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:


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.
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 \( st_j \), but another statement \( st_i \), preceding \( st_j \) (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
We were taken into account.

which descriptive labels were also assigned.

Visual Web Test Repair

3.3 Test Breakages and How To Repair Them

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

an analogous data collection study [24]. For each subject, and for each version $V_i$ and its accompanying test suite $T_i$, we executed $T_i$ on the subsequent version $V_{i+1}$. 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 taken into account.

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

Table 1: Subject systems and their test suites

<table>
<thead>
<tr>
<th>Web Applications</th>
<th>Releases (#)</th>
<th>LOCs ($)</th>
<th>Test Suites</th>
<th>LOCs ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddressBook</td>
<td>48</td>
<td>8,757</td>
<td>1,344/28</td>
<td>61,826/43</td>
</tr>
<tr>
<td>Claroline</td>
<td>12</td>
<td>338,129</td>
<td>492/41</td>
<td>20,287/43</td>
</tr>
<tr>
<td>Collaborative</td>
<td>14</td>
<td>150,696</td>
<td>560/40</td>
<td>22,710/38</td>
</tr>
<tr>
<td>PPMA</td>
<td>12</td>
<td>556,164</td>
<td>276/23</td>
<td>16,732/47</td>
</tr>
<tr>
<td>Total/Average</td>
<td>86</td>
<td>263,436</td>
<td>2,672/33</td>
<td>121,555/43</td>
</tr>
</tbody>
</table>

Let us consider the login page of AddressBook in Figure 1a, and the accompanying WebDriver test θ. 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 θ 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 $l : D \rightarrow \emptyset \land e \notin D \land \exists D' \in \text{neigh}(D) \mid l : D' \rightarrow \{e\}$.

As a concrete example consider Figure 1a, specifically the pages in which the user can insert a new entry. The test θ 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 subsequent version 7.0.0 θ, 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 5. 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 $e' \in st_j$ such that $(e', st_j) \rightarrow st_j$ (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 $l : D \rightarrow \emptyset \land \exists D' \in \text{neigh}(D) \mid l : D' \rightarrow \{e\}$. 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 θ 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.
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, \( l : D \rightarrow \{ e \} \) in \( V \), and \( l : D' \rightarrow \{ e' \} \) in \( V' \) where \( e \neq e' \).

Consider Figure 1 again. Suppose that the test \( \#4 \) is repaired so as to reach the edit page on version 7.0.0 (for instance, as in \( \#6 \)). 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 fail 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 \( l' : D' \rightarrow \{ e \} \).

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 subcategories 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 (387/733), 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.

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). A tester would run \( TS \) by means of the first module of the presented approach, the Visual Execution Tracer \( \Theta \). Such a module collects, for each test, a variety of visual information (e.g., screenshots). Then, as the application \( V \) evolves into a new version \( V' \), a tester may wish to test \( TS \) to check if regressions have occurred. To this aim, the tester would use the second module of our approach, the Visual Test Repair \( \Theta \), 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 \( \Theta \). 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].
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 `visualSearch` 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.
These changes may render our technique fragile, because the visual (i.e., the captured template can be shifted with respect to the refer-
ence 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 $\mathcal{S}$, 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 (a), 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 (b) 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 (c). 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 $\text{centre}$ 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

Figure 5: Computer vision pipeline for robust web element detection (left) and visual-based local crawling for workflow repair.
web page. If a match is found, the workflow is repaired by adding a transition to that page through the matched element (Figure 5 (d) shows how the local crawling approach works for the example of Figure 1b (6)). 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 (Visual Test 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:

**RQ1 (effectiveness):** How effective is the proposed visual approach at repairing test breakages compared to a state-of-the-art DOM-based approach?

**RQ2 (performance):** What is the overhead and runtime of executing the visual approach compared to the DOM-based one?

**RQ3:** How is repaired by adding a transition to the page through the matched element. The **efficiency** of the repair is evaluated by comparing the baseline repair time to the repair time obtained by our approach.

**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 determine 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 (RQ1).** 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

<table>
<thead>
<tr>
<th>Breakages (#)</th>
<th>AddressBook</th>
<th>Claroline</th>
<th>Collabtive</th>
<th>PPMA</th>
<th>All Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rep. (%)</td>
<td>96</td>
<td>117</td>
<td>84</td>
<td>188</td>
<td>417</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>162</td>
<td>31</td>
<td>50</td>
<td>17</td>
<td>45</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Breakages (#)</th>
<th>AddressBook</th>
<th>Claroline</th>
<th>Collabtive</th>
<th>PPMA</th>
<th>All Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rep. (%)</td>
<td>17</td>
<td>47</td>
<td>166</td>
<td>187</td>
<td>417</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>28</td>
<td>72</td>
<td>43</td>
<td>32</td>
<td>175</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Rep. (%)</td>
<td>5</td>
<td>43</td>
<td>9</td>
<td>81</td>
<td>138</td>
</tr>
</tbody>
</table>

Performance (RQs). 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 trace a test suite, the repair time, and the total running time, in seconds, for each approach. We compared the visual-based approach to a DOM-based approach, which employed nearly 4 minutes more. In the latter case, 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 on these 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

<table>
<thead>
<tr>
<th>Trace Generation</th>
<th>Repair Time</th>
<th>AddressBook</th>
<th>Claroline</th>
<th>Collabtive</th>
<th>PPMA</th>
<th>All Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>seconds</td>
<td>% correct</td>
<td>incorrect</td>
<td>seconds</td>
<td>% correct</td>
<td>incorrect</td>
<td>seconds</td>
</tr>
<tr>
<td>Original</td>
<td>With VET</td>
<td>Slowdown</td>
<td>WATER</td>
<td>VET</td>
<td>WATER</td>
<td>VET</td>
</tr>
<tr>
<td>AddressBook</td>
<td>50</td>
<td>207</td>
<td>4.1</td>
<td>127</td>
<td>388</td>
<td>594</td>
</tr>
<tr>
<td>Claroline</td>
<td>155</td>
<td>861</td>
<td>5.6</td>
<td>600</td>
<td>767</td>
<td>5,571</td>
</tr>
<tr>
<td>Collabtive</td>
<td>622</td>
<td>1,780</td>
<td>2.9</td>
<td>1,930</td>
<td>1,913</td>
<td>621</td>
</tr>
<tr>
<td>PPMA</td>
<td>228</td>
<td>458</td>
<td>2.0</td>
<td>802</td>
<td>646</td>
<td>2,972</td>
</tr>
</tbody>
</table>

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 mis-selected element is among those retrieved by its heuristics. Water, 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 Water 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 Collaborative 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 Collaborative, 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 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).

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 ($\frac{192.15}{0.66}$).

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 oracles, where testers can manually inject visual checks at specific places of the test’s execution. This has two drawbacks: (1) inserting the expected images 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 need 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 [25] 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 activities 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 WreaDiff [49] and XPERT [50] utilize an image similarity technique based on color 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. Ramlar and Ziebemayr [46] use visual test automation joint with computer vision techniques to test physical properties of a mechatronic system, whereas Kiraç 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


