Visual Web Test Repair

Andrea Stocco
University of British Columbia
Vancouver, BC, Canada
astocco@ece.ubc.ca

Rahulkrishna Yandrapally
University of British Columbia
Vancouver, BC, Canada
rahulky@ece.ubc.ca

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

Web tests are prone to break frequently as the application under test evolves, causing much maintenance effort in practice. To detect the root causes of a test breakage, developers typically inspect the test's interactions with the application through the GUI. Existing automated test repair techniques focus instead on the code and entirely ignore visual aspects of the application. We propose a test repair technique that is informed by a visual analysis of the application. Our approach captures relevant visual information from tests execution and analyzes them through a fast image processing pipeline to visually validate test cases as they re-executed for regression purposes. Then, it reports the occurrences of breakages and potential fixes to the testers. Our approach is also equipped with a local crawling mechanism to handle non-trivial breakage scenarios such as the ones that require to repair the test's workflow. We implemented our approach in a tool called Vista. Our empirical evaluation on 2,672 test cases spanning 86 releases of four web applications shows that Vista is able to repair, on average, 81% of the breakages, a 41% increment with respect to existing techniques.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

web testing, test repair, computer vision, image analysis

ACM Reference Format:
Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236063

1 INTRODUCTION

Test automation techniques are used to enable end-to-end (E2E) functional testing of web applications [57]. In this context, the tester verifies the correct functioning of the application under test (AUT) by means of automated test scripts. Such scripts automate the set of manual operations that the end user would perform on the web application's graphical user interface (GUI), such as delivering events with clicks, or filling in forms, and they are typically used for regression testing [6, 19, 21, 43, 45].

Despite their wide adoption, E2E tests are known to be fragile. Changes as simple as repositioning GUI elements on the page or altering the selections in a drop-down list can cause the test to break. In literature, instances of these problems are known as *test breakages* [15–17, 24]. A test breakage is defined as the event that occurs when the test raises exceptions or errors that do not pertain to the presence of a bug or a malfunction of the application under test. This is different from cases in which tests expose *failures*, meaning they raise exceptions which signal the presence of one or more bugs in the production code. In the latter case, the developer is required to correct the application, whereas in the former case, the tester must find a fix for the broken test.

Researchers have categorized breakages happening as test suites for web applications are maintained [24]. Web element *locators* have emerged as the main cause of fragility of tests, confirming previous anecdotal findings [17, 30].

Existing automated web test repair techniques [11, 23] essentially consider only the DOM of the web page as a source where to find possible repairs. Thus, these techniques are, in many cases, either unable to correct breakages, or produce many false positives. Moreover, breakages do not always occur at the precise point in which the test execution stops, which renders automated repair even more challenging. To detect their root causes, testers manually inspect the tests' interaction with the application, carefully verifying the action performed by each test step on the GUI, which is tedious and time-consuming.

In this paper, we propose a novel test repair algorithm, implemented in a tool named Vista, that leverages the visual information extracted through the execution of test cases, along with image processing and crawling techniques, to support automated repair of web test breakages. The key insight behind our approach is that the manual actions and reasoning that testers perform while searching for repairs can be automated to a large extent by leveraging and combining differential testing, computer vision, and local crawling.

Vista performs visual online monitoring capturing visual snapshots of the correct execution of the test cases. When the tests are replayed on a new version of the application, it uses the web elements' visual appearance to validate each test statement prior to their execution. On the occurrence of a breakage, Vista triggers a series of repair heuristics on the fly, to find potential fixes to report to the user. Upon inspection, the tester can confirm or discard the suggestions. In addition, using an automated local visual-based crawl exploration mechanism, Vista is able to handle complex breakage scenarios, such as those that break the test's execution workflow, e.g., when elements are moved to another page, or a new web page is added in between test steps.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236063

We evaluated Vista on a benchmark of 733 individual breakages from 2,672 test cases spanning 82 releases of four open source web applications. For each of them, we documented the type of breakage, and the position of the associated repair, resulting in a taxonomy of test breakages in web applications from a repair-oriented viewpoint.

Our paper makes the following contributions:

- The first repair-oriented taxonomy of test breakages in web applications.
- An algorithm for visually monitoring and validating web test cases, based on a fast image processing pipeline.
- A novel approach for repairing broken test cases using visual analysis and crawling.
- An implementation of our algorithm in a tool named Vista, which is publicly available [58].
- An empirical evaluation of Vista in repairing the breakages found in our study. Vista was able to provide correct repairs for 81% of breakages, with a 41% increment over an existing DOM-based state-of-the-art approach.

2 A WEB TEST BREAKAGE TRAVELOGUE

E2E web tests are known for being fragile in the face of software evolution [24, 30]. Even a minor change in the DOM or GUI might break a previously developed test case. The test would then need to be repaired manually to match the updated version of the application, even if conceptually the functionality is unaltered, and no errors are present in the application code.

Characterization of a Breakage. In order to clarify the scope of our work, it is important to emphasize the difference between *test breakages* and *test failures*. We consider a *test breakage* as the event that occurs when a test T_n that was used to work and pass on a certain version V_n, fails to be applicable to a version V_{n+k} ($k \geq 1$) due to changes in the application code that interrupt its execution unpremeditatedly. This is different from cases when tests desirably expose a program *failure* and hence do something for which they have been designed (i.e., exposing regression faults). In this paper, we focus our analysis on test breakages.

Study of Breakages. In a recent study, researchers have categorized breakages as test suites for web applications were evolved [24]. The study shows that web element *locators* are the main cause of fragility (74% of the totality of breakages).

Indeed, the mapping between locators and DOM elements is massively affected by structural changes of the web page, that may render tests inapplicable, just because the locators (and not the tests themselves) become ineffective [11, 24, 30]. The addition or repositioning of elements within the DOM can in fact cause locators to "non-select" or "mis-select" elements across different versions of the same web page. In the former case, a locator is unable to retrieve the target element, whereas in the latter case a locator selects a different DOM element from the one that was used to target.

Concerning the *temporal* characterization of test breakages, researchers distinguish between direct, propagated, and silent breakages [24]. A breakage is called *direct* when the test stops at a statement st_i, and st_i has to be repaired in order to let the test pass or continue its execution. With *propagated* breakages, on the other hand, the test stops at a statement st_i, but another statement st_j, preceding st_i (i.e., $i > j$), must be repaired in order to make the test pass or continue its execution. Finally, *silent* breakages do not manifest explicitly because the test neither stops nor fails, but yet diverges from its original intent, and only by manually checking its execution (for example by looking at the actions performed on the GUI), the tester can detect the mis-behaviour.

Existing Web Test Repair Approaches. Test repair techniques have been proposed in recent years [11, 15–17, 20]. In the web domain, the state-of-the-art test repair algorithm is Water [11], a differential technique that compares the execution of a test over two different releases, one where the test runs correctly and another where it breaks. By gathering data about these executions, Water examines the DOM-level differences between the two versions and uses heuristics to find and suggest potential repairs. While the repair procedure of Water has a straightforward design and can manage a considerable number of cases related to locators or assertions, it has a number of limitations that derive from its DOM-related narrowness: First, considering only on the DOM as a source where to find possible repairs may be insufficient to find candidate fixes. Second, this can lead to a large number of false positives [11]. Third, the algorithm attempts repairs always at the point in which the test stops, which makes it impossible to handle propagated breakages.

3 A STUDY ON TEST BREAKAGES & REPAIRS

3.1 Breakages Root Cause Analysis

Given the predominance of locator-related test breakages [17, 24], we focus our analysis on locator problems. Existing taxonomies related to web and GUI breakages [24, 26, 28] lack a temporal quantification of breakages, even less they propose approaches to repair them. Indeed, knowing the exact position in which a test fails is arguably a prerequisite for developing effective automated test repair solutions. Although repairing locators seems mostly a mechanical and straightforward activity, to conduct an effective root cause analysis, testers must consider all aspects behind breakages (e.g., their causes and positions in the tests) and link them together to devise possible repairs that do not change the intended test scenario.

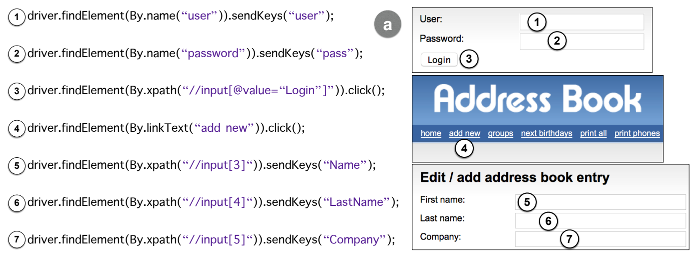
3.2 Study Design

To gain an understanding of the variety of web test repairs, we conducted a study to categorize the breakages from a repair-oriented perspective. The findings of our study highlight the complexity and the variety of breakage scenarios that can occur in web tests, which automated test repair techniques should aim to handle.

With the intention of studying realistic test regression scenarios, we selected open source web applications that were used in the context of previous research on web testing, for which multiple versions and Selenium test cases were available.

Table 1 (Web Applications) shows information about the selected applications, including their names, the numbers of releases considered, and the average number of lines of code, counted using `cloc` [13]. Table 1 (Test Suites) provides data on the test suites, including the total number of test cases counted across all releases along with the average number of test cases per release, and the total number of statements in all test cases counted across all releases along with the average number of statements per test case.

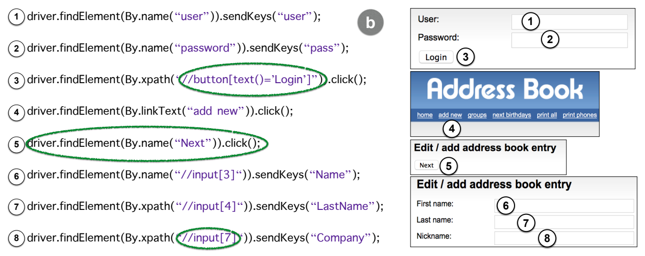
Procedure. To collect breakage information for each web application, we followed a systematic and iterative procedure, similar to



```
1 driver.findElement(By.name("user")).sendKeys("user");
2 driver.findElement(By.name("password")).sendKeys("pass");
3 driver.findElement(By.xpath("//input[@value="Login"]")).click();
4 driver.findElement(By.linkText("add new")).click();
5 driver.findElement(By.xpath("//input[3]")).sendKeys("Name");
6 driver.findElement(By.xpath("//input[4]")).sendKeys("LastName");
7 driver.findElement(By.xpath("//input[5]")).sendKeys("Company");
```

a
User:
Password:
Login
Address Book
home add new groups next birthdays print all print phones
Edit / add address book entry
First name:
Last name:
Company:

(a) *Version 6.2.12*



```
1 driver.findElement(By.name("user")).sendKeys("user");
2 driver.findElement(By.name("password")).sendKeys("pass");
3 driver.findElement(By.xpath("//button[text()='Login']")).click();
4 driver.findElement(By.linkText("add new")).click();
5 driver.findElement(By.name("Next")).click();
6 driver.findElement(By.name("//input[3]")).sendKeys("Name");
7 driver.findElement(By.xpath("//input[4]")).sendKeys("LastName");
8 driver.findElement(By.xpath("//input[7]")).sendKeys("Company");
```

b
User:
Password:
Login
Address Book
home add new groups next birthdays print all print phones
Edit / add address book entry
Next
Edit / add address book entry
First name:
Last name:
Nickname:

(b) *Version 7.0.0*

Figure 1: AddressBook web application, version 6.2.12 (1a) and version 7.0.0 (1b), along with Selenium WebDriver tests.

an analogous data collection study [24]. For each subject, and for each version V_n and its accompanying test suite T_n, we executed T_n on the subsequent version V_{n+1}. As locator breakages occurred, we noted information about the *type* of each breakage, the *position* of the associated repair, and assigned descriptive labels. Reviewing these descriptions allowed equivalence classes to be identified, to which descriptive labels were also assigned.

To continue the experimental work, however, we had to repair the breakages. To minimize bias and subjectivity, we utilized the same construct used by the locator (i.e., if an `id` attribute is changed, we used the new `id` attribute). In the cases where this was not possible, we favoured locators that we believe were the most robust and reliable [34]. We repeated this process until all the versions were taken into account.

Our benchmark comprises 733 individual test breakages, distributed as follows: 50 for AddressBook, 165 for Claroline, 218 for Collabtive, and 300 for PPMA.

Table 1: Subject systems and their test suites

	Web Applications		Test Suites	
	Releases (#)	LOCs (K)	Tot/Avg (#)	LOCs (K)
AddressBook	48	8,757	1,344/28	61,826/43
Claroline	12	338,129	492/41	20,287/43
Collabtive	14	150,696	560/40	22,710/38
PPMA	12	556,164	276/23	16,732/47
Total/Average	86	263,436	2,672/33	121,555/43

3.3 Test Breakages and How To Repair Them

We first describe the breakage scenarios our study revealed.

Basic Terminology. At a high level, each web test statement is a tuple `<locator, action, value>`. The locator component specifies the web element the test statement is interacting with. A locator l is a function on a DOM state $\mathcal{D}$. Notationally, $l : \mathcal{D} \to \{e\}$ where e is the web element returned by the locator l when applied to $\mathcal{D}$.

Non-Selection • Same Page. A non-selection occurs when a locator l applied to a DOM state $\mathcal{D}$ returns no elements—formally, $l : \mathcal{D} \to \emptyset$, but the target element e is still present on the page ($e \in \mathcal{D}$). Then, possible repairs require to find another locator $l' \mid l' : \mathcal{D} \to e$.

Let us consider the login page of AddressBook in Figure 1a, and the accompanying WebDriver test ⓐ. In the new subsequent version 7.0.0, as a result of the application evolution, the login button gets modified as follows: `<input value=‘‘Login’’></input>` becomes `<button>Login</button>`.

When executed on version 7.0.0, the test ⓑ will stop at Line 4 while attempting to locate the login button. At a visual inspection of the two GUIs, a tester would expect the test to work, because her perception is immaterial as far as changes at DOM-level are concerned. It is indeed evident that the target element is *visually* still present on the page, and its position *on the GUI* has not changed.

Non-Selection • Neighbouring Page. Notationally this can be expressed as $l : \mathcal{D} \to \emptyset \wedge e \notin \mathcal{D} \wedge \exists\, \mathcal{D}' \in neigh(\mathcal{D}) \mid l : \mathcal{D}' \to \{e\}$.

As a concrete example consider Figure 1a, specifically the pages in which the user can insert a new entry. The test ⓐ clicks on the “add new” link on the home page (Line 4), and fills in the “First name”, “Last name” and “Company” text fields (Lines 5–7). When executed on the successive version 7.0.0 ⓑ, the test will raise an exception of kind `NoSuchElementException` at Line 5, when attempting to locate the “First name” text field. Indeed, a new intermediate confirmation page has been added, and the navigational workflow of the test must be corrected to reflect that of the new web application.

From a testing perspective, the “First name” text field can no longer be found on the web page (test state) following the execution of the statement at Line 5. However, conceptually, the repair action that needs to be triggered in order to correct the test has nothing to do with the locator at Line 6. In fact, by only looking at the exception, it is arduous for the tester to understand what the actual problem is, unless the *visual execution* of the test is taken into consideration. A possible solution would require to (1) detect that the web element e no longer exists as part of the test state st_i in version V, (2) try to match the e in the neighbouring states of st_i in the new version V', which in turn requires to (3) find a web element $e' \in st_i$ such that $(e', st_i) \to st_j$ (the “Next” button in Figure 1b).

Non-Selection • Removed. The third and last non-selection scenario concerns a web element being removed from a web page. Formally, this can be expressed as $l : \mathcal{D} \to \emptyset \wedge \nexists\, \mathcal{D}' \in neigh(\mathcal{D}) \mid l : \mathcal{D}' \to \{e\}$. Let us consider the application being evolved in the reverse order as depicted in Figure 1 (i.e., from version 7.0.0 to version 6.2.12). The test ⓑ would stop at Line 6, when trying to select the “Next” button, which was on a page that is no longer present. In this case, the only possible fix is to *delete* the statement at Line 6.

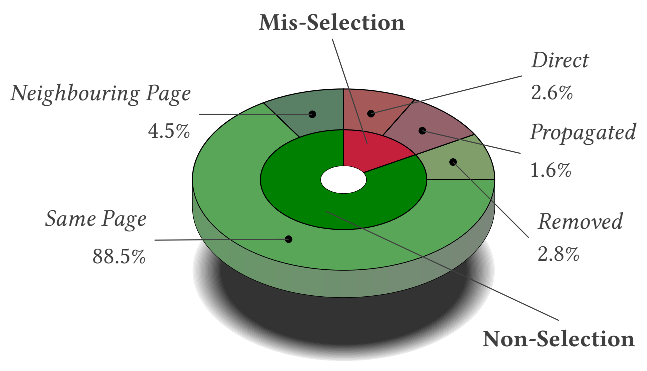


Figure 2: Causal (inner pie) and Temporal (outer pie) Repair-Oriented Characterization of Locator Breakages.

Mis-Selection • Direct and Propagated. The addition or repositioning of elements within the DOM can cause locators to "mis-select" elements across different versions of the same web page. Specifically, a mis-selection occurs when a locator selects a different DOM element from the one that was used to target. Notationally, $l : \mathcal{D} \longrightarrow \{e\}$ in V, and $l : \mathcal{D}' \longrightarrow \{e'\}$ in V' where $e \neq e'$.

Consider Figure 1 again. Suppose that the test ⓐ is repaired so as to reach the edit page on version 7.0.0 (for instance, as in ⓑ). On the new version 7.0.0, the statements at Lines 6–7 will execute correctly, whereas at Line 8 (in the new version) the test will fill in the field "Nickname", instead of the field "Company".

The mis-selection problem can lead to unpredictable test executions, that diverge from the test's intended behaviour. The test may continue its execution until it reaches a point in which an action cannot be performed or an element cannot be found, but the actual repair has to be triggered *in a previous test statement*.

In fact, the point in which the repair must be applied varies depending on where the mis-selection first originated. Repairing mis-selections requires finding another locator $l' \mid l' : \mathcal{D}' \longrightarrow \{e\}$.

3.4 Findings

Figure 2 shows the distribution of the different classes of locator breakages and relative repairs. Our study revealed two major categories, each of which has specific subtypes.

The most prevalent category concerns *Non-Selection* of web elements (695/733), all raising direct breakages in the tests. Concerning the associated repairs, the vast majority were found on the *Same Page*, whereas other scenarios refer to web elements that were moved to a *Neighbouring Page*, or being *Removed* from any web page. The second main category consists of *Mis-Selection* of web elements (38/733), whose repairs were always found within the same page, and of which 2/3 led to *Direct* breakages, and 1/3 to *Propagated* breakages. We have not observed silent breakages in our benchmark. Additionally, we collected 94 test *failures*—meaning the tests exposing actual bugs—and 22 failures due to obsolete assertion values. In this work, we only focus on repairing locator breakages.

Overall, 329/2,672 tests (12%) exhibited at least one breakage, and ≈80% of those tests suffered from *multiple* breakages. Figure 3 shows box-plots of the distribution of such breakages per test in each subject system. We observe that, on average, between 1–4 breakages are present per test.

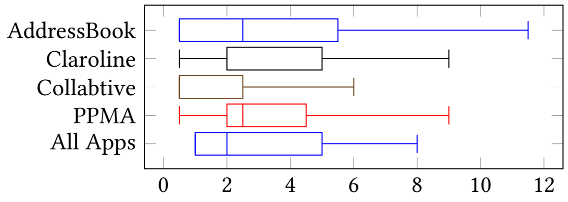


Figure 3: Distribution of test breakages per broken test case.

To summarize, (1) test suites break frequently as the web application evolves, (2) breakages may occur multiple times within the same test, (3) breakages fall into multiple recurrent patterns which we call breakage scenarios in this paper.

4 APPROACH

The goal of our approach is to detect the occurrence of locator breakages as tests execute, and find potential fixes to report to the user. Our focus is on the classes of breakages described in Section 3.3. In a nutshell, our approach captures the visual execution trace of each test statement for a version V_n and uses it to validate their correctness (or to find potential repairs) when they are executed on a subsequent version V_{n+k} $(k \geq 1)$.

Figure 4 illustrates the usage scenario of our approach, which requires a correct version of the web application V along with its working test suite TS (i.e., in which all tests pass) ❶. A tester would run TS by means of the first module of the presented approach, the Visual Execution Tracer ❷. Such a module collects, for each test, a variety of visual information (e.g., screenshots) ❸. Then, as the application V evolves into a new version V', a tester may wish to use TS to check if regressions have occurred ❹. To this aim, the tester would use the second module of our approach, the Visual Test Repair ❺, which runs each test case of TS against the new version V', and makes use of the information about the previous execution traces to *validate* each test statement. On the detection of breakages, our approach attempts to find candidate repairs that can fix them. At the end of the process, the approach outputs the validated (and eventually repaired) test suite TS', together with report information ❻. The tester can then manually analyze the report along with the repaired test cases. TS' now represents a working test suite for the version V' which can be used as a baseline for our approach when V' will evolve further. The manual effort required is potentially significantly reduced in comparison to a user carefully verifying each executing test and manually searching for fixes as breakages occur. We now detail each step of our approach.

4.1 Visual Execution Trace Collection

In the first part, the approach captures information about the *dynamic visual execution trace* of the test cases, associating each test statement with its visual information.

More in detail, for each test statement, our approach captures the screenshot of the web page and the coordinates and size of the web element's bounding box in order to get a visual locator [35].

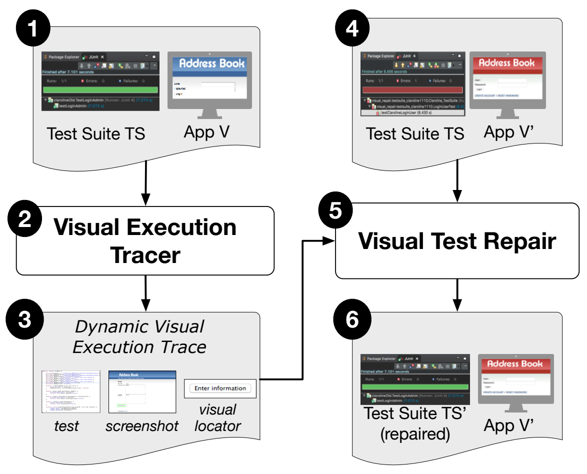


Figure 4: Overview of our approach

Algorithm 1: Visual Test Repair

```
    Input: $T$: A test case developed for web application $V$, $EX$: execution trace of $T$ when
           executed on $V$, $U$: the URL of subsequent version $V'$
    Output: $T'$: A verified/repaired $T$ working on $V'$, repairMap: repairs for each test step
1   trace ⟵ loadExecutionTrace($EX$)
2   driver ⟵ loadApp($U$)
3   statements ⟵ getStatements(T), repairMap ⟵ Map⟨statement, <repair, locator> ⟩
4   foreach test statement $st_i$ ∈ statements do
5       locator ⟵ getLocator($st_i$)
6       visLocator ⟵ getVisualLocator(trace, $st_i$)
7       webElement ⟵ driver.findElement(locator)
8       /* Manage non-selection of web elements. */
9       if webElement == null then
10          webElemVisual ⟵ VISUALSEARCH(driver, $st_i$, visLocator)
11          if webElemVisual == null then
12              webElemVisual ⟵ LOCALCRAWLING(driver, $st_i$, visLocator)
13              if webElemVisual == null then
14                  repairMap.add($st_i$, <remove, null>)
15              else
16                  repairMap.add($st_i$, <addBefore, webElemVisual>)
17              end
18          else
19              repairMap.add($st_i$, <update, webElemVisual>)
20          end
21          /* Manage mis-selection of web elements. */
22      else if webElement ≠ null then
23          webElemVisual ⟵ VISUALSEARCH(driver, $st_i$, visLocator)
24          if EQUIVALENT(webElement, webElemVisual) == false then
25              repairMap.add($st_i$, <update, webElemVisual>)
26          end
27      end
28      CONFIRMENTERREPAIR(repairMap, $st_i$)
29      driver ⟵ EXECUTESTATEMENT($st_i$, $V'$)
30  end
31  $T'$ ⟵ SYNTHESIZETEST(repairMap)
32  return $T'$, repairMap
```

4.2 Visual Test Repair

Algorithm 1 illustrates the main procedure of our algorithm for the visual-augmented execution of test cases. The procedure takes as input a test case T, the dynamic visual execution trace EX of T on a version V of the web application, and the URL U of the subsequent version V' of the web application. The outputs are a test T', validated or corrected to work on V', and the map of repairs.

Initialization. The initial part involves loading the dynamic visual execution trace EX, and opening a WebDriver session with the new version V' (Lines 1–2).

Visual-Augmented Execution. The information contained in EX is used to *visually* validate each statement st_i of T, when executed on V' (main loop Lines 4–30). The validation proceeds as follows. First, the DOM-based locator utilized by st_i is extracted, along with its visual locator (i.e., an image) in V (Lines 5–6). Then, the `driver` instance is used to query the DOM of V' to observe if the original DOM locator returns a web element (Line 7).

Detecting and Repairing Non-Selection breakages. If no element is returned, we treat it as a non-selection happening on the *Same Page*. Given that st_i, if executed, will break in V', we attempt to find a candidate repair through a series of countermeasures.

The first heuristic tries to search for the web element *visually* on the same page. The `visualSearch` function (Line 10) uses advanced computer vision algorithms to retrieve the target web element by matching the visual locator captured in V on the current GUI of V' (see details in Section 4.3). If a result is found, the corresponding DOM element's XPath is retrieved, and saved in the map of candidate repairs (Line 19). Before proceeding to the next statement, the approach outputs the outcome of the verification phase to the user, who can inspect, confirm, or enter a manual fix before executing st_i (for brevity, such details are encapsulated within the `confirmEnterRepair` and `executeStatement` functions at Lines 28 and 29).

If the visual search on the same page returned no elements, then our approach assumes that such an element no longer exists on the current page and considers it as a broken workflow. A local exploration of the application state space is hence triggered (procedure `localCrawling` of Line 12) in order to find the element in a *Neighbouring Page* (see details in Section 4.3). In each new state discovered by the exploration, the crawler executes the `visualSearch` procedure to locate the target element. If a match is found in at least one of those states, the XPath of the element to reach that page is saved in the map of repairs (Line 16), and marked as a test statement that needs to be added *before* st_i in the repaired test case (thus creating the missing transition).

On the contrary, if a match is not found, i.e., no elements were found through local crawl, our approach considers the web element as *Removed* from the application, thus it suggests the deletion of st_i (Line 14).

Detecting and Repairing Mis-Selection breakages. If a web element was returned by the original DOM locator (Line 7), our approach attempts to validate the correctness of the selection by using the previously collected visual information (Lines 22–27). The `equivalent` function checks the equivalence of the web elements found by the DOM locator and the visual locator. If they do differ, our approach considers it a possible case of *Mis-Selection* that could lead to a direct or propagated breakage. The procedure stores the alternative web element's XPath (Line 25), it reports the mismatch to the user and asks for her input.

Outputs. At last, a repaired test T' is automatically created upon the given suggestions/manual repairs (Line 31), and the algorithm terminates. In the following section, we describe the `visualSearch` and `localCrawling` procedures that underlie at the functioning of our approach.

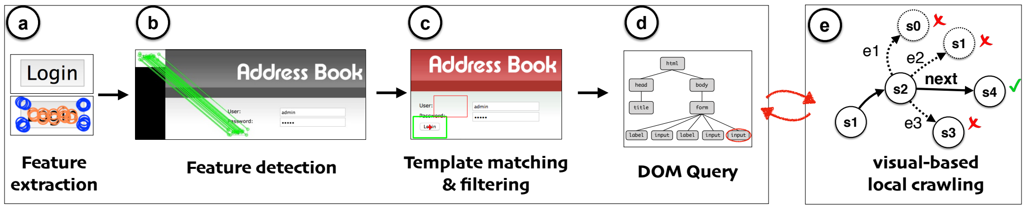


Figure 5: Computer vision pipeline for robust web element detection (left) and visual-based local crawling for workflow repair.

4.3 Visual Search of Web Elements

Computer vision (CV) provides techniques for analyzing web pages the way end users perceive them. Thereby, we provide a short background of the CV solutions that are relevant in this work.

Challenges of Template Matching. Identifying web elements *visually* across different versions (i.e., pages) of a web application can be tackled with an image analysis technique known as *template matching*. Template matching aims to detect the occurrence of a specific object image (template) in another reference image [8]. The template is slid over the image, and a similarity measure is computed at each pixel location. The top-left corner point of the area where the maximum similarity has been found is then returned.

In our context, the matching technique must handle *translation* (i.e., the captured template can be shifted with respect to the reference image) and *scaling* (i.e., the aspect ratio of the template is not preserved in the reference image, or a different font is used) problems. Indeed, web applications are often modified to accommodate cosmetic or stylistic changes to align the GUI with the latest trends. These changes may render our technique fragile, because the visual locators captured on the *old* GUI might be ineffective on the *new* GUI of the application. Additionally, standard template matching algorithms are not effective in detecting the presence/absence of a template, which is instead of paramount importance for the accuracy of our algorithm. Even though stricter similarity threshold values might be used, in our exploratory experiments this strategy failed to provide robust or generalizable results. Thus, we explored a more advanced CV solution, namely feature detection.

Feature Detection. The philosophy of this method is to find certain "important" points (*key-points*) in the template image and store information about the neighbourhood of those key-points (*key-point descriptors*) as a description of the template. Then, after finding the descriptors of key-points in the reference image, the algorithm tries to match the two descriptor sets (one from the template and one from the reference image) using some notion of similarity, and see how many descriptors match.

Thus, the `visualSearch` procedure adopts a pipeline of CV algorithms, which is graphically illustrated in Figure 5.

Feature Detection for Template Absence & Presence. We implemented a detector based upon two extensively-used and accurate feature detection algorithms from the CV literature, SIFT [36, 37] and FAST [47, 48]. The detector uses these algorithms to detect the key-points from the template image using SIFT descriptors ⓐ, and then adopts a Flann-based descriptor matcher with a distance threshold ratio of $\gamma = 0.8$, as per Lowe's paper [37]. The detector returns a positive result on the presence of the template if at least 70% of the key-points are matched. We used a combination of two feature detection algorithms because we found, through experimentation, that SIFT and FAST complement each other. As such, SIFT performs well mostly for textual-based templates whereas FAST can handle the cases where the template represents an "empty" text field, as it is specifically designed for corner detection. In our pictorial example ⓐ, most of the key-points detected by SIFT are in fact nearby the "Login" label (orange circles), whereas FAST detected key-points also in proximity of the corners (blue circles). In ⓑ we can see how the algorithms matched the key-points onto the new GUI of the application. For further details, the interested reader is referred to the relevant literature [36, 37, 47, 48].

Template Matching with Visual Filtering and NMS. In the next step, if the feature detection returned a positive result about the presence of the template image, a template matching technique is executed ⓒ. We experimented with different algorithms available in the open-source computer vision library OpenCV [44]. We found the Fast Normalized Cross Correlation algorithm [7] with a similarity threshold $\delta = 0.99$ to be optimal in our setting.

Yet, any template matching technique might return (1) false visual matches, as well as (2) multiple overlapping bounding boxes around the area of interest. Since our procedure ought to return exactly one result, a post-processing algorithm is responsible for discarding the false matches and merging all redundant detections. In brief, (1) the matches that do not fall in the region where the key-points have been found are discarded, and (2) a non-maxima suppression operation (NMS) is also applied [39] (basically NMS assumes highly overlapping detections as belonging to the same object [9]). Thus, only the *closest* match is returned (see the green thick rectangle over the "Login" button).

To summarize, the three CV algorithms (SIFT, FAST, and Fast Normalized Cross Correlation) operate synergistically to find a consensus on the location of the best match.

From GUI to DOM. At last, in order to retrieve the DOM element corresponding to a specific point of coordinates (x, y), the `visualSearch` function queries the browser through JavaScript to retrieve the DOM element whose bounding box *centre* contains x and y ⓓ. (otherwise, a DOM ancestor of the searched web element—as a `form` or `div` container—will be erroneously returned).

Local Crawling for Workflow Repair. Manually repairing every broken workflow is tedious and frustrating, since even a medium-size web application may contain tens of GUI screens and hundreds of GUI actions. It is hence likely infeasible for a tester to quickly explore this state space to choose replacement actions from. Fortunately, a web crawler can do this automatically. To this aim, the `localCrawling` function explores the state space of V' looking for a visual match in the immediate neighbouring pages of the current web page. If a match is found, the workflow is repaired by adding a transition to that page through the matched element (Figure 5 ⓓ shows how the local crawling approach works for the example of Figure 1b ⓑ). Otherwise, our approach considers the element as removed. Thus, to repair the workflow, it suggests the deletion of the corresponding test step.

4.4 Implementation

We implemented our approach in a tool called Vista (**Vis**ual **T**est Repair), which is publicly available [58]. The tool is written in Java, and supports Selenium test suites written in Java. However, our overall approach is more general and applicable to test suites developed using other programming languages or testing frameworks. Vista gets as input the path to the test suites, collects the visual execution traces by means of PESTO [35]. Our visual detection pipeline adopts algorithms available in the open-source computer vision library OpenCV [44]. Vista makes use of the traces to retrieve potential repairs and generates a list of repaired test cases. For the local crawling exploration, Vista features a Crawljax [41, 42] plugin that incorporates the `visualSearch` function. In our evaluation, we limited the crawler's exploration depth to one (1), its running time to 2 minutes, and configured it to return the first match found. This was a conservative choice, since the number of DOM states and events can be numerous. While this limits the search capability, this design choice kept the running time acceptable.

For the interested reader, more technical details can be found in our accompanying demo paper [56].

5 EMPIRICAL EVALUATION

We consider the following research questions:

RQ$_1$ (effectiveness): How effective is the proposed visual approach at repairing test breakages compared to a state-of-the-art DOM-based approach?

RQ$_2$ (performance): What is the overhead and runtime of executing the visual approach compared to the DOM-based one?

RQ$_1$ aims to evaluate the effectiveness of Vista at detecting and repairing test breakages, and how this varies across the breakage classes. RQ$_2$ aims to evaluate the overhead and running time of the two main modules of Vista: the Visual Execution Tracer and the Visual Test Repair.

Object of Analysis. We used the same subjects and test suites discussed in Section 3.2, for which 733 breakages were identified.

Independent Variables. In our study, the visual-aided approach is represented by Vista. As a DOM-based approach we chose Water. However, we have not adopted the implementation described in the original paper [11]. In fact, there are fundamental design differences between the algorithms: Water is an offline technique that runs a test, collects information about the breakages, and runs the repair as a *post-processing* procedure. Our algorithm, instead, is designed to analyze the test suite at runtime, and to attempt repairs in an *online* mode. Given that the scope of this paper is to compare the effectiveness at repairing breakages, we implemented a DOM-based version of our approach by (i) customizing the Visual Execution Tracer to collect DOM information and (ii) invoking Water's repair heuristics (referred to as `RepairLocators` in [11]) within our main Algorithm 1. For simplicity, however, in the evaluation section we refer to the two competing tools as Vista and Water.

Dependent Variables. To measure effectiveness (RQ$_1$), we counted the number of *correct repairs* as well as the amount of *manual repairs* triggered by the two techniques on our benchmark. Regarding efficiency (RQ$_2$), we utilize two metrics. First, we measured the overhead imposed by the Visual Execution Tracer on the test suite to create the visual traces. As a second metric, we counted the time spent by each competing technique at repairing breakages.

Procedure. For each subject application, we applied Water and Vista to each test T_n in which a breakage was observed on the successive release V_{n+1}. For each tool, and for each breakage, we manually examined the first proposed repair *rep* to determined its correctness. If *rep* was found correct upon manual inspection, we incremented the number of correct repairs. When both tools were unable to repair, we incremented the number of *manual* repairs.

5.1 Results

Effectiveness (RQ$_1$). Table 2 presents the effectiveness results. For each subject, the table reports the number of breakages, and the amount of correct repairs triggered by Water and Vista both numerically and percentage-wise. The results are further divided into the various breakage classes. Totals across all applications are also provided.

Overall, we can notice that Water was able to repair 420/733 breakages (57%), whereas Vista found correct repairs for 592/733 breakages (81%). Vista was hence able to correct 172 breakages more than Water, a 41% increment.

Looking at the specific breakage classes, each tool performed well with respect to *Same Page*: Water repaired correctly 63% of the times, whereas Vista 84%. Concerning *Neighbouring Page*, no correct repairs were found by Water, while Vista was 33% successful. About elements being *Removed*, both tools performed equally, detecting 14/21 cases (67%).

The main difference between the two approaches emerges when considering *Mis-Selection* cases: Water was never able to detect any mis-selection of elements, whereas Vista detected and repaired correctly on average 80% of them (avoiding 94% of *Direct* breakages and preventing 67% of *Propagated* ones).

Overall, the visual approach implemented by Vista was constantly superior to Water, over all the considered applications. Individual improvements range from +28 (45-17) for AddressBook to +69 (119-50) for Claroline.

Table 3 compares the two approaches further. The first row shows the number of cases in which both tools were able to correct the breakages. The second row, instead, illustrates the cases in which Vista prevailed, whereas the third row reports the cases that Water repaired correctly whereas Vista did not. Lastly, the last fourth row indicates the number of breakages that neither tool was able to correct, and that were hence fixed manually.

In more than half of the cases (56%), both tools found the correct repair. However, in 175 cases Vista repaired a breakage that Water could not fix. Conversely, only in three cases Water prevailed over the visual approach. Finally, 19% of breakages were repaired *manually* because neither technique was able to fix them. We will discuss those cases in Section 6.

Table 2: Repair results across all applications and across all breakage classes

	AddressBook					Claroline					Collabtive					PPMA					All Apps				
	Breakages (#)	Rep. WATER (#)	%	Rep. VISTA (#)	%	Breakages (#)	Rep. WATER (#)	%	Rep. VISTA (#)	%	Breakages (#)	Rep. WATER (#)	%	Rep. VISTA (#)	%	Breakages (#)	Rep. WATER (#)	%	Rep. VISTA (#)	%	Breakages (#)	Rep. WATER (#)	%	Rep. VISTA (#)	%
Non-Selection																									
Same Page	23	3	13	22	96	162	50	31	117	72	189	166	88	184	97	275	187	68	219	80	649	406	63	542	84
Neighbouring Page	1	0	0	1	100	0	0	0	0	0	11	0	0	10	91	21	0	0	0	0	33	0	0	11	33
Removed	14	14	100	14	100	0	0	0	0	0	3	0	0	0	0	4	0	0	0	0	21	14	67	14	67
Mis-Selection																									
Direct	8	0	0	8	100	3	0	0	2	67	7	0	0	7	100	0	0	0	0	0	18	0	0	17	94
Propagated	4	0	0	0	0	0	0	0	0	0	8	0	0	8	100	0	0	0	0	0	12	0	0	8	67
Total	50	17	**34**	45	**90**	165	50	**30**	119	**72**	218	166	**76**	209	**96**	300	187	**62**	219	**73**	733	420	**57**	592	**81**

Table 3: Comparison between DOM and Visual Repair Strategies for all breakages, and amount of Manual Repairs.

		AddressBook	Claroline	Collabtive	PPMA	All Apps
VISTA ✓	WATER ✓	17	47	166	187	417
VISTA ✓	WATER ✗	28	72	43	32	175
VISTA ✗	WATER ✓	0	3	0	0	3
VISTA ✗	WATER ✗	5	43	9	81	138

Performance (RQ$_2$). To assess the performance of running our approach, we measured the execution time on a macOS machine, equipped with a 2.3GHz Intel Core i7 and 16 GB of memory.

Table 4 (Trace Generation) shows the average time required to run the test suites *without* (Column 2) and *with* (Column 3) the Visual Execution Tracer (VET) module, and Column 4 reports the relative slowdown. Overall, our technique imposed a 3.6× slowdown compared to normal test suites execution (+38 minutes). The biggest overhead occured for Collabtive (+19 minutes), whereas the lowest occured for AddressBook (+3 minutes).

Concerning the execution time of our repair technique, Table 4 (Repair Time) shows the total running time, in seconds, for each tool. To allow a fair comparison, we split the results between the cases in which the tools were successful (Columns 5 and 6) and unsuccessful (Columns 7 and 8).

In the former case, WATER was overall 7% faster than VISTA, which employed nearly 4 minutes more. In the latter case, WATER was 600% slower as compared to VISTA. Application-wise, WATER was faster on AddressBook and Claroline (about -4 and -3 minutes, respectively), but slower on Collabtive and PPMA (+17 s and +2.6 minutes, respectively). We will comment those results in Section 6.

6 DISCUSSION

In this section, we discuss some of our findings, tool design decisions and limitations, as well as threats to validity of our study.

Our empirical study confirmed the fragility of E2E web tests when used for regression testing scenarios (Section 3.2). Although such tests did fail 94 times due to actual bugs, breakages (733) are still a largely predominant issue affecting test suite maintenance.

Table 4: Performance results

	Trace Generation			Repair Time			
	seconds		**%**	**correct (s)**		**incorrect (s)**	
	Original	With VET	Slowdown	WATER	VISTA	WATER	VISTA
AddressBook	50	207	4.1×	127	388	594	59
Claroline	155	861	5.6×	600	767	5,571	592
Collabtive	622	1,780	2.9×	1,930	1,913	621	82
PPMA	228	458	2.0×	802	646	2,972	659
Total	1,055	3,306	3.6×	3,459	3,714	9,758	1,392

This is why repairing web tests is often considered a tedious activity that leads the test suites to be abandoned [12]. This is certainly due to the high frequency at which those tests break, but also due to the dearth of solid tooling solutions in this area.

6.1 Explaining Results across Approaches

Effectiveness. Our evaluation revealed that our visual-based approach can successfully repair a large number of breakages correctly, outperforming an existing DOM-based solution.

We investigated why *all* breakages were not repaired. We enumerate some of the main reasons next, which pertain both to the inherent nature of our subjects and tests and some of the design choices and limitations of the two competing approaches.

WATER could not detect any *Mis-Selection* case because the misselected element is among those retrieved by its heuristics. VISTA, on the contrary, does not trust *a priori* DOM-based locators and validates them by means of their visual appearance.

For analogous reasons, the DOM-based repair strategy was ineffective when searching elements with local crawling (*Neighbouring Page* cases), whereas VISTA could detect nearly one third of them. The failing cases were due to the retrieving of no elements (or wrong ones). In PPMA, a challenging case occurred: the desired element was hidden in a hoverable drop-down menu, whereas the crawler does not detect such interaction patterns as it was instructed to search in the neighbouring pages.

Concerning the *Removed* cases, both techniques failed seven times because the crawler retrieved a wrong element. For the *Same Page* category, both techniques performed quite well, however, Vista achieved a 33% improvement in the correct repair rate.

For the Claroline application, Water outperformed Vista in three cases. These breakages concerned buttons whose DOM attributes were stable across the two considered releases whereas their visual appearance did change substantially making our visual technique ineffective.

Looking at the results on a per-application basis, AddressBook and Collabtive tests experienced breakages pertaining to all classes. On such applications, Vista performed better than Water, with 165% and 11% improvements on the number of correct repairs, respectively. For AddressBook, the DOM evolved quite frequently across the releases. Thus, for a tester it would be challenging to find reliable attributes upon which to create stable web element locators (and hence robust test suites). Also the GUI of the web application evolved, e.g., with changes in background colour and font size to make it more appealing and user-friendly. However, our approach demonstrated to be robust to both shifting and scaling (invariant to translation and scale transformation).

Five mis-selection breakages could not be corrected by either of the two approaches. Those cases refer again to buttons whose tag changed (from `<img>` to `<i>`) as well as their visual appearance (from buttons to icons).

In Collabtive, the high number of breakages is explained by the numerous XPath locators used in the test suite, which are typically impacted by minor DOM changes. However, such DOM-level modifications did not jeopardize the effectiveness of Water. Also Vista had its best best repair results because the GUI remained quite stable across the analyzed releases.

For Claroline and PPMA, almost all repairs were found on the *Same Page*, with Vista being 134% and 17% more effective than Water, respectively. Claroline and PPMA were also the applications in which more manual repairs were needed. Claroline experienced significant DOM and GUI evolutions, whereas for PPMA our technique failed due to timing issues (e.g., delays or page reloads that were needed to display the web elements on the GUI).

Performance and Overhead. When the tools are successful, their running time is comparable, with Water being on average slightly faster than Vista, across all breakages. However, looking at the results in conjunction with the repair rate, suggests a different picture. Water repaired 420 breakages correctly in 58 minutes (8.2 s/repair), whereas Vista repaired 592 breakages correctly in 62 minutes (6.3 s/repair). On the contrary, when the tools were unable to repair, Water employed 139 minutes more than Vista (31.2 *vs* 9.9 s/repair).

These low results are explained by two reasons: (1) if the DOM-based heuristic fails to find the target element on the same state, the local crawling is unnecessarily executed, and (2) one of Water's heuristics searches for web elements having *similar* XPaths to the target element's. Being real world applications, our subjects have web pages with significant DOM sizes. Thus, Water was often forced to compute the similarity score on dozens of web elements, whereas Vista required only two feature detection and one template matching operations on images of reasonable sizes. This gives us confidence in the applicability of our technique in common web testing scenarios.

6.2 Estimated Manual Effort Saving

Our study revealed that 329/2,672 tests of our benchmark (12%) experienced at least one breakage for which the test engineer must find an appropriate repair for. Manually inspecting the application to find a correct fix and creating a locator for *each* broken statement can be however a time-consuming task.

In a recent test repair work [23], authors report an average *manual* repair time of 15 minutes/breakage. On our dataset, hypothetically speaking, the estimated manual effort reduction thanks to our technique would be 148 hours ($\frac{592 \cdot 15}{60}$).

Our own experience in repairing tests, gained during the empirical study, corroborates the costliness of the task. For example, in AddressBook, one test for the search functionality broke 11 times when applied from version 4.02 to version 4.1.1, due to three non-selections (*Removed*) and seven mis-selections (*Direct*). Vista created the dynamic visual execution trace of this test in 22 s, and then found correct repairs for all 11 breakages in 57 s. Thus, in this specific case, our technique is advantageous only if the manual detection and repair performed by a human tester takes more than 80 s (7 s/breakage), which seems likely infeasible in practice.

However, comparative experiments with human testers are necessary to measure the costs associated with repairs. Before undertaking such a study, we evaluated whether our technique has prospect for success against a state-of-the-art repair algorithm.

6.3 Applications

Vista can be used by test engineers to validate and repair their E2E web test cases. Each statement of the tests is associated with its visual trace. Hence, this information can also aid testers to monitor the behaviour of the test suite across time and perform more effective *root cause analysis*.

Our technique can also play a role in *automating software oracles*. A similar approach is implemented in tools such as Applitools [2], where testers can manually inject visual checks at specific places of the test's execution. This has two drawbacks: (1) the insertion of the check-points must be performed manually, (2) this extra-code clutters the initial test code, with statements that do not pertain to the test scenario itself. On the contrary, our technique can be introduced smoothly in existing testing infrastructures, as it neither impacts nor modifies the source code of the test cases.

Vista can be utilized as a *runtime monitoring* technique for the detection of tests misbehaviours. Our study shows that Vista goes beyond simple detection and can efficiently suggest a high number of correct repairs at runtime, while requiring a low overhead. Hence, our approach can be utilized as an automated *self-repair* testing technique, and can also be integrated with robust locator generation algorithms [31, 33, 34].

Finally, our taxonomy can drive the design of automated test repair techniques such as the one presented in this work.

6.4 Limitations

Vista depends on the uniqueness of the templates used to identify web elements. When feature detection fails, template matching is not executed, thus undermining the applicability of our approach. Concerning the local exploration, since crawling depth is limited to one, we manage workflow repairs that pertain to the addition/removal of one statement only. However, if two subsequent statements needs to be removed, our technique does so by running the crawler twice. Moreover, our implementation does not currently support the creation of general purpose statements, such as the ones that need input data, or the repair of assertion values.

6.5 Threats to Validity

We limited the bias of having produced test suites ourselves by choosing existing test suites. This also ensures, to some extent, that the chosen objects of analysis are non-trivial, therefore representative of test suites that a real web tester would implement. The manual evolution task poses a threat to validity that we tried to mitigate by following a systematic approach in repairing the breakages.

Concerning the generalizability of our results, we ran our approach with a limited number of subjects and test suites. However, we believe the approach to be applicable in a general web testing scenario (unless strict timing constraints apply), even though the magnitude of the results might change. To limit biases in the manual selection of the versions, we considered *all* the available releases after those for which the test suites were developed for.

Lastly, we constructed our taxonomy based on the analysis of several hundreds of tests on real-world applications. However, we do not claim that our taxonomy represents all possible breakage scenarios. However, taxonomies typically evolve as additional observations are made.

7 RELATED WORK

Web and GUI Test Repair. We already discussed and evaluated Water [11]. A recent work [23] adopted Water to repair the breakages happening to the intermediate minor versions between two major releases of a web application.

Grechanik et al. [22] analyze an initial and modified GUI for differences and generate a report for engineers documenting ways in which test scripts may have broken. Zhang et al. [60] describe FlowFixer, a tool that repairs broken GUI workflows in desktop applications. Memon [40] presents an event-based model of a GUI which is used to determine the modifications during software evolutions. In similar fashion, Gao et al. [20] present SITAR, a model-based repair algorithm of unusable GUI test scripts.

ReAssert is a tool to repair mainly assertions in unit tests for Java applications [14–17]. Huang et al. [25] describe a genetic algorithm to automatically repair GUI test suites by evolving new test cases that increase the test suite's coverage while avoiding infeasible sequences.

Differently from the aforementioned works, our test repair technique uses a computer vision-based approach to fix classes of breakages specific to the web testing domain that have not been reported in general GUI desktop applications.

Breakage Prevention. Recent papers have considered increasing the robustness and maintainability of web test suites. In order to make test scripts robust, several tools producing smart web element locators have been proposed [3, 31, 33, 34, 59]. A study by Leotta et al. [29] discusses the development and maintenance cost associated with both DOM and visual test suites for web applications. Additionally, Stocco et al. [52–55] investigate the automated generation of page objects that confine causes of test breakages to a single class, a form of breakage prevention.

Computer Vision to assist SE tasks. Recently, software engineering community has witnessed an increasing adoption of CV techniques to assist or solve common software engineering tasks.

One of the foundational works on computer vision applied to testing is by Chang and colleagues. Their tool Sikuli [10] allows testers to write a visual test script that uses images to specify which GUI components to interact with and what visual feedback to be observed. Their work shows how this approach can facilitate a number of testing activites such as unit testing, regression testing, and test-driven development. On the same line, JAutomate [1] is a visual record-and-replay tool for the testing of GUI-based applications.

CV techniques have been employed to detect cross-browser incompatibilities (XBIs). WebSee [38] compares whole images with a perceptual difference technique, whereas WebDiff [49] and X-PERT [50] utilize an image similarity technique based on image colour histogram matching.

The tool PESTO [32, 35, 51] migrates DOM-based web tests to visual tests. It does so by using a standard template matching algorithm for the automatic construction of visual locators, which we used for the development of the Visual Execution Tracer.

Feng et al. [18] use a combination of visual analysis and NLP techniques to assist the inspection of crowdsourced test reports. Ramler and Ziebermayr [46] use visual test automation joint with computer vision techniques to test physical properties of a mechatronic system, whereas Kıraç and colleagues [27] used an image processing pipeline for test oracle automation of visual output systems. Bajammal et al. [4] use visual methods generate reusable web components from a mockup in order to facilitate UI web development. In another work, Bajammal et al. [5] use visual analysis to infer a DOM-like state of HTML canvas elements, thus making them testable by commonly used testing approaches.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed Vista, a novel web test repair technique based on a fast image-processing pipeline. First, we used 2,672 tests spanning 86 releases of four subject systems to collect a dataset of 733 individual test breakages, for which a repair-oriented taxonomy has been presented. Then, we empirically evaluated our technique at repairing these breakages. Overall, Vista was able to repair correctly on average 81% of such breakages, outperforming the state-of-the-art solution, while incurring an acceptable overhead.

For future work, we plan to investigate the potential for hybridization, i.e., joining DOM- and visual- heuristics in a single solution. We also intend to run a controlled experiment with human subjects to measure the accuracy of the suggested repairs.

Lastly, we plan to experiment our technique with more subjects, and study its capabilities to repair mobile test suites.

REFERENCES

[1] E. Alégroth, M. Nass, and H. H. Olsson. 2013. JAutomate: A Tool for System- and Acceptance-test Automation. In *Proceedings of IEEE 6th International Conference on Software Testing, Verification and Validation (ICST '13)*. 439–446.

[2] Applitools 2018. Applitools. Visual app testing and monitoring. https://applitools.com/. (2018). Accessed: 2017-08-01.

[3] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2015. Synthesizing Web Element Locators. In *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Computer Society, 331–341.

[4] Mohammad Bajammal, Davood Mazinanian, and Ali Mesbah. 2018. Generating Reusable Web Components from Mockups. In *Proceedings of 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE '18)*. IEEE Computer Society.

[5] Mohammad Bajammal and Ali Mesbah. 2018. Web Canvas Testing through Visual Inference. In *Proceedings of 11th International Conference on Software Testing, Verification and Validation (ICST '18)*. IEEE Computer Society, 193–203.

[6] Robert V. Binder. 1996. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability* 6, 3-4 (1996), 125–252.

[7] Kai Briechle and Uwe D Hanebeck. 2001. Template matching using fast normalized cross correlation. In *Optical Pattern Recognition XII*, Vol. 4387. International Society for Optics and Photonics, 95–103.

[8] Roberto Brunelli. 2009. *Template Matching Techniques in Computer Vision: Theory and Practice*. Wiley Publishing.

[9] Gilles Burel and Dominique Carel. 1994. Detection and localization of faces on digital images. *Pattern Recognition Letters* 15, 10 (1994), 963 – 967.

[10] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI testing using computer vision. In *Proceedings of 28th ACM Conference on Human Factors in Computing Systems (CHI '10)*. ACM, 1535–1544.

[11] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of 1st International Workshop on End-to-End Test Script Engineering (ETSE '11)*. ACM, 24–29.

[12] Laurent Christophe, Reinout Stevens, Coen De Roover, and Wolfgang De Meuter. 2014. Prevalence and Maintenance of Automated Functional Tests for Web Applications. In *Proceedings of 30th International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE, 141–150.

[13] Cloc 2018. Counts blank lines, comment lines, and physical lines of source code in many programming languages. https://github.com/AlDanial/cloc. (2018).

[14] Brett Daniel, Danny Dig, Tihomir Gvero, Vilas Jagannath, Johnston Jiaa, Damion Mitchell, Jurand Nogiec, Shin Hwei Tan, and Darko Marinov. 2011. ReAssert: A Tool for Repairing Broken Unit Tests. In *Proceedings of 33rd International Conference on Software Engineering (ICSE '11)*. ACM, 1010–1012.

[15] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On Test Repair Using Symbolic Execution. In *Proceedings of 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, 207–218.

[16] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting Repairs for Broken Unit Tests. In *Proceedings of 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, 433–444.

[17] Brett Daniel, Qingzhou Luo, Mehdi Mirzaaghaei, Danny Dig, Darko Marinov, and Mauro Pezzè. 2011. Automated GUI Refactoring and Test Script Repair. In *Proceedings of First International Workshop on End-to-End Test Script Engineering (ETSE '11)*. ACM, 38–41.

[18] Yang Feng, James A. Jones, Zhenyu Chen, and Chunrong Fang. 2016. Multi-objective Test Report Prioritization Using Image Understanding. In *Proceedings of 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. ACM, 202–213.

[19] Mark Fewster and Dorothy Graham. 1999. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley Longman Publishing Co., Inc.

[20] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI Test Script Repair. *IEEE Transactions on Software Engineering* 42, 2 (feb 2016), 170–186.

[21] Z. Gao, C. Fang, and A. M. Memon. 2015. Pushing the limits on automation in GUI regression testing. In *Proceedings of IEEE 26th International Symposium on Software Reliability Engineering (ISSRE '15)*. 565–575.

[22] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *Proceedings of 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 408–418.

[23] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. WATERFALL: An Incremental Approach for Repairing Record-replay Tests of Web Applications. In *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, 751–762.

[24] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do Record/Replay Tests of Web Applications Break?. In *Proceedings of 9th International Conference on Software Testing, Verification and Validation (ICST '16)*. IEEE, 180–190.

[25] Si Huang, Myra B. Cohen, and Atif M. Memon. 2010. Repairing GUI Test Suites Using a Genetic Algorithm. In *Proceedings of 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, 245–254.

[26] Ayman Issa, Jonathan Sillito, and Vahid Garousi. 2012. Visual Testing of Graphical User Interfaces: An Exploratory Study Towards Systematic Definitions and Approaches. In *Proceedings of IEEE 14th International Symposium on Web Systems Evolution (WSE '12)*. IEEE Computer Society, 11–15.

[27] M. Furkan Kıraç, Barış Aktemur, and Hasan Sözer. 2018. VISOR: A fast image processing pipeline with scaling and translation invariance for test oracle automation of visual output systems. *Journal of Systems and Software* 136 (2018), 266 – 277.

[28] V. Lelli, A. Blouin, and B. Baudry. 2015. Classifying and Qualifying GUI Defects. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. 1–10.

[29] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2014. Visual vs. DOM-based Web Locators: An Empirical Study. In *Proceedings of 14th International Conference on Web Engineering (ICWE '14)*, Vol. 8541. Springer, 322–340.

[30] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. 2016. Approaches and Tools for Automated End-to-End Web Testing. *Advances in Computers* 101 (2016), 193–237.

[31] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing Web Test Cases Aging by means of Robust XPath Locators. In *Proceedings of 25th International Symposium on Software Reliability Engineering Workshops (ISSREW '14)*. IEEE Computer Society, 449–454.

[32] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Automated Migration of DOM-based to Visual Web Tests. In *Proceedings of 30th Symposium on Applied Computing (SAC '15)*. ACM, 775–782.

[33] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2015. Using Multi-Locators to Increase the Robustness of Web Test Cases. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. IEEE, 1–10.

[34] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2016. ROBULA+: An Algorithm for Generating Robust XPath Locators for Web Testing. *Journal of Software: Evolution and Process* (2016), 28:177–204.

[35] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2018. PESTO: Automated migration of DOM-based Web tests towards the visual approach. *Software Testing, Verification And Reliability* 28, 4 (2018).

[36] D. G. Lowe. 1999. Object recognition from local scale-invariant features. In *Proceedings of 7th IEEE International Conference on Computer Vision*, Vol. 2. 1150–1157.

[37] David G. Lowe. 2004. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision* 60, 2 (01 Nov 2004), 91–110.

[38] S. Mahajan and W. G. J. Halfond. 2015. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *Proceedings of 8th IEEE International Conference on Software Testing, Verification and Validation (ICST '15)*. 1–10.

[39] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A. Efros. 2011. Ensemble of exemplar-SVMs for Object Detection and Beyond. In *Proceedings of 2011 International Conference on Computer Vision (ICCV '11)*. IEEE Computer Society, 89–96.

[40] Atif M. Memon. 2008. Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing. *ACM Transactions on Software Engineering and Methodologies* 18, 2, Article 4 (nov 2008), 36 pages.

[41] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web* 6, 1 (2012), 3:1–3:30.

[42] Ali Mesbah, Arie van Deursen, and Danny Roest. 2012. Invariant-based Automatic Testing of Modern Web Applications. *IEEE Transactions on Software Engineering* 38, 1 (2012), 35–53.

[43] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21, 1 (2014), 65–105.

[44] OpenCV 2018. Open Source Computer Vision Library. https://opencv.org. (2018).

[45] Rudolf Ramler and Klaus Wolfmaier. 2006. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In *Proceedings of 1st International Workshop on Automation of Software Test (AST '06)*. ACM, 85–91.

[46] R. Ramler and T. Ziebermayr. 2017. What You See Is What You Test - Augmenting Software Testing with Computer Vision. In *Proceedings of 10th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2017)*. 398–400.

[47] Edward Rosten and Tom Drummond. 2005. Fusing points and lines for high performance tracking. In *IEEE International Conference on Computer Vision*, Vol. 2. 1508–1511.

[48] Edward Rosten, Reid Porter, and Tom Drummond. 2010. FASTER and better: A machine learning approach to corner detection. *IEEE Transaction on Pattern Analysis and Machine Intelligence* 32 (2010), 105–119.

[49] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. 2013. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proceedings of 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, 702–711.

[50] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, 1–10.

[51] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2014. PESTO: A Tool for Migrating DOM-based to Visual Web Tests. In *Proceedings of 14th International Working Conference on Source Code Analysis and Manipulation (SCAM '14)*. IEEE Computer Society, 65–70.

[52] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2015. Why Creating Web Page Objects Manually If It Can Be Done Automatically?. In *Proceedings of 10th IEEE/ACM International Workshop on Automation of Software Test (AST '15)*. IEEE/ACM, 70–74.

[53] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2016. Automatic Page Object Generation with APOGEN. In *Proceedings of 16th International Conference on Web Engineering (ICWE '16 - Demo Track)*. Springer, 533–537.

[54] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2016. Clustering-Aided Page Object Generation for Web Testing. In *Proceedings of 16th International Conference on Web Engineering (ICWE '16)*. Springer, 132–151.

[55] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2017. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal* 25, 3 (Sept. 2017), 1007–1039.

[56] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Web Test Repair Using Computer Vision. In *Proceedings of 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018 - Demonstration Track)*. ACM.

[57] Paolo Tonella, Filippo Ricca, and Alessandro Marchetto. 2014. Recent Advances in Web Testing. *Advances in Computers* 93 (2014), 1–51.

[58] Vista 2018. Web Test Repair using Computer Vision. https://github.com/saltlab/vista. (2018).

[59] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust Test Automation Using Contextual Clues. In *Proceedings of 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 304–314.

[60] Sai Zhang, Hao Lü, and Michael D. Ernst. 2013. Automatically Repairing Broken Workflows for Evolving GUI Applications. In *Proceedings of 2013 International Symposium on Software Testing and Analysis (ISSTA '13)*. ACM, 45–55.