flakiness.

In future work, we plan to evaluate FlakyLoc empirically in several web applications that have test suites with flaky test cases. We need to evaluate more extensively its effectiveness, properly identifying the cause of flakiness, but also to quantify the involved costs: in fact detecting flakiness is known as a very costly activity, and also locating the causes requires resources to re-execute the test cases under several configurations. The spectrum-based analysis is done automatically, but the execution of the configurations is controlled manually. In future work, we plan to also automate the executions, integrating the technique in the web methodologies with Model-Driven Engineering.

Acknowledgements

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under project TestEAMoS (TIN2016-76956-C3-1-R) and project POLOLAS (TIN2016-76956-C3-2-R), ERDF funds, and by the European Project ElasTest in the Horizon 2020 research and innovation program (GA No. 731535).

References

[1] A. Bertolino, “Software Testing Research: Achievements, Challenges, Dreams,” in 2007 Future of Soft. Eng., 2007, pp. 85–103.

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests,” in Proceedings of the ACM SIGSOFT Symposium on the Foundations of Soft. Eng., 2014, vol. 16-21-Nove, pp. 643–653.

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: the developer's perspective," in to appear FSE19/ESEC, 2019, pp. 830–840.

[4] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Transactions on Soft. Eng., vol. 42, no. 8. pp. 707–740, 2016.

[5] J. Morán, C. Augusto, A. Bertolino, C. de la Riva, and J. Tuya, "Debugging Flaky Tests on Web Applications," in Proceedings of the 15th Int. Conf. on Web Information Systems and Technologies, 2019, pp. 454–461.

[6] M. J. Escalona and G. Aragón, "NDT. A model-driven approach for web requirements," IEEE Trans. Softw. Eng., vol. 34, no. 3, pp. 377–394, May 2008.

[7] J. A. García-García, M. J. Escalona, F. J. Domínguez-Mayo, and A. Salido, "NDT-Suite: A Methodological Tool Solution in the Model-Driven Engineering Paradigm," J. Softw. Eng. Appl., vol. 07, no. 04, pp. 206–217, 2014.

[8] Selenium HQ, "Selenium – Web Browser Automation," 2019. [Online]. Available: https://www.seleniumhq.org/. [Accessed: 29-Jun-2019].

[9] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in Proceedings – 18th IEEE Int. Working Conf. on Source Code Analysis and Manipulation, 2018, pp. 1–23.

[10] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, and C. Stein, "RepeatedTest (JUnit 5.2.0 API)," 2019. [Online]. Available: https://junit.org/junit5/docs/5.2.0/api/org/junit/jupiter/api/RepeatedTest.html. [Accessed: 29-Jun-2019].

[11] Pivotal Software, "Repeat (Spring Framework 5.1.8.RELEASE API)," 2014. [Online]. Available: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/Repeat.html. [Accessed: 28-Jun-2019].

[12] Google, "FlakyTest |Android Developers," 2019. [Online]. Available: https://developer.android.com/reference/android/support/test/filters/FlakyTest.html. [Accessed: 28-Jun-2019].

[13] F. Apache Software, "Maven Surefire Plugin – Rerun failing tests," 2018. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html. [Accessed: 29-Jun-2019].

[14] Q. Luo and J. Micco, "Flaky Test Handler v1.04," 2015. [Online]. Available: https://plugins.jenkins.io/flaky-test-handler. [Accessed: 29-Jun-2019].
[15] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," J. Syst. Softw., vol. 152, pp. 165–181, Jul. 2019.
[16] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, B. K. Ray, and D. S. Moebus, "Orthogonal Defect Classification—A Concept for In-Process Measurements," IEEE Trans. Softw. Eng., vol. 18, no. 11, pp. 943–956, 1992.
[17] Google, "Google Testing Blog: TotT: Avoiding Flakey Tests," 2008. [Online]. Available: https://testing.googleblog.com/2008/04/tott-avoiding-flakey-tests.html. [Accessed: 02-Nov-2019].
[18] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in Proceedings – 2018 IEEE Int.Conf.on Soft. Maintenance and Evolution, 2018, pp. 534–538.
[19] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms Using Association Rules," in Proc. – Int. Conf. on Soft. Eng., 2015, vol. 2, pp. 39–48.
[20] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing," in Proceedings – 2017 IEEE/ACM 39th Int. Conf. on Soft. Eng., 2017, pp. 712–723.
[21] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in Proceeding – 2015 IEEE 31st International Conf. on Soft. Maintenance and Evolution, 2015, pp. 101–110.
[22] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?," in Proceedings – 2017 IEEE Int. Conf. on Soft. Maintenance and Evolution, ICSME 2017, 2017, pp. 1–12.
[23] K. Mužlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in SIGSOFT/FSE 2011 – Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Soft. Eng., 2011, pp. 496–499.
[24] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in Proceedings of the 40th Int. Conf. on Soft. Eng.- ICSE '18, 2018, pp. 433–444.
[25] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in Proceedings of the 28th ACM SIGSOFT Int. Symposium on Soft. Testing and Analysis, 2019, pp. 101–111.

[26] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "IDFlakies: A framework for detecting and partially classifying flaky tests," in Proceedings – 2019 IEEE 12th Int. Conf. on Soft. Testing, Verification and Validation, 2019, pp. 312–322.

[27] M. Fowler, "Eradicating Non-Determinism in Tests," Martin Fowler Personal Blog, 2011. [Online]. Available: https://martinfowler.com/articles/nonDeterminism.html. [Accessed: 11-Nov-2019].

[28] J. Micco, "Flaky Tests at Google and How We Mitigate Them," Google Testing Blog. p. 4, 2016.

[29] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: a framework for automatically fixing order-dependent flaky tests," in Proceedings of the 2019 27th ACM Joint Meeting on Eur. Soft. Eng. Conf. and Symposium on the Foundations of Soft. Eng., 2019, pp. 545–555.

[30] Z. Gao, "Quantifying Flakiness and Minimizing Its Effects on Software Testing," University of Maryland, 2017.

[31] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," Softw. Test. Verif. Reliab., vol. 15, no. 3, pp. 167–199, Sep. 2005.

[32] C. Nie and H. Leung, "A survey of combinatorial testing," ACM Comput. Surv., vol. 43, no. 2, pp. 1–29, Jan. 2011.

[33] ISO/IEC/IEEE, "29119-4:2015 -ISO/IEC/IEEE International Standard for Software and systems engineering —Software testing — TR-2017-35316 Part 4: Test techniques." pp. 1–150, 2015.

[34] P. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing," in COMPASS – Proceedings of the Annual Conf, on Computer Assurance, 1994, pp. 69–79.

[35] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in Proceedings – 27th Annual NASA Goddard / IEEE Soft. Eng. Work., 2003, pp. 91–95.

[36] J. Huller, "Reducing Time to Market With Combinatorial Design Method Testing," Int. Council on Systems Eng. (INCOSE) Conf. 2000.

[37] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An Empirical Investigation of Program Spectra," SIGPLAN Not. (ACM Spec. Interes. Gr. Program. Lang., vol. 33, no. 7, pp. 83–90, 1998.

[38] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "Empirical investigation of the relationship between spectra differences and regression faults," Softw. Test. Verif. Reliab., vol. 10, no. 3, pp. 171–194, Sep. 2000.

[39] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization," in Proceedings – Testing: Academic and Industrial Conf. Practice and Research Techniques, TAIC PART-Mutation 2007, 2007, pp. 89–98.
[40] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in 20th IEEE/ACM Int. Conf. on Automated Soft. Eng., 2005, pp. 273–282.
[41] P. F. Pérez, "Fullteaching: A web application to make teaching online easy." Universidad Rey Juan Carlos, 2017.

Biographies

Jesús Morán received the Ph.D. degree in computing from the University of Oviedo, Spain, in 2019. He is a Lecturer of the Computer Science Department with the University of Oviedo, Spain. He is a member of the Software Engineering Research Group. His research interests include software testing, Big Data technologies, and distributed programming.

Cristian Augusto received the degree in Computer Science in Information Technology from the University of Oviedo, Gijon, Spain in 2018. He is currently finishing his master's degree in Computer Engineering into Oviedo

University. His interest research areas in the field of Software Engineering are Big Data, privacy-preserving techniques and Software Testing mainly focused on the efficient use of resources in the test process. He has also been part since 2018 of the Software Engineering Research Group (GIIS) at the Oviedo University.

Antonia Bertolino received the M.S. degree in electronic engineering from the University of Pisa, Pisa, Italy, in 1985. She is a Research Director with the Italian National Research Council–Institute of Information Science and Technologies (ISTI), Pisa, Italy. Her research focuses on software and service testing. Ms. Bertolino is an Associate Editor for Transactions on Software Engineering and Methodology, Empirical Software Engineering Journal, and Journal of Software: Evolution and Process. She also serves as Senior Editor for the Journal of Systems and Software. She has been the General Chair of the 2015 International Conference on Software Engineering, Florence, Italy.

Claudio De La Riva received the Ph.D degree in computing from the University of Oviedo, Spain, in 2004. He is an Associate Professor of the Computer Science Department with the University of Oviedo, Spain. He is a member of the Software Engineering Research Group. His research interests include software verification and validation, software quality and software testing, mainly focused on testing database applications and services.

Javier Tuya received the Ph.D. degree in engineering from the University of Oviedo, Oviedo, Spain, in 1995. He is a Professor with the University of Oviedo, Oviedo, Spain, where he is the Research Leader of the Software Engineering Research Group. He is the Director of the Indra-Uniovi Chair, a member of the ISO/IEC JTC1/SC7/WG26 Working Group for the recent ISO/IEC/IEEE 29119 Software Testing Standard, and a Convener of the corresponding UNE National Body Working Group. His research interests in software engineering include verification and validation, and software testing for database applications and services.