**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:**


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.
2跑步的示例

我们考虑一个真实的回归测试场景，并解释DOM基的WEB测试修复器，优势将为一个视觉基于的方法。

图1展示了两个版本的AddressBook web应用，其中一个试验主题被用来在我们的实验研究[10]。我们考虑一个测试场景，在其中新出现的网页被添加到地址簿。在第一个版本6.2.12（图1a），测试②登录到应用（行1-3），点击“添加新”（行4）链接，并在新的用户名填入信息（行5-7）。

在新版本的AddressBook被发布后，一个测试人员可能会希望执行这个测试来检查是否在执行到新的网页。然而，当执行在版本7.0.0（图1b），测试②将停止因为多个断言是执行的。

非选择。首先，执行将停止在行3，当试图在“登录”按钮定位，因为其属性值已经被从HTML。在视觉查看的两个GUI，然而，一个测试者会希望测试来工作，因为其视觉性是不可见的，因为变化在DOM级别是相关的。因为，其实显而易见的是目标元素（例如，“登录”）在视觉上仍然是存在的，它的位置在GUI没有改变。

这是一个简单的直接断言，因为测试的场景是未改变的，没有bug是（最终）存在的在应用。然而，这个测试是不可行的因为同步化与应用是丢失的，并且一个固定需要被找到。

这个目标，一个测试者可能希望使用Web来自动地在行5处固定住断言。特别是，另一个定位者用于“登录”按钮③需要被生成，而不是在视觉上依赖于“断言”属性值。Web将试图获取关于断言的信息（例如，XPath，和各种属性）通过分析DOM的前一个版本6.2.12，并匹配这样的信息在已经修改的DOM版本7.0.0。不幸地，Web的技术是不成功的在这个情况，因为(i)属性值已经被从DOM，和(ii)都XPath和标记的属性改变（从输入到被删除），这是不可能的为Web的自动地修复。

断破工作流程。第二个非破坏性的断破发生在行5。当尝试在“名”文本字段定位，这个测试将引发一个NoSuchElementException实例。确实，一个新的中间确认页面被添加了（图1b），和导航工作流的测试必须被修正来反映新修改的WEB应用。

从一个测试者视角的，”名”文本字段不会再被找到在网页（测试状态）的执行后，根据测试的行4。然而，概念上，修复行动将需要被触发以修正测试已经没有任何东西去做的在行5。事实上，通过在视觉上检测在JUnit的断言，它在挑战了测试者来检测这个异常，不可能的，即使视觉执行的测试是被考虑。然而，即使是WEB是不成功的，因为工具将试图修复被在行5的一个新的断言（这个技巧只添加了声明在表单内，且不应用于一般的断破工作流程的）。这将试图修复被在行5的一个新的断言（这个技巧只添加了声明在表单内，且不应用于一般的断破工作流程的）。

误选择。最后，声明在行5-6将执行正确，但是声明在行7将填入这field“NickName”而不是field“Company”。在这个文献，这个被知道为一个非选择的问题[1, 10]。非选择的web元素可以得出不可预见的测试执行，这会从测试的意图行为，特别是依赖于一种行为被在行5进行的行为，当它达到一个点，一个行动不能被执行或一个元素不能被发现，但是实际的修复需要被在先前的测试声明（propagated breakage）。WEB不是被设计来检测非选择；然而那些场景是很难检测给一个测试者，因为只是在手动的视觉查看中他们是可以被识别的。断破的陈述。图1b⑤显示的是由Vista修复的，这将工作地正确地在AddressBook版本7.0.0。特别是在(i)非选择和非选择的被修正由更新标记的组件的测试声明（行3和5），和(ii)被破坏的Workflow是被修正通过添加一个新的测试声明到一个新的页面（因此创建了缺失的过渡）。在这一段的演示，我们展示我们的工具设计和实施。

3VISTA工具

我们发展VISTA背后的基本想法是有一个算法，通过考虑visual执行的测试可能需要被洞察到，WEB应用的 feasibility的DOM基的定位器通过它们的视觉显示，潜在地预见了断破的出现。另外，视觉的定位器也可能会被用在新修改的GUI来匹配定位器的。
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 \( r = 0.8 \). 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 \( \text{elementFromPoint}(x, y) \) 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 Crawjax [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 Conclusion 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


