VISTA: Web Test Repair using Computer Vision

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

Repairing broken web element locators represents the major maintenance cost of web test cases. To detect possible repairs, testers typically inspect the tests' interactions with the application under test through the GUI. Existing automated test repair techniques focus instead on the code and ignore visual aspects of the application. In this demo paper, we give an overview of VISTA, a novel test repair technique that leverages computer vision and local crawling to automatically suggest and apply repairs to broken web tests. URL: https://github.com/saltlab/Vista

CCS CONCEPTS

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

KEYWORDS

web testing, test repair, computer vision, image analysis

ACM Reference Format:
Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. VISTA: Web Test Repair using Computer Vision. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264592

1 INTRODUCTION AND MOTIVATION

Automated web tests created with tools such as Selenium are renown for being fragile as the the web application under test evolves [6]. Researchers have singled out web element *locators* as the main cause of fragility [3]. Locators are commands used by test automation tools to identify elements on a web page, hanging on specific properties found in the Document Object Model (DOM), such as the element's identifier, XPath, or text.

Test Breakage Problem. Unfortunately, the DOM tends to be a quite volatile structure, which is massively updated both for evolution and cosmetic purposes. Even simple modifications such as elements repositioning can negatively impact the mapping between locators and web elements, making tests inapplicable. In literature, instances of these problems are referred to as test breakages [3]. A broken test is different from a failing test, because the natural software evolution is the cause of the test's malfunction rather than the presence of bugs in the production code. Thus, the repair activity must be triggered on the test code, rather than on the application's.

How Testers Repair. While repairing locators might seem a fairly mundane task, it instead accounts for a number of different scenarios that makes it quite challenging and time-consuming [10]. When a test t that was used to function on a version V_1 breaks on a successive version V_2, a tester needs to understand the *root cause* behind the breakage and a possible *fix* for it. This process involves at least four steps. (1) The tester inspects the error stack trace or the console, which may contain information about the origin of breakage (e.g., "`NoSuchElementException` *occurred. Unable to locate element with* `name=password`"). (2) The tester inspects t, looking for the statement st related to the error message. (3) The tester browses the GUI of V_2, trying to identify the portion of GUI related to st. (4) The tester inspects either the DOM, or the GUI, or both the DOM and the GUI of V_2 to find potential fixes. While doing so, the tester may possibly need to manually exercise the same broken scenario of t (i.e., all the actions in the statements preceding st), in order to replicate the breakage occurred at st and gather insights on possible repairs.

Challenges of Manual Repair. A *first challenge* in repairing web tests derives from the fact that testers often need to inspect and link the test code behaviour with the modifications perpetrated to the GUI and the DOM of the evolved application. In other words, breakages are often repaired by finding candidate solutions through the inspection of the DOM and the GUI *at the same time*. For this reason, it is arguably more challenging and time-consuming to repair Selenium tests than standard JUnit tests for desktop applications, for which the error messages are typically more informative and IDE features make debugging activities easier.

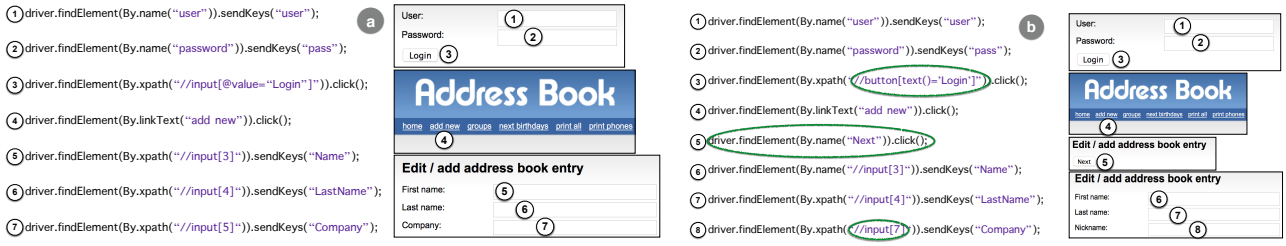
A *second challenge* is related to the time needed to correct such breakages, which can be significant [2, 4]. One of the main reasons is due to the low tooling support by existing test automation frameworks in understanding the root causes behind test breakages and how they do relate with the changes made in the web applications.

The Idea. Our insight is using the GUI and visual technologies to support the detection of breakages, by checking the GUI actions performed by the tests and validating them at runtime (in a similar way as testers do), timely detecting deviations from the correct behaviour. In this way, we can anticipate the occurrence of breakages, and trigger repair procedures that suggest potential fixes to the testers. Existing locator repair techniques [1, 2] are limited when the web application undergoes drastic structural changes because they only consider the DOM as a source where to find repairs.

The Tool. Our tool VISTA uses the visual information obtained by the tests' execution and, along with image processing and crawling techniques, supports the automated repair of locator breakages.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264592



(a) *Version 6.2.12* (b) *Version 7.0.0*

Figure 1: AddressBook web application evolution (v. 6.2.12 → v. 7.0.0), along with Selenium WebDriver test cases.

2 RUNNING EXAMPLE

We consider a real test regression scenario and explain the limitations of WATER [1], an existing DOM-based web test repair solution, which motivate the need for a novel visual-based approach.

Figure 1 shows two versions of the AddressBook web application, one of the experimental subjects used in our empirical study [10]. We consider a test scenario in which a new entry is added to the address book. In the first version 6.2.12 (Figure 1a), the test ⓐ logs into the application (Lines 1–3), clicks on the "add new" link (Line 4) and fills in the form with the new user information (Lines 5–7).

When the new version of AddressBook is released, a tester may wish to run this test to check whether regressions have occurred into the application during development. However, when executed on version 7.0.0 (Figure 1b), the test ⓐ will cease to function because multiple breakages are present.

Non Selections. First, the execution will stop at Line 3, when attempting to locate the "Login" button, because the attribute `value` has been removed from the HTML. At a visual inspection of the two GUIs, however, a tester would expect the test to work, because her perception is immaterial where changes at DOM-level are concerned. Indeed, it is evident that the target element (i.e., the "Login" button) is *visually* still present on the page, and its position *on the GUI* has not changed.

This is a simple instance of **direct breakage**, because the test scenario is unaltered, and no bugs are (eventually) present in the application. However, the test is inapplicable because the synchronization with the application is lost, and a fix needs to be found. To this aim, a tester may wish to use WATER to automatically fix the broken statement at Line 3. Specifically, another locator for the "Login" button ③ needs to be generated, rather the relying on the "broken" attribute `value`. WATER will attempt to gather information about the broken element (such as the XPath, and the various attributes) by analysing the DOM of the previous version 6.2.12, and match such information on the evolved DOM of version 7.0.0. Unfortunately, WATER's technique is ineffective in this case, because (i) the attribute `value` has been deleted from the DOM, and (ii) both the XPath and the tag of the target element have changed (from `input` to `button`), which render impossible for WATER's heuristic to identify it on the evolved DOM and apply its automatic repair.

Broken Workflows. A second non-trivial breakage happens at Line 5. When attempting to locate the "First name" text field, the test will raise an exception of kind `NoSuchElementException`. Indeed, a new intermediate confirmation page has been added (Figure 1b), and the navigational workflow of the test must be corrected to reflect that of the new modified 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 4. 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 raised by JUnit, it is challenging for the tester to detect this problem, unless the *visual execution* of the test is taken into consideration. Even the use of WATER is unsuccessful, because the tool would attempt to repair the broken statement at Line 5 (the technique only handles addition of statements within forms, and does not apply to general broken workflow scenarios).

Mis Selections. Lastly, the statements at Lines 5–6 will execute correctly, whereas the statement at Line 7 will fill the field "Nickname", instead of the field "Company". In literature, this is known as a mis-selection problem [1, 10]. Mis-selection of web elements can lead to unpredictable test executions, that diverge from the test's intended behaviour. Depending on the kind of actions being performed, the test's execution might continue 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* (**propagated breakage**). WATER is not designed to detect mis-selections; however those scenarios are very challenging to detect for a tester, because only at a manual visual inspection of the test's execution, one can recognize such breakage patterns.

Repaired Test. Figure 1b ⓑ shows the test as repaired by VISTA (repairs are highlighted), that works correctly on AddressBook version 7.0.0. Specifically, (i) the non-selection and mis-selection are corrected by updating the locator component of the test statements (Lines 3 and 5), and (ii) the broken workflow is corrected by adding a new test statement to reach the new page (hence creating the missing transition).

In the following of this demo paper, we illustrate our tool design and implementation.

3 VISTA TOOL

Our intuition behind the development of VISTA is that an algorithm taking into consideration the visual execution of the tests might be able to validate the feasibility of DOM-based locators through their visual appearance, potentially anticipating the occurrence of breakages. Additionally, the visual locators can be also used to match the target element in the new evolved GUI. An assumption

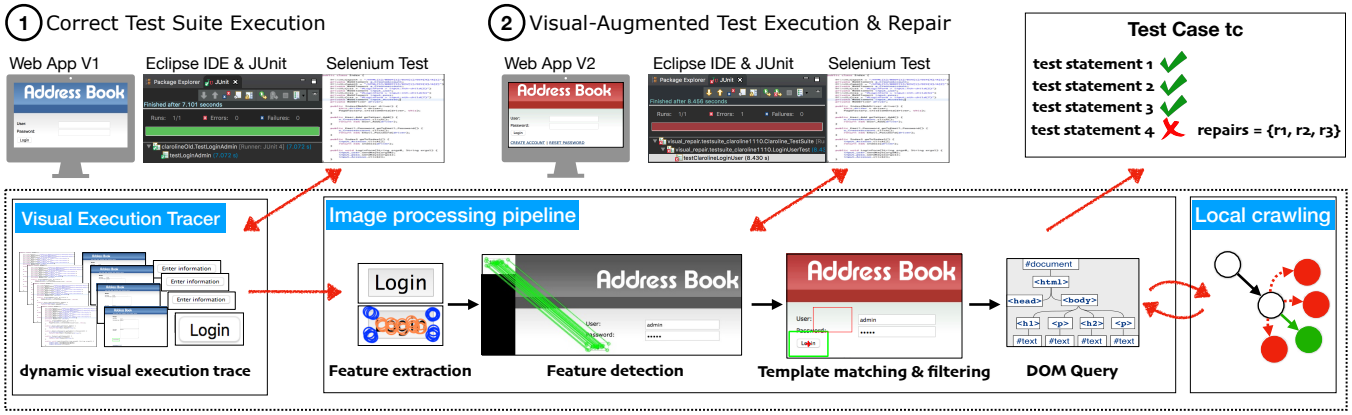


Figure 2: High-Level Architecture of VISTA

of this work is that the GUI of a web application is less prone to be drastically changed between two consecutive releases, whereas the DOM gets updated more frequently. However, matching web elements between two GUIs is challenging and several issues needs to be solved. Among all (1) finding an accurate visual matching technique that can handle multiple visual matches (visual false positives), and, in the case a good visual match is found, (2) retrieving the corresponding element in the DOM. (3) Lastly, in the case of broken workflows, it would be desirable to automate the local exploration of the application's state space, looking whether the target element has repositioned to another test state.

3.1 Tool Architecture

Figure 2 shows the high level architecture of VISTA, which is logically composed by two main modules: the *Visual Execution Tracer* and the *Visual-Augmented Test Runner*. VISTA is written in Java and executes Selenium test cases within the Eclipse IDE analysing their visual execution trace to detect the occurrence of locator breakages and finding potential fixes *at runtime* to report to the users for inspection.

In the following we explain the two modules on the running example described in Section 2.

3.2 Visual Execution Tracer

In the first phase, our tool records the visual interactions of each test statement with a correct version of the application (Web App V1). Keeping the association between statements, DOM locators, and their visual appearance is important because it is close to the *mental model* that testers create when they manually validate the execution of the tests through, for instance, eye-balling.

Such a mapping can be captured only at runtime, while tests execute, because the visual appearance of the rendered elements may change during the application's execution and some elements may be not visible until specific events occur. To this aim, the Visual Execution Tracer integrates the tool PESTO [7, 9] that uses aspect-oriented programming to intercept Selenium WebDriver method calls (e.g., `click()`) and automatically creates visual locators for each web element composing the test cases. A visual locator is the portion of the rendered web page that *uniquely* identifies that web element on the screen [7].

For instance, for the test breakage at Line 3 in Figure 1a, the login submit button (in the HTML `<input value=''Login''>`) is identified through the XPath locator `//input[@value=''Login'']`. The tool (i) saves the entire screenshot of the web page, (ii) retrieves the web element coordinates and sizes through WebDriver, and finally (iii) crops a rectangle image centred on the web element. Note that a visual locator is not always the precise crop of the web element's bounding box. PESTO can also manage cases in which a larger crop—taking into account the web element's visual context—is necessary in order to visually differentiate it from other visually similar web elements appearing on the page (e.g., multiple text fields in a form).

When the test execution terminates, all this information (test statements, corresponding screenshots and visual locators) is made persistent as a `json` file and used by the second main component of the tool.

3.3 Visual-Augmented Test Runner

In the second phase, VISTA runs the tests on the new evolved version of the application (Web App V2). The Visual-Augmented Test Runner executes each test statement in a controlled loop environment in which the result of the action performed by the statement on the web application is validated by a series of steps.

First, the tool pools the DOM of the application with the original locator `//input[@value=''Login'']` to observe if an instance of a `WebElement` object is returned by WebDriver. In case of **non-selections** (e.g., no web elements are associated with the locator in the new DOM), VISTA attempts at verifying if the web element is still visually present on the web page (by means of the visual locator saved before), and if so, it generates a new locator (see Section 3.4).

Conversely, if an element is retrieved by the original DOM locator, a further sanity check is performed, still relying on the visual search of the web element. VISTA checks the equivalence of the two `WebElement` objects retrieved by the two locators: if they do target the same web element, the approach has *visually validated* the test statement, which is executed. Then, the approach proceeds to validating the next statement.

In case of disagreement between the visual and DOM locators, a possible case of **mis-selection** might have occurred, and Vista outputs the result to the tester, who needs to resolve the dispute by selecting the correct locator (if any).

A third breakage scenario occurs when neither the DOM nor the visual locator is able to select any web element in the current DOM (test state). This is the case of Line 5 of Figure 1a, in which the target web element has repositioned to a new web page. In this case, Vista triggers a local crawling of the state space of the web application, looking for matches which are one-level distant from the current page. If a match is found in any of the web pages, the workflow is repaired by adding a transition to that page generating a new statement (i.e., a locator) for the matched element.

If all these validation checks are not successful, Vista assumes the web element as being removed from the application and suggests the deletion of the statement to the user.

3.4 Key Components of Vista

For the development of the visual component of Vista, we have pipelined different algorithms available from the open-source computer vision library OpenCV (version 2.4.9) into a custom detector. The detector aims at assessing the presence of the visual locator in the new DOM, and, if so, at searching for the best visual match in the GUI and its correspondent DOM element. In the following we briefly illustrate each step.

3.4.1 Image Processing Pipeline. The image detector combines two feature detection algorithms, SIFT and FAST. The detector extracts the key-points from the template image using SIFT descriptors, and then adopts a Flann-based descriptor matcher with a distance threshold ratio of $\gamma = 0.8$. If at least 70% of the key-points are matched, the Fast Normalized Cross Correlation template matching algorithm with a similarity threshold $\delta = 0.99$ is executed.

In case of multiple or false visual matches, our procedure discards the matches that do not fall in the region where the key-points have been found through a non-maxima suppression (NMS) operation. In this way, only the *closest* match is returned (see the green thick rectangle over the "Login" button in Figure 2).

3.4.2 From GUI to DOM. Once the best visual match has been found, we still need to retrieve the correspondent DOM element whose bounding box centre has coordinates (x, y). This operation can be done in different ways, such as parsing the DOM into a spatial structure (e.g., a R-Tree), for easier querying. Willing to provide a runtime validation technique, this solution failed to provide acceptable performance results in our exploratory experiments because the parsing operation is costly, and its complexity scales up with the number of elements in the DOM tree.

Thus, Vista simply queries the browser through the JavaScript command `elementFromPoint(x,y)` that returns the DOM element whose bounding box contains x and y. Those parameters need to specify the centre of the bounding box otherwise a DOM ancestor of the searched web element—as a `form` or `div` container—will be erroneously returned.

3.4.3 Locator Generator. The XPath of the retrieved web element can already be considered a valid repair for locator breakage. However, Vista can synthesize different DOM locators based on the attributes of the element itself, such as `id`, or `name`, discarding attributes considered fragile and prioritizing the final list based on the alleged robustness [5, 6].

3.4.4 Local Crawling for Workflow Repair. For the local crawling exploration, Vista features a Crawljax [8] plugin that incorporates the image processing pipeline. In this way, the crawler can search the desired web element visually, thus looking for repairs in the neighbourhood of the breakage site.

For an empirical evaluation of Vista in repairing the breakages of different breakage classes, we refer the reader to our full paper [10]. Vista was able to provide correct repairs for 81% of breakages, with a 41% increment over Water.

4 CONCLUSIONS AND FUTURE WORK

In this paper we described Vista, a novel web test repair technique based on a fast image-processing pipeline. While Vista has shown promising results [10], we are considering several improvements.

For future work, we plan to investigate alternative visual techniques such as OCR, and evaluating the effect of varying the template sizes on the tool's accuracy.

Perhaps more interesting is the potential for hybridization, i.e., joining DOM- and visual- heuristics in a single solution. Indeed, the visual search function can be improved by bringing in additional information that can help filter the multiple visual matches more intelligently. As an example, one can collect both DOM information and the method's call stack of the elements in order to verify the semantic equivalence of the elements between different versions.

For the interested reader, the source code and a demo video can be found on the tool's repository: https://github.com/saltlab/vista

REFERENCES

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

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

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

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

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

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

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

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

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

[10] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18)*. ACM.