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 when 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 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 execute. 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:

ACM Reference Format:

1 INTRODUCTION
Test automation techniques are used to enable end-to-end (E2E) functional testing of web applications [54]. 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 [4, 17, 19, 41, 43].

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 the literature, instances of these problems are known as test breakages [13–15, 22]. 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 [22]. Web element locators have emerged as the main cause of fragility of tests, confirming previous anecdotal findings [15, 28].

Existing automated web test repair techniques [9, 21] essentially consider only the DOM 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 of 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.

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 [55].

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 TRAVELOGE

E2E web tests are known for being fragile in the face of software evolution [22, 28]. 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 unprecenditadely. 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 [22]. 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 changes to the web page structure, that may render tests inapplicable, just because the locators (and not the tests themselves) become ineffective [9, 22, 28]. 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 [22]. A breakage is called direct when the test stops at a statement $s_1$, and $s_1$ 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 $s_j$, but another statement $s_k$, preceding $s_j$ (i.e., $1 > j$), must be repaired in order to let 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 the recent years [9, 13–15, 18]. In the web domain, the state-of-the-art test repair algorithm is Water [9], 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 [9]. 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 [15, 22], we focus our analysis on locator problems. Existing taxonomies related to web and GUI breakages [22, 24, 26] lack a temporal quantification of breakages, even less they propose approaches to solve 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 from our study highlight the complexity and the variety of 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 for which (i) multiple versions and (ii) 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 [11]. 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 an analogous data collection study [22]. 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 [32]. 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 Collaborative, and 300 for PPMA.

### 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 $D$. Notationally, $l : D \rightarrow \{e\}$ where $e$ is the web element returned by the locator $l$ when applied to $D$.

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

As an example, consider the login page of AddressBook in Figure 1a, and the accompanying Web Driver test 2. In the new subsequent version of AddressBook (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 2 will then stop at Line 4 when 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 where 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 (1), specifically the pages in which the user can insert a new entry. The test 2 clicks on the “add new” link on the home page (Line 5), and fills in the “First name”, “Last name” and “Company” text fields (Lines 6–8). Suppose now to replay this test on the successive version 7.0.0 (2), for regression purposes. The test will raise an exception of kind NoSuchElementException at Line 6, when attempting to locate the “First name” text field 2. Indeed, a new intermediate confirmation page has been added 3, 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 $s_t$ in version $V$, (2) try to match the $e$ in one of the neighbouring states of $s_t$ in the new version $V'$, which in turn requires to (3) find a web element $e' \in s_{t'}$ such that $(e', s_{t'}) \rightarrow s_{t}$ (the “Next” button in our example 3).

**Non-Selection • Removed.** The third and last non-selection scenario concerns a web element being removed from a web page. Formally, $l : D \rightarrow \emptyset \land \exists D' \in \text{neigh}(D) \mid l : D' \rightarrow \{e\}$. As a concrete example, consider the example of Figure 1b, with the application being evolved in the reverse order as depicted in the figure (thus considering going from version 7.0.0 to version 6.2.12). The test 4 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.

**Table 1: Subject systems and their test suites**

<table>
<thead>
<tr>
<th>Web Applications</th>
<th>Test Suites</th>
</tr>
</thead>
<tbody>
<tr>
<td>Releases (#)</td>
<td>LOCs ($)</td>
</tr>
<tr>
<td>AddressBook</td>
<td>48</td>
</tr>
<tr>
<td>Claroline</td>
<td>12</td>
</tr>
<tr>
<td>Collaborative</td>
<td>14</td>
</tr>
<tr>
<td>PPMA</td>
<td>12</td>
</tr>
</tbody>
</table>

**Figure 1: AddressBook web application, version 6.2.12 (a) and version 7.0.0 (b), along with Selenium WebDriver tests.**
Propagated will fill in the field “Nickname”, instead of the field “Company”.

whereas at Line 8 (which will be Line 9 in the new version) the test shows box-plots of the distribution of such breakages per test in ≈breakages. We have not observed silent breakages in our bench-

2ments (38/733), whose repairs were always found within the same

Page the vast majority were found on the

ing direct breakages in the tests. Concerning the associated repairs,

Non-Selection category concerns

categories, each of which has specific subtypes. The most prevalent

breakages and relative repairs. Our study revealed two major cat-

Figure 2 shows the distribution of the different classes of locator

breakages and relative repairs. Our study revealed two major cat-

3.4 Findings

Figure 2 shows the distribution of the different classes of locator

breakages and relative repairs. Our study revealed two major cat-

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 $V_k$ 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 for 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) $\odot$. A tester

would run $TS$ by means of the first module of the presented approach, the

Visual Execution Tracer $\odot$. Such a module collects, for each test,

a variety of visual information (e.g., screenshots) $\odot$. Then, as the

application $V$ evolves into a new version $V'$, a tester may wish to

use $TS$ to check if regressions have occurred $\odot$. To this aim, the

tester would use the second module of our approach, the Visual

Test Repair $\odot$, 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 $\odot$. 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 dy-
namic 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 for it. A

visual locator is the portion of the rendered web page that uniquely

identifies that web element on the screen [33].
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 4–50). 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 screenshot 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 the 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 4.4). 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 it 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 both are returned. In the following sections, we describe the visualSearch and localCrawling procedures that underlie at the functioning of our approach.
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 [6]. 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 [34, 35] and FAST [45, 46]. 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 [35]. 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 version of the web page. For further details, the interested reader is referred to the relevant literature [34, 35, 45, 46].

**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 [42]. We found the Fast Normalized Cross Correlation algorithm [5] 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 [37] (basically NMS assumes highly overlapping detections as belonging to the same object [7]). Thus, only the closest match is returned (see the green thick rectangle over the “Login” button).

In brief, 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.** Finally, in order to retrieve the DOM-element corresponding to a specific point of coordinates $(x, y)$, visualSearch queries the browser through JavaScript to get 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).

4.4 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 ineffectual for a tester to quickly explore this 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 (Figure 6). If a match is found, the workflow is repaired by adding a transition to that page through the matched element (Figure 6 shows how the local crawling approach works for the example of Figure 1b @). Otherwise, our approach...
4.5 Implementation

We implemented our approach in a tool called Vista (Visual Test Repair), which is publicly available [55]. 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 [31], runs the visual detection algorithms using the open-source computer vision library OpenCV [42]. 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 [39, 40] 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.

5 EMPIRICAL EVALUATION

We consider the following research questions:

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

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

RQ1 aims to evaluate the effectiveness of Vista at detecting and repairing test breakages, and how this varies across the breakage classes. RQ2 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 [9]. 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 [9]) within our main algorithm. For simplicity, in the evaluation section, we refer to the two competing tools as Vista and Water.

**Dependent Variables.** To measure effectiveness (RQ1), 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 (RQ2), 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 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 proposed visual approach by Vista was constantly superior to Water, no matter the application being considered. 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 breakages that Water was able to repair correctly that Vista was not. In the second row, instead, are the cases in which Vista prevailed, whereas the third row reports the cases in which both tools were able to correct the breakages. Lastly, the last fourth index indicates the number of breakage that neither tool was able to correct, and thus we had to repair manually. Additionally,

no breakages were found to be unrepairable, or tests needed to be discarded. As evident from Table 3, in more than half of the cases (56%), both tools found the correct repair. However, there are 175 cases in which 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 in order to continue the experimental work, because neither technique was able to fix them. We will discuss those cases in Section 6.

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

<table>
<thead>
<tr>
<th></th>
<th>AddressBook</th>
<th>Claroline</th>
<th>Collabtive</th>
<th>PPMA</th>
<th>All Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vista</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Water</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vista</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Water</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vista</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Water</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vista</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Water</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Vista</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Water</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Performance results

<table>
<thead>
<tr>
<th></th>
<th>Trace Generation</th>
<th>Repair Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(seconds)</td>
<td>% correct</td>
</tr>
<tr>
<td><strong>Original</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>With VET</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Slowdown</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AddressBook</strong></td>
<td>50</td>
<td>207</td>
</tr>
<tr>
<td><strong>Claroline</strong></td>
<td>155</td>
<td>861</td>
</tr>
<tr>
<td><strong>Collabtive</strong></td>
<td>622</td>
<td>1,780</td>
</tr>
<tr>
<td><strong>PPMA</strong></td>
<td>228</td>
<td>458</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1,055</td>
<td>3,306</td>
</tr>
</tbody>
</table>

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. This is why repairing web tests is often considered a tedious activity that leads the test suites to be abandoned [10]. 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 Vista can repair more than 80% of the total number of breakages correctly, outperforming Water in 175 cases.

We investigated why all breakages were not repaired. We enumerate some of the main reasons next, which pertain to both 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 Water’s heuristics.
Visual Web Test Repair

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 of Water was ineffective when searching elements with local crawling (Neighbouring Page), 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 dropdown menu, whereas the crawler is designed 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. Only in three cases (for the Claroline application), Water was able to find a correct repair when Vista did not. These cases concern 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 16% 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 button to icon). 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 were at a level that did not jeopardize the effectiveness of Water. Also Vista had its 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 impacted nor modifies the source code of the test cases. On the contrary, our technique can be introduced smoothly in existing testing infrastructures, as it neither clutters the initial test code, with statements that do not pertain to the check-points must be performed manually, (2) this extra-code 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 root cause analysis.

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 the 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 is 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 study, we evaluated whether our technique has prospect for success against a state-of-the-art repair algorithm.

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 [21], 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 (182 vs 260).

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 the 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 is 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 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 behaviours of their tests 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 [29, 31, 32]. Finally, our taxonomy can drive the design of other test repair techniques.
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.

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

7 RELATED WORK

Web and GUI Test Repair. We already discussed and evaluated WATER [9]. A recent work [21] adopted WATER to repair the breakages happening to the intermediate minor versions between two major releases of a web application. Grechanik et al. [20] 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. [57] describe FLOWFIXER, a tool that repairs broken GUI workflows in desktop applications. Memon [38] presents an event-based model of a GUI which is used to determine the modifications during software evolutions. In similar fashion, Gao et al. [18] 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 [12–15]. Huang et al. [23] 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, 29, 31, 32, 56]. A study by Leotta et al. [27] discusses the development and maintenance of DOM and visual test suites. Additionally, Stocco et al. [50–53] 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. The use of CV techniques to assist common software engineering tasks is getting popular in the recent years. One of the foundational works on computer vision applied to testing is by Chang and colleagues. Their tool SIKULI [8] 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. Feng et al. [16] use a combination of visual analysis and NLP techniques to assist the inspection of crowdsourced test reports. PESTO [30, 33, 49] migrates DOM-based web tests to visual tests. Ramler and Ziebemayr [44] have illustrated the use of visual test automation joint with computer vision techniques to test physical properties of a mechatronic system, whereas Kıraca and colleagues [25] used an image processing pipeline for test oracle automation of visual output systems. Finally, CV techniques have been employed to detect cross-browser incompatibilities (XBs). WEBSER [56] compares whole images with a perceptual difference technique, whereas WEBDIFF [47] and X-PERT [48] utilize an image similarity technique based on image colour histogram matching.

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 the 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


