ERRATUM: *Leveraging Flexible Tree Matching to Repair Broken Locators in Web Automation Scripts*

Sacha BRISSET, *Romain* ROUVOY, *Lionel* SEINTURIER, *Renaud* PAWLAK

Abstract

Web applications are constantly evolving to integrate new features and fix reported bugs. Even an imperceptible change can sometimes entail significant modifications of the *Document Object Model* (DOM), which is the underlying model used by browsers to render all the elements included in a web application. Scripts that interact with web applications (*e.g.* web test scripts, crawlers, or robotic process automation) rely on this continuously evolving DOM which means they are often particularly fragile. More precisely, the major cause of breakages observed in automation scripts are *element locators*, which are identifiers used by automation scripts to navigate across the DOM. When the DOM evolves, these identifiers tend to break, thus causing the related scripts to no longer locate the intended target elements.

For this reason, several contributions explored the idea of automatically repairing broken locators on a page. These works attempt to repair a given broken locator by scanning all elements in the new DOM to find the most similar one. Unfortunately, this approach fails to scale when the complexity of web pages grows, leading either to long computation times or incorrect element repairs. This article, therefore, adopts a different perspective on this problem by introducing a new locator repair solution that leverages tree matching algorithms to relocate broken locators. This solution, named ERRATUM, implements a holistic approach to reduce the element search space, which greatly eases the locator repair task and drastically improves repair accuracy. We compare the robustness of ERRATUM on a large-scale benchmark composed of realistic and synthetic mutations applied to popular web applications currently deployed in production. Our empirical results demonstrate that ERRATUM outperforms the accuracy of WATER, a state-of-the-art solution, by 67%.

Keywords: Element locator, Locator repair, Robotic process automation, Tree matching, Web crawling, Web testing, Locator Repair, DOM-based Locators

1. Introduction

The implementation of automated tasks on web applications (apps), like crawling or testing, often requires software engineers to locate specific elements in the DOM (*Document Object Model*) of a web page. To do so, software engineers or automation/testing tools often rely on CSS (*Cascading Style Sheets*) or XPath selectors to query the target elements they need to interact with. Unfortunately, such statically-defined locators tend to break along time and deployments of new versions of a web application. This often results in the failure of all the associated automation scripts (including test cases) that apply to the modified web pages.

While several existing works focus on repairing tests on GUI applications, there are surprisingly very few test repair solutions targeting web interfaces [1]. These solutions either propose to *i)* generate locators that are robust to changes (so-called *robust locator problem*), or *ii)* repair locators that are broken by the changes applied to the web pages (so-called *locator repair problem*). Unfortunately, most of the existing solutions in the literature fail to accurately relocate a broken locator, thus leaving all the related web automation scripts as broken [2]. More specifically, state-of-the-art solutions to the locator repair problem, WATER [3] and VISTA [4], tend to rely on the intrinsic properties of the element whose locator needs repairing to locate its matching element on the modified page. However, this approach fails to leverage the element position and relations with the rest of the DOM, thus ignoring valuable contextual insights that may greatly help to repair the locator.

In this article, we adopt a more holistic approach to the locator repair problem: instead of focusing on the element whose locator is broken individually, we leverage a tree matching algorithm to match all elements between the two DOM versions. Intuitively, using a holistic approach to repair a broken locator should significantly improve accuracy by reducing the search space of candidate elements in the new version of the page: for example, if the parent of the element whose locator is broken is easily identifiable (*e.g.*, the item of a menu) a tree matching algorithm will use this information to relocate the target element in the modified page with better accuracy. Additionally, if more than one locator is broken on a given web page, our approach will repair all of them consistently at once. The holistic solution we propose, named ERRATUM,[1] more specifically leverages an efficient *Similarity-based Flexible Tree Matching* (SFTM) algorithm we implemented to repair all broken locators by matching all changes in a web page with high accuracy. SFTM is at the heart of ERRATUM's approach. It is a tree matching algorithm specialized in matching web pages providing fast computation times and high accuracy when compared to other generic tree matching solutions. To do so, SFTM makes use of a distinctive characteristic of DOM trees: the labels of the nodes (i.e. node attributes and tags) contain a high amount of information that can be leveraged to prune the space of possible matchings between two trees.

Evaluating solutions to both robust locator and locator repair problems requires to build a dataset of web page versions—*i.e.*, (original page, modified page) pairs. Unfortunately, previous works assessed their contributions on hardly-reproducible benchmarks of limited sizes (never beyond a dozen of websites). In this article, we rather evaluate the robustness of our approach against the state of the art by introducing an open benchmark, which covers a wider range of changes that can be found in modern web apps. Concretely, our open benchmark considers over 83k+ locators on more than 650 web apps. It combines *i)* a *synthetic dataset* generated from random mutations applied to popular web apps and *ii)* a *realistic dataset* replaying real mutations observed in web apps from the Alexa Top 1K[2], which ranks the most popular websites worldwide.

When evaluated on both datasets, our results demonstrate that ERRATUM outperforms the state-of-the-art solution, namely WATER [3], both in accuracy (67% improvement on average) and performances, when more than 3 locators require to be repaired in a web page.

Concerning the potential applications of ERRATUM, while we introduce and evaluate our solution within the well-studied context of locator repair, we also discuss a novel testing architecture centered around ERRATUM allowing to entirely replace all locator-based interactions. This architecture intends to support much more interactive and robust script editions in the context of web testing, web crawling, and robotic process automation.

[1]ERRATUM stands for "*rEpaiRing bRoken locATors Using tree Matching*"

[2]`https://www.alexa.com/topsites`

Overall, the key contributions of this article consist of:

1. proposing a solution to the locator repair problem inspired from the *Flexible Tree Matching* (FTM) algorithm,
2. implementing and integrating an efficient extension of FTM algorithm to repair broken locators,
3. providing a novel, reproducible, large-scale benchmark dataset to evaluate both the robust locator and locator repair problems,
4. reporting on an empirical evaluation of our approach when solving the locator repair problem,
5. proposing a novel script edition architecture centered on ERRATUM.

The remainder of this article is organized as follows. Section 2 introduces the state-of-the-art approaches in the domain of robust locators and locator repair, before highlighting their shortcomings. Section 3 formalizes the locator problem we address in this article. Section 4 introduces our approach, ERRATUM, which leverages an efficient flexible tree matching solution that we identified. Section 5 describes the locator benchmark we designed and implemented. Section 6 reports on the performance of our approach compared to the state-of-the-art algorithms. Finally, Section 7 presents some perspectives for this work, while Section 8 concludes.

2. Background & Related Work

We deliver a novel contribution to the locator repair problem, which has been initially studied in the domain of web testing. In this section, we thus introduce the required background and we describe state-of-the-art approaches to repair broken locator, and in particular the literature published in the domain of web test repair.

2.1. Introducing Web Element Locators

To detect regressions in web applications, software engineers often rely on automated web-testing solutions to make sure that end-to-end user scenarios keep exhibiting the same behavior along changes applied to the system under test. Such automated tests usually trigger interactions as sequences of actions applied on selected elements and followed by assertions on the updated state of the web page. For example, *"click on button e_1, and assert that the text block e_2 contains the text '*`Form sent`*'"*. To develop such test scenarios, a software engineer can 1. manually write web test scripts to interact with the application, or 2. use record/replay tools [5, 6, 7] to visually record their scenarios. In both cases, the scenario requires to identify the target elements on the page [e_1, e_2] in a deterministic way, which is usually achieved using XPath, a query language for selecting elements from an XML document. For example, let us consider the following HTML snippet describing a form:

```
<form method="post" action="index.php">
    <input type="text"   name="username"/>
    <input type="submit" value="send"/>
</form>
```

The following XPath snippets describe 3 different queries, which all result in selecting the submit button: `/form/input[2]`, `/form/input[@value="Send"]`, `input[@type="submit"]`. In the literature, such element queries or identifiers are named *locators* [8].

In practice, automated tests are often subject to breakages [2]. It is important to understand that a test *breakage* is different from a test *failure* [4]: a test failure successfully exposes a regression of the application, while a test is said to be broken whenever it can no longer apply to the application (*e.g.*, the test triggered a click on a button e, but e has been removed from the page). While there can be many causes to test breakage, [2] reports that 74 % of web tests break because one of the included locators fails to locate an element in a web page.

2.2. Generating Web Element Locators

The fragility of locators remains the root cause of test breakage, no matter they have been automatically generated (*e.g.*, in the case of record/replay tools), or manually written. To tackle this limitation, several studies have focused on generating more robust locators. This includes ROBULA [8], ROBULA+ [9], which are algorithms that apply successive refining transformations from a raw XPath query until it yields a locator that exclusively returns the desired element. Leotta *et al.* [10] also propose a graph-based algorithm to generate locators, but has not provided any evaluation to the algorithm. Another work by Yandrapally *et al.* leveraged contextual clues to generate locators [11]. These clues rely mostly on the content surrounding the element to locate which may be problematic in case the content changes. LED [12] uses a SAT solver to select several elements at once, but is never evaluated on different DOM versions. Finally, some works combine several locator generators with a voting mechanism to locate a single element with more robustness [13, 14, 15]. However, all these approaches, which consider a limited set of locator generators, strongly depend on the accuracy of individual algorithms to agree upon a single and relevant locator. While automatically generating locators can speed up the definition of test cases, it becomes a keystone for visually-generated test cases based on record/replay tools. In the end, the reliability of test cases built using such a tool depends mostly on the quality of the locators it automatically generates [2].

2.3. Repairing Web Element Locators

While some solutions to the robust locator problem, as presented above, aim to prevent locators from breaking, others focus on repairing broken locators. In this context, the repair tool considers *a)* the descriptor of the locator, *b)* the last version of the page on which the locator was still functional (D), *c)* the new version of the page on which the locator is broken (D').

WATER. In this area, WATER [3] provides an algorithm to fix broken tests. The process of repairing a test involves several steps: 1. running the test, 2. extracting the causes of failure and, 3. repairing the locator, if broken. The last part is particularly challenging. To relocate a locator from one version to another, WATER scans all elements in the new version and returns the most similar one to the element in the original version with regards to intrinsic properties (*e.g.*, absolute XPath, classes, tag). Hammoudi *et al.* [16] further studied the locator repair part of WATER and found that repairing tests over finer-grained sequences of change (typically commits) contributes to improving accuracy.

VISTA. Using a completely different approach, VISTA [4] is a recent technique that adopts computer vision to repair locators. VISTA falls within the category of computer vision-aided web tests [17, 18, 19]. However, while using computer vision succeeds in repairing most of the *invisible* changes, such solutions tend to fail when the content, the language, or the visual rendering of the website changes. Furthermore, visual-based solutions fail to locate dynamic elements that only appear through user interactions (*e.g.*, a dropdown menu).

3. Locator Problem Statement

Figure 1 summarizes the steps to follow when writing or repairing a locator in a web test script. When a test breaks, the repairing process generally includes three main steps: 1. extract the cause of the breakage 2. if a locator caused the breakage, the element is first relocated then 3. a new locator

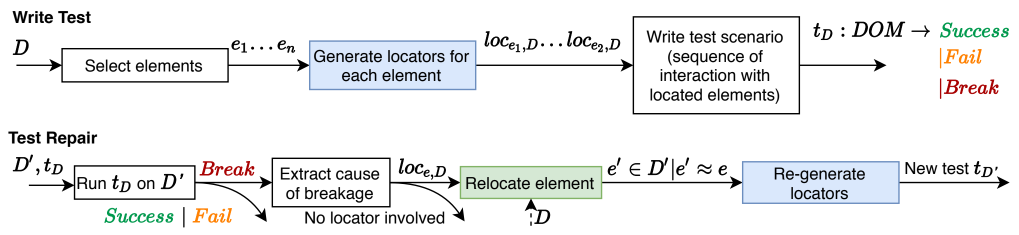

Write Test
D
Select elements
$e_1 \dots e_n$
Generate locators for each element
$loc_{e_1,D} \dots loc_{e_2,D}$
Write test scenario (sequence of interaction with located elements)
$t_D : DOM \rightarrow Success$
$|Fail$
$|Break$
Test Repair
D', t_D
Run t_D on D'
Break
Success | Fail
Extract cause of breakage
$loc_{e,D}$
No locator involved
Relocate element
D
$e' \in D' | e' \approx e$
Re-generate locators
New test $t_{D'}$

Figure 1: Illustration of the locator problem statement in automated tests combining the *robust locator* (in blue) and the *locator repair* (in green) problems.

is generated/written. Beyond automated tests, this problem can also arise in more general web automation scripts covering web content crawling and *Robotic Process Automation* (RPA), which heavily rely on locators to automate the navigation across web applications.

In this section, we formalize the description of two locator-related problems highlighted in Figure 1, namely the *robust locator* (in blue) and *locator repair* (in green) problems for the general case of web automation scripts.

3.1. Problem Notations

We consider that a given web page can change for various reasons, such as 1. content variation, 2. page rendered for different regions/languages, or 3. release of the web application. No matter the cause, we distinguish D and D' as two versions of the same web page observed before and after a change, respectively. More specifically, we define the following similarity notations:

1. $D \approx D'$ if scripts written for D are expected to apply on D';
2. Given 2 web elements $e \in D$ and $e' \in D'$, $e \approx e'$ if e and e' refer to semantically equivalent elements (*e.g.*, the same menu item observed in pages D and D');
3. By extension of (2), given a set of elements $E = e_1 ... e_n \subset D$ and $E' = e'_1 ... e'_{n'} \subset D'$, $E \approx E'$ if $n = n'$ and, for each $i \in [1..n]$, $e_i \approx e'_i$.

Based on the above similarity notation, we provide the following definitions:

Definition 3.1. Given a page D, and a set of elements $E = e_1 ... e_n$, the pair $(loc_{E,D}, eval)$ is a **locator** of E with regard to D if:

$$eval(loc_{E,D}, D) = E \qquad (1)$$

where $loc_{E,D}$ is a descriptor of E and $eval$ an evaluation function that returns a set of web elements from a descriptor and an evaluation context (*e.g.*, a web page).

In the case of XPath-based locators, the descriptor $loc_{E,D}$ refers to an XPath query describing the elements E in the page D and $eval$ the XPath solver.

Definition 3.2. Let mut be a mutation function that transforms the page D into another page D', such as $mut(D) = D'$. mut is said to be a **mutation** of D if $D \approx D'$.

Definition 3.3. Given a locator $L = (loc_{E,D}, eval)$, L is **robust** to a mutation function mut if:

$$eval(loc_{E,D}, mut(D)) \approx E \tag{2}$$

Finally, we note $\lambda(e)$ the label of the node e in the DOM tree. The label of a node comprises the tag, the attributes and their values and the textual content. However, in the context of Erratum, we willingly ignore the content as described in section 4.3.

3.2. Problem Statement

Given the above definitions, we can formalize the locator problem statement along with the two following research questions.

RQ 3.1. Robust Locator. For any subset of elements on a given page D, how to automatically generate locators that are robust to mutations of D?

When evaluating a locator on a new page D', the only available information to describe the targeted element is the descriptor $loc_{E,D}$, which often remains insufficient (cf. state-of-the-art techniques).

On the other hand, in the context of *locator repair*, the original page D from which $loc_{E,D}$ was built is available. Thus, using definition 3.1, this piece of information allows to locate the originally selected elements $eval(loc_{E,D}, D) = E$.

RQ 3.2. Locator Repair. Given two pages D, D', such that $D \approx D'$ and a set of elements $E \in D$, how to locate the elements $E' \approx E$ in D'?

To the best of our knowledge, existing solutions to both *robust locator* and *locator repair* focus on the restricted case of $|E| = 1$.

Once the *locator repair problem* is solved (*i.e.*, E' are correctly located), we need to generate new locators, which brings us back to the situation of the robust locator problem (cf. RQ. 3.1).

In this article, we thus present a novel approach to solve the locator repair problem.

4. Repairing Locators with Erratum

The previous section formalized both *robust locator* and *locator repair* problems. The approach we report in this article, Erratum, therefore matches the DOM trees of 2 versions of a web page to solve the *locator repair* problem. Several tree matching solutions exist in the literature, such as *Tree Edit Distance* (TED) [20] or tree alignment [21]. This section therefore motivates and explains how Erratum leverages tree matching to repair locators, before discussing the choice of a tree matching implementation fitting Erratum's requirements.

4.1. Applying Tree Matching to Locator Repair

Embedding tree matching allows Erratum to leverage the tree structure in the same way an XPath-based solution would, while offering the flexibility of a more statistic-based solution. Intuitively, a tree matching algorithm should consider all easily identifiable elements on a page (elements with rare tags, unique classes, ids, or other attributes) as *anchors* to relocate less easily identifiable elements.

Figure 2 illustrates the benefits of a more holistic approach using tree matching. In the example, the locator of element `a` (in blue) breaks because the mutations between D and D' entails a change in its absolute XPath (`/body/div/a`). Attempting to repair such a broken locator by relying on the properties of the original element alone (state-of-the-art approaches like [3, 4]) is often challenging and can easily lead to a mismatch. By using tree matching (cf. right-bottom of Figure 2), matching the parent of the element to locate (`div#menu`) brings instead a strong contextual clue to accurately relocate the element $\texttt{a}_1$' whose locator was broken [22].

Formally, given a pair of page versions D and D', we:

1. parse D and D' into DOM trees T and T'. Consequently, *nodes*(T) is the set of elements in the DOM tree T;

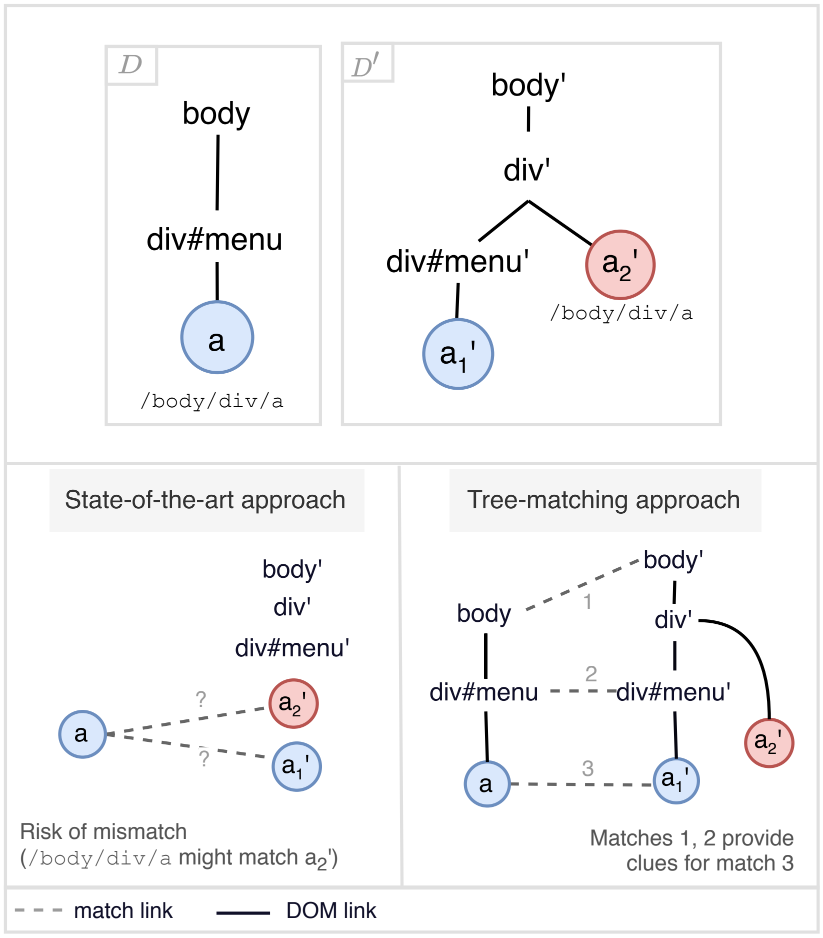


Figure 2: State-of-the-art Vs. tree matching locator repair.

2. apply tree matching to T, T' yielding a matching $M \subset nodes(T) \times nodes(T')$. If the resulting matching M is accurate, then $\forall (e, e') \in M, e \approx e'$;
3. use the resulting matching M to repair the broken locator(s).

Regarding the test repair process illustrated in Figure 1, our approach thus fits in the block "relocate element" (in green) by matching the elements of D in D' and reporting the relocated element. Thus, once the element is relocated using tree matching—*i.e.* ERRATUM found $e' \in D' | e \approx e'$—we only need to generate a new locator $loc_{e',D'}$ to achieve the test repair process. This task can be performed using solutions to the robust locator problem, like ROBULA [8], and is therefore considered as out of the scope of this article.

4.2. Integrating a Scalable Tree Matching Algorithm

The state-of-the-art approach to match two trees is *Tree Edit Distance* (TED) [20]. When comparing two trees T and T', TED-based approaches rely on finding the optimal sequence of relabels, insertions and deletions that transforms T into T'. Unfortunately, TED might be unsuitable to match real-life web pages due to two core restrictions [23]: 1. if two nodes e and e' are matched, the descendants of e can only match with the descendants of e', and 2. the order of siblings must be preserved. Furthermore, TED is computationally expensive ($O(n^3)$ for the worst-case complexity [24]) and, more practically, our preliminary experimentation has shown that applying the state-of-the-art implementation of TED, named APTED [25], on the *YouTube* page takes several minutes. We believe that, in addition to qualitative restrictions, such computation times are not acceptable when periodically repairing locators on real websites.

Further studies of TED proposed to improve computation times [26, 27, 28], but at the cost of even more restrictive constraints on the produced matching (*e.g.*, the tree alignment problem [26] restricts the problem to transformations where insertions are performed before deletions).

To the best of our knowledge, the only contribution that provides a solution to the general (restriction-free) tree-matching problem is the *Flexible Tree Matching* (FTM) algorithm [23]. FTM models tree matching as an optimization problem: given two trees T and T' how to build a set of pairs $(e, e') \in T \times T'$ such that the similarity between all selected node pairs is maximal. The similarity used by FTM combines both the labels and the topology of the tree.

However, as shown in [22], the theoretical complexity of FTM is high ($O(n^4)$) and the implementation of FTM was shown to take more than an hour to match a web page made of only 58 nodes, while the average number of nodes on a web page observed in our dataset is $1,507$. Consequently, we believe that such computation times make FTM unpractical in the context of locator repair.

4.3. Matching DOM Trees by Similarity

Given the limitation of FTM, Erratum implements a *Similarity-based Flexible Tree Matching* (SFTM) algorithm, which is an extension of state-of-the-art FTM to improve the computation times of FTM without any restriction on the resulting matching.

In the context of Erratum, the SFTM algorithm assumes that, given a web page, several elements are easily identifiable by considering their intrinsic properties. The algorithm first assigns scores to all possible matches between nodes from the two trees based on their label and only then uses the topology of the trees to adjust these scores.

Almost all existing tree matching algorithms rely first and foremost on the topology of the trees. Conversely, SFTM relies mostly on the labels of the trees and only makes use of topology in a second step, to fine-tune the already computed scores. Intuitively, matching two sets of labels is significantly easier than trying to match trees, which is the reason why SFTM achieves such competitive performances. The only trade-off of this approach is that it requires the labels of the trees to be highly differentiating (i.e. carry a lot of information). Fortunately, this is the case for the great majority of web pages.

We walk through the key steps of the SFTM algorithm we implemented in Erratum by using HTML snippets reported in Figure 3. The figure provides two versions (D and D') of a simplified HTML code sample extracted from the homepage of the famous search engine *duckduckgo.com.* In this example, our purpose is to relocate $\mathrm{a}_1 \in D$ with $\mathrm{a}_1' \in D'$.

Unlike the state-of-the-art matching algorithms, SFTM first tries to match elements in D' whose labels are similar to D, before using these matched elements to adjust the similarity of surrounding elements in the tree. For example, the *similarity scores* of the tuple (a_1, a_1') links will increase as their direct parents (div_3, div_3') are matched with confidence. Figure 4 summarizes the SFTM algorithm's key steps and the remainder of this section provides an overview of its integration in Erratum (cf. green box in Figure 1).

(a) Original document *D*.

```
<div class="content-info__item">  div_1
    <div class="item__title">...</div>  div_2
    <div class="item__subtitle">  div_3
        ...
        <a href="/plugins">Plugins</a>  a_1
    </div>
</div>
```

(b) Updated document D' from *D*.

```
<div class="items-wrap">  div'_4
    <div class="item">  div'_1
    <div class="item__title">...</div>  div'_2
        <div class="item__subtitle">  div'_3
            ...
            <a href="/extensions">Extensions</a>  a'_1
        </div>
    </div>
    <div>  div'_5
        ...
        <a href="/newsletter">Newsletter</a>  a'_2
    </div>
</div>
```

Figure 3: Two versions of an HTML snippet extracted from the homepage of *duckduckgo.com*.

The interested reader can refer to our technical report [22] for an exhaustive description of our SFTM algorithm.

Step 1: Node Similarity. Elements of DOMs D and D' are compared. The first step consists in computing an *initial similarity* $s_0 : D \times D' \to [0,1]$. For each pair of nodes $(e, e') \in D \times D'$, $s_0(e, e')$ measures how similar the labels of e and e' are. In HTML pages, the label of a node $e \in D$ that we use is the set of tokens obtained from applying a tokenizer to the HTML code describing e. This may include the type of the HTML element, its attributes, and eventually the raw content associated to this element—*i.e.*, thus ignoring the content from the child elements.

Example 4.1. The *label* computed for div_1 (cf. Figure 3) includes the following tokens: $\{$`div`, `class`, `content-info__item`$\}$.

To compute s_0, SFTM first indexes the labels of each node of D. The idea of this step is to prune the space of possible matches by pre-matching nodes with similar labels. When indexing, to improve the accuracy of s_0, we apply the *Term Frequency-Inverse Document Frequency* (TF-IDF) [29] formula to take into account how rare each token is.

Example 4.2. Following our previous Example 3, when considering the match (div_1, div'_1):

1. token `div` will yield very few similarity points, since it is included in the labels of almost all the nodes,
2. token `content-info__item` will increase the score significantly, as it only appears once in both documents.

In general, very common tokens bring very few information to the relevance of a given match, while they cause a significant increase of potential matches to consider. That is why, in order to reduce the computation times, the algorithm rules out most common tokens.

Step 2: Similarity Propagation. The initial similarity s_0 only takes into account the labels of nodes. In this second step, the idea is to enrich the information contained in s_0 by leveraging the topology of the trees D and D'.

Example 4.3. In Example 3, it is hard to choose the correct match $m_1 = (a_1, a'_1)$ over the incorrect one $m_2 = (a_1, a'_2)$ by only considering labels, since all three elements share the same set of tokens: $\{$`a`, `href`$\}$. In the similarity propagation step, we leverage the fact that the parents of a_1 and a'_1 are

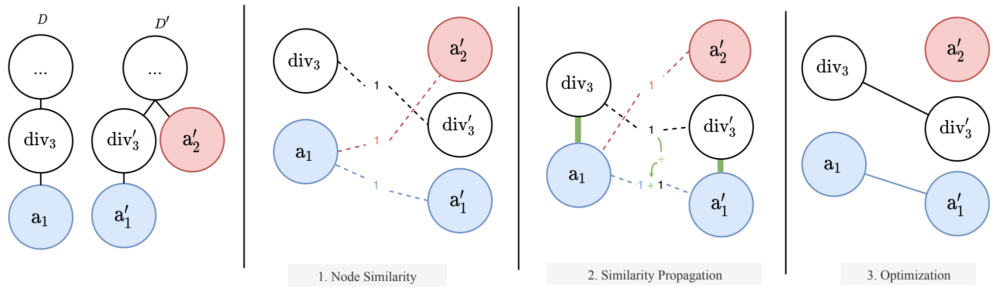


Figure 4: Key steps followed by our *Similarity-based Flexible Tree Matching* (SFTM) algorithm.

similar to increase the similarity between a_1 and a'_1, thus preferring m_1 over m_2.

In general, for each considered match $(e, e') \in D \times D'$, the parents of e and e' gets more similarity points if e and e' are similar and inversely, $s(e, e')$ is increased if the parents of e and e' are similar (with regard to s_0). We call s the final similarity produced by this step.

Step 3: Optimization. Producing the optimal matching with regards to the computed similarity means selecting the full set of matches such that each element of D is matched with at most one element of D' and the sum of similarity scores of the selected matches is maximum.

In order to approximate the optimal set of matches, SFTM implements the Metropolis algorithm [30]. The idea is to randomly walk through several possible configurations (set of matches) to converge towards the optimal one.

At the end of the optimization step, the SFTM algorithm yields a full matching $M \subset D \times D'$ comprising matches between nodes of D and D'. These matches can be analyzed by ERRATUM to locate broken locators and fix them by generating new locators in the target document D'.

Best and worst case scenarios. The greatest strength of ERRATUM's approach based on SFTM is to handle variations of label e.g. renaming a class, adding an id, removing an attribute. This is because of the fuzzy nature of the first similarity steps of SFTM: as long as two elements' label share enough rare tokens, they will match. For elements that have few information in their label (e.g. a bare *div* tag with no attributes) the algorithm will still be able to rely on the parents and children of the node whose label may contain more discriminative information.

The worst case scenario happens when making structural changes around nodes with with labels containing few information. For example, if the content of a bare *div* node is moved to another *div* node. matching the first *div* node accurately will become very challenging. More than that, since the node that was moved contains high amount of information, the matching of this node will propagate to its new parents thus increasing the chances of mismatch in the surrounding of the moved node.

5. The Robust Locator Benchmark

In the context of this article, we are interested in covering the following research questions:

RQ 5.1. How does ERRATUM perform in solving the locator repair problem (cf. RQ. 3.2) when compared to state-of-the-art solutions?

RQ 5.2. What are the factors influencing the accuracy of WATER and ERRATUM?

RQ 5.3. How quickly can ERRATUM repair broken locators when compared to state-of-the-art solutions?

This section, therefore, describes the benchmark we propose to assess these questions.

5.1. Evaluated Locator Repair Solutions

We compare two solutions: 1. ERRATUM, our approach to repair broken locators by leveraging flexible tree matching, and 2. WATER, the reference implementation of a locator repair technique applied to web test scripts [3].

The original algorithm of WATER analyses a given test case, finds the origin of the test breakage, and suggests potential repairs to the developer. In the context of this article, we are interested in the most challenging part of the algorithm: the part that repairs broken locators, if needed. Given the originally located element $e \in D$, WATER attempts to find $e' \in D'$ such that $e' \approx e$ by scanning over all elements in D' such that $tag(e') = tag(e)$ and selecting the elements most similar to e. The similarity between two elements e_1, e_2 used by WATER mostly consists in computing the Levenshtein distance between the absolute XPaths of both elements ($Levenshtein(XPath(e_1), XPath(e_2))$) combined with other element properties similarity (*e.g.*, visibility, z-index, coordinates). In our evaluation, we re-implemented this part of the WATER algorithm to compare its performance to ERRATUM.

We initially considered VISTA [4] as a baseline, even though the approach they use (computer vision) is radically different from ERRATUM and WATER. However, despite our efforts, we failed to run their implementation and received no answer when trying to contact the authors.

Note that, in this evaluation, we focus on *single-element locator* cases of the locator repair problem (we only try to repair *single-element locators*). The reasons for this decision are: 1. The state-of-the-art solutions to both repair and robust locator problems only treat this case and in particular, WATER can only repair locators locating a single element, 2. ERRATUM reasons on the whole trees, so locating several independent elements is done the same way as locating a group of elements.

5.2. Versioned Web Pages Datasets

In the remainder of this article, we propose two datasets to compare ERRATUM and WATER against potential evolutions of web pages. Given two versions of the same page D, D', and a set of elements $E \subset D$, the locator repair problem consists in locating a set of elements $E' \subset D'$, such that $E' \approx E$. To evaluate the performance of a locator repair tool, we thus need what we call a **DOM versions dataset**: a dataset of pairs (D, D'), such that $D \approx D'$.

A DOM version dataset is also required to evaluate solutions to the robust locator problem. To build such a dataset, previous works on locator repair [9, 8] and robust locator [4, 3, 16] manually analyzed different versions of a few open source applications (like Claroline, AddressBook or Joomla). These evaluations are significantly limited in size (never beyond a dozen of websites considered) and hard to reproduce since the exact versions of the open source applications used are often not provided or available.

In our study, we therefore introduce the first large-scale, reproducible, real-life *DOM versions dataset* that can be used to assess locator repair solutions, and is composed of two parts:

1. A **MUTATION dataset** [22] generated by applying random mutations to a given set of web pages (see Section 5.2.1),
2. A **WAYBACK dataset** collects past versions of popular websites from the Wayback API (see Section 5.2.2).

Then, for each pair (D, D') in the dataset, our experiments consist of selecting a set of elements to locate in D and in comparing both ERRATUM and WATER trying to find the corresponding element on D'.

Table 1 describes both datasets in terms of:

1. # **Unique URLs**: the number of unique URLs among the total of version pairs in the dataset. The duplication is due to the fact that there can be several mutations or successive versions of the same web page. In the case of the WAYBACK dataset, more popular websites are more represented (see Section 5.2.2);
2. # **Version pairs**: the number of considered pairs of web pages (D, D'),
3. # **Located elements**: the number of elements $e \in D$ that any solution should locate in D'.

Table 1: Description of the MUTATION & WAYBACK datasets.

Dataset	MUTATION	WAYBACK
# Unique URLs	650	64
# Version pairs	3,291	2,314
# Located elements	49,305	34,421

The two datasets we provide are complementary. Since the MUTATION dataset is generated by mutating elements from an original DOM D, the ground truth matching between D and its associated mutation D' is known to easily evaluate the solution on a very large amount of version pairs. However, since the versions are artificially generated, this dataset is synthetic and, as such, might not entirely reflect the actual distribution of mutations happening along a real-life website lifecycle.

Then, the WAYBACK dataset is composed of real website versions mined from the Wayback API: an open archive that crawls the web and saves snapshots of as many websites as possible at a rate depending on the popularity of the website.[3] In the WAYBACK dataset, mutations between D and D' are not synthetic, but as a result, the ground truth matching between D and D' is unknown. In our evaluation, we thus had to manually label a sample of the results obtained on this dataset, which limits the scalability of the experiment compared to the MUTATION dataset.

The following sections provide more details on how both datasets were built.

[3]`https://archive.org/help/wayback_api.php`

Table 2: Mutations applied in the MUTATION dataset [22].

Type	**Mutation operators**
Structure	`remove, duplicate, wrap, unwrap, swap`
Attribute	`remove, remove words`
Content	`replace with random text, change letters, remove, remove words`

5.2.1. Building the MUTATION *dataset.*

We extend the technique we introduced to generate a MUTATION dataset in [22]. The mutation dataset is built by applying a random amount of random mutations to a set of original webpages: for each original DOM D, 10 mutants are created by applying mutations to D. Since the mutations applied to D to construct each mutant D' are known, the ground truth matching between D and D' is also known. Knowing the ground truth matching on the mutation dataset allows us to evaluate our locator repair solution on a very large dataset. Table 2 describes the set of DOM mutations that can be observed along evolution of a web page.

The original websites from which mutants were generated were randomly selected from the Top 1K Alexa. Figure 6 depicts the distribution of DOM sizes in this synthetic dataset.

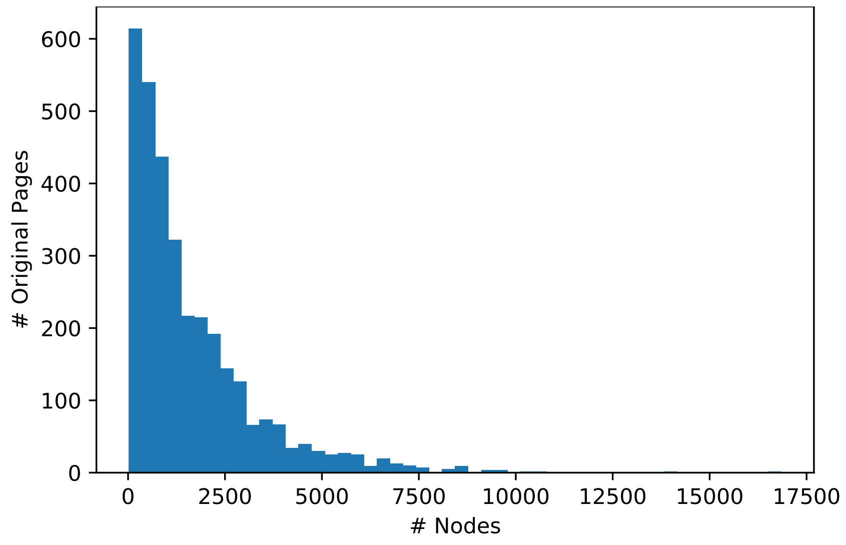


Figure 5: Distribution of DOM sizes (in number of nodes) in the MUTATION dataset.

This dataset was built with an automation tool that we made available along with its source code 8. From a given list or source URLs, our tool creates a dataset of randomly mutated webpages following the above-described approach.

5.2.2. Buidling the WAYBACK *dataset.*

This dataset encloses a list of (D, D') DOM pairs where D and D' are two versions of the same page (*e.g.*, *google.com* between 01/01/2013 and 01/02/2013). Two versions can be separated by different gaps in time. In this section, we explain how we used the Wayback API to build this dataset. The Wayback API can be used to explore past versions of websites. The two endpoints we used to build the dataset can be modeled as the following functions:

$$\begin{aligned} versionsExplorer :: &\quad (url, duration) \rightarrow &\quad timestamp[] \\ versionResolver :: &\quad (url, timestamp) \rightarrow &\quad document \end{aligned}$$

The *versionsExplorer* retrieves the list of available snapshots between two dates, while the *versionResolver* returns the snapshot of a given *url* at the requested timestamp.

Using these endpoints, for each website URL considered, we:

1. retrieved the timestamps of all versions between 2010 and today using the *VersionExplorer*,
2. generated a list of all pairs of timestamps with one of following differences in days (±10%): [7, 15, 30, 60, 120, 240, 360],
3. picked up to $1,000$ random elements from the list of timestamps pairs,
4. resolved each selected timestamp pair using the *versionResolver*.

Similarly to the MUTATION dataset, the URLs we fed to our algorithm were taken from the Top 1K Alexa. Since both datasets are based on the same set of URLs (taken from Alexa), the distribution of the WAYBACK dataset is very similar to the MUTATION one (cf. Figure **??**).

5.2.3. Selecting the elements to repair.

ERRATUM and WATER operate in different ways. ERRATUM takes two trees (D, D') and returns a matching between each element of the trees, thus solving any possible broken locator between D and D'. The algorithm extracted from WATER is a more straightforward solution to the locator repair problem as formally described (cf. Section 3.2): it takes a pair of

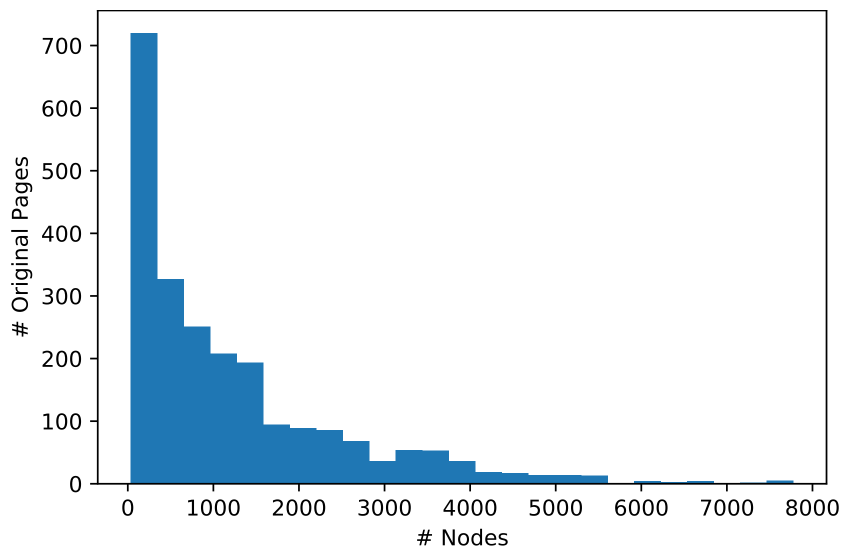
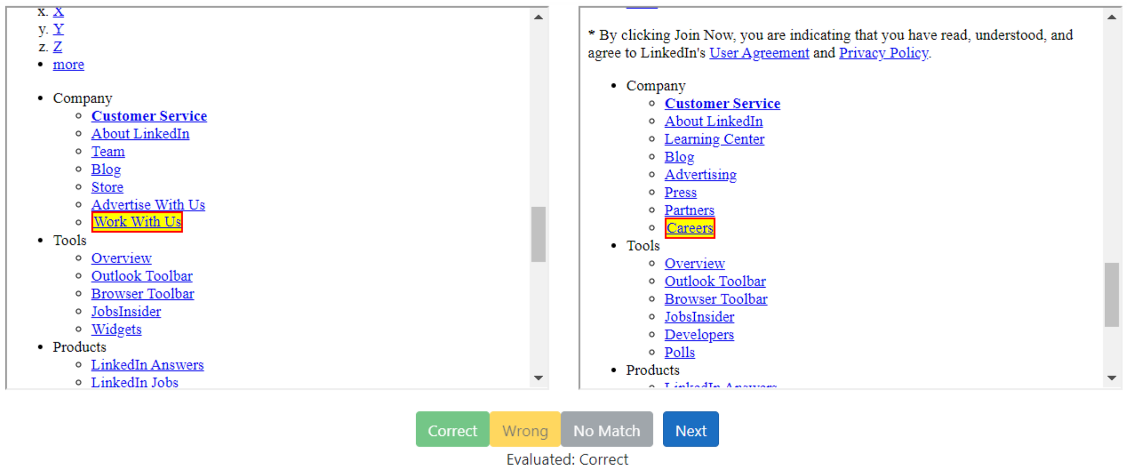


Figure 6: Distribution of DOM sizes (in number of nodes) in the Wayback dataset.

DOM versions (D, D') and an element $e \in D$ as input and returns an element $e' \in D'$ (or *null* if it fails to find any candidate for the matching).

Consequently, in the case of the WATER algorithm, the following question arises: given a pair (D, D') taken from the DOM version dataset, which elements of D should be picked for repair? Ideally, we would try to locate every element of D in D' to obtain a comprehensive comparison with Erratum. Unfortunately, the computation times of WATER make it impractical to locate every single element from D in D'. Selecting realistic targets for locators is a non-obvious task since many elements in the DOM would not be targeted in a test script (*e.g.*, large container blocks, invisible elements, aesthetic elements). Therefore, for each version pair, we randomly select up to 15 clickable elements from D. We focus on clickable elements as this is the most common use case for web UI testing (to trigger interactions), and WATER has specific heuristics to enhance its accuracy on links. By considering clickable elements, we 1. make sure to choose realistic elements, and 2. compare to WATER on its most typical use case.

Regarding the sample size, considering 15 elements per web page leads to selecting 34,000+ elements in both datasets. As the average number of nodes per web page in each dataset is around $1,500$, this means that there are more than 3.6M candidate locators for repair in each dataset. Therefore, the confidence interval at 95 % of the measurements applied to the 34K sample


x. X
y. Y
z. Z
- more

- Company
  - **Customer Service**
  - About LinkedIn
  - Team
  - Blog
  - Store
  - Advertise With Us
  - Work With Us
- Tools
  - Overview
  - Outlook Toolbar
  - Browser Toolbar
  - JobsInsider
  - Widgets
- Products
  - LinkedIn Answers
  - LinkedIn Jobs

* By clicking Join Now, you are indicating that you have read, understood, and agree to LinkedIn's User Agreement and Privacy Policy.

- Company
  - **Customer Service**
  - About LinkedIn
  - Learning Center
  - Blog
  - Advertising
  - Press
  - Partners
  - Careers
- Tools
  - Overview
  - Outlook Toolbar
  - Browser Toolbar
  - JobsInsider
  - Developers
  - Polls
- Products

Evaluated: Correct

Figure 7: Labeling a given element matched by ERRATUM on two versions of the Linkedin homepage. The screenshot comes from the visual matching application we created to manually label disagreements between ERRATUM & WATER.

of located elements is 0.5 %.

5.3. Evaluating of the Matched Elements

On the MUTATION dataset, the signatures attributes are preserved after mutations (but ignored when applying either locator repair solution), thus providing the ground truth matching between the DOMs of a version pair.

For the WAYBACK dataset though, this information is not available. For each version pair (D, D'), the evaluation of both solutions yields to a list of suggested matching $(e, e'_{ERRATUM})$ and (e, e'_{WATER}) where $e \in D$ and $e'_{ERRATUM}, e'_{WATER} \in D'$. In both cases, e' may be null in case no matching was found. Given the above situation, the labeling process consists of determining whether the matching element of e is $e'_{ERRATUM}$, e'_{WATER}, or neither. In many cases, $e'_{ERRATUM} = e'_{WATER}$ (consensus). We choose to focus our manual labeling effort on cases where WATER and ERRATUM disagree and assume that both solutions are right otherwise.

Thus, to label the disagreements between ERRATUM and WATER, we developed a web application (cf. Figure 7) to display the identified elements on both versions of the DOM version pair and label the matching as either correct or wrong.

When we defined the similarity equivalence between two elements (cf.

Definition 3.1), we mentioned the potential subjective part of the measure. To lessen this subjective part and label the proposed matchings as objectively as possible, we systematically recommended the following guidelines:

1. Sometimes, matched elements are not visible (it happens when the visibility of some parts of the page is triggered dynamically). In this case, if elements in both versions are not visible, the locator is skipped, otherwise, the matching is considered as mismatch;
2. Sometimes, a link appears in different locations on the website (often sign-in links). Matching two such links from different locations is considered wrong even though the two links might be assumed to have a similar semantic value. Therefore, we always consider the surrounding of the located element to judge whether the matching is correct or mismatch;

6. Empirical Evaluation

This section evaluates locator repair solutions along with two criteria, accuracy and performance, to answer our research questions.

6.1. Evaluation of Repair Accuracy

In this section we answer RQ.5.1: How does ERRATUM perform in solving the locator repair problem (RQ. 3.2) when compared to state-of-the-art solutions?

Repair accuracy on the MUTATION dataset. Figure 8 summarizes the distribution of the accuracy of ERRATUM and WATER as a violin plot over the $3,291$ version pairs of our MUTATION dataset. For each version pair (D, D'), the reported accuracy ratio corresponds to the ratio of the 15 selected elements from D that are accurately located in D'. The figure shows

There are two ways a repair solution can fail to locate an element $e \in D$ in D': 1. a mismatch, when the original element $e \in D$ has been matched to the wrong element $e' \in D'$, or 2. a no-match, when the algorithm does not manage to locate e in D'. In case of failure, a no-match is always preferred to a mismatch, since a no-match alerts the developer about failure. Thus, considering the two classes of errors on the MUTATION dataset, Table 3 summarizes the ratio of no-match and mismatch reported by both solutions. In particular, the data shows a significant advantage in favor of ERRATUM when it comes to reducing locator mismatches, compared to WATER.

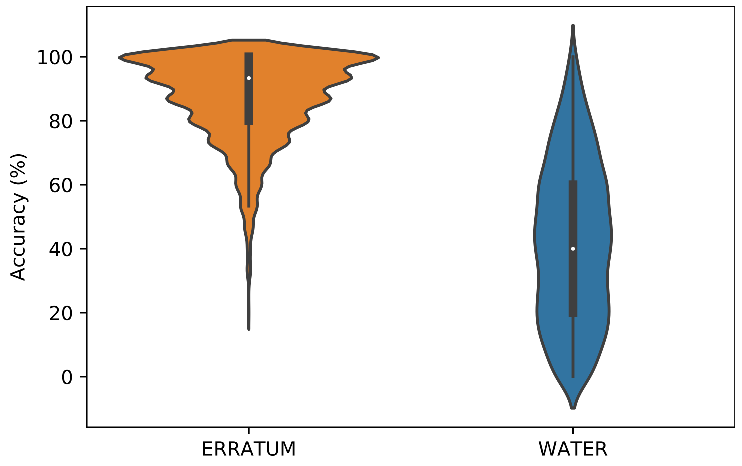


Figure 8: Accuracy distribution of ERRATUM and WATER on the MUTATION dataset.

Table 3: Errors distribution of ERRATUM and WATER on the MUTATION dataset.

	ERRATUM		**WATER**	
correct	$42,876$	(87.0%)	$20,740$	(42.1%)
mismatch	$4,420$	(9.0%)	$26,820$	(54.4%)
no-match	$2,009$	(4.0%)	$1,745$	(3.5%)
Total:	$49,305$	(100%)	$49,305$	(100%)

To further understand which factors influence the accuracy of ERRATUM and WATER (RQ.5.2), we studied the evolution of accuracy according to three factors: 1. the type of mutations applied, 2. the size of the DOM (number of nodes) of the original page D, and 3. the mutation ratio applied to the original page D to obtain the mutant D'.

First, to assess the impact of the mutation type on the accuracy of ERRATUM and WATER, we used a constrained version of the MUTATION dataset with only one mutation operation applied for each mutant. In the original MUTATION dataset, a mutant D' of a page D is obtained by picking a random number l of random nodes $n_1, n_2 ... n_l \in D$ and applying a random mutation type (cf. Table 2) to each node. In the constrained version, we use the same original pages D, but select a single random mutation operation per mutant D'. We then apply the mutation operation to l randomly selected nodes: $n_1, n_2 ... n_l$. For each original page D, the result is a list of mutants such that each mutant D' was obtained using only one mutation operation on a random amount of random nodes. Figure 9 depicts the sensitivity of both

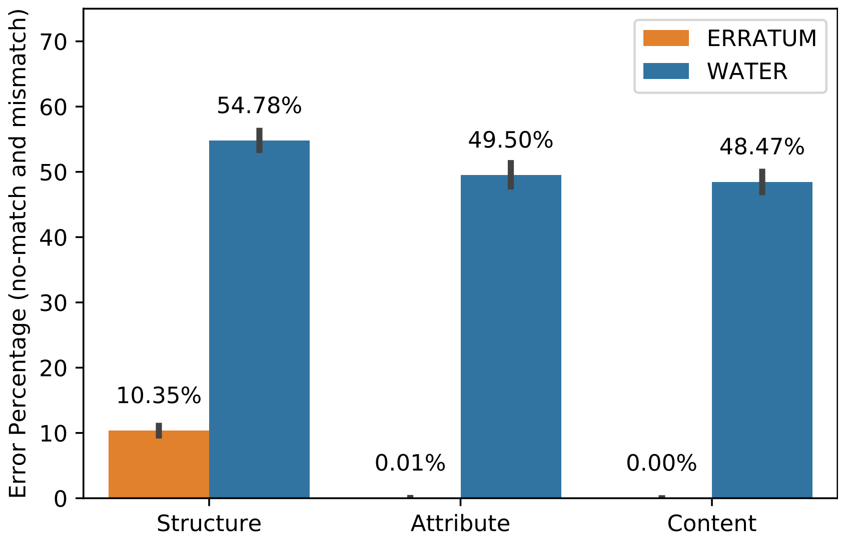


Figure 9: Error percentage according to the mutation type.

locator repair solutions on this alternative dataset. The vertical lines on top of each bar represent the confidence interval. The figure highlights that ERRATUM is almost exclusively sensitive to structural mutations. In particular, the average accuracy of ERRATUM is not sensitive to content mutations on the page, which is expected since the algorithm ignores the content of the nodes by default. The very low sensitivity of ERRATUM to attributes related mutations is more surprising as attributes account for a major part of the similarity metric of the algorithm. For this reason, we believe that the mutation of attributes might have more impact when combined with structural mutations, which does not happen in the constrained MUTATION dataset.

Then, regarding the impact of the size of the DOM, our analysis concludes that WATER loses accuracy when the number of nodes increases (cf. Figure 10), while ERRATUM exhibits a more stable performance. We believe this is a key insight in understanding the limitation of WATER when compared to ERRATUM. For each element $e \in D$ to locate, WATER searches through all same-tag elements in D' (the *candidates*) and picks the closest one to D, with respect to WATER's chosen similarity metric. We believe that the sensitivity of WATER to the number of nodes comes from the fact that the number of *candidate* matchings for a given element e tends to grow with the size of the DOM, which increases the complexity of the ordering-by-similarity task. Conversely, additional nodes provide more "anchor" points

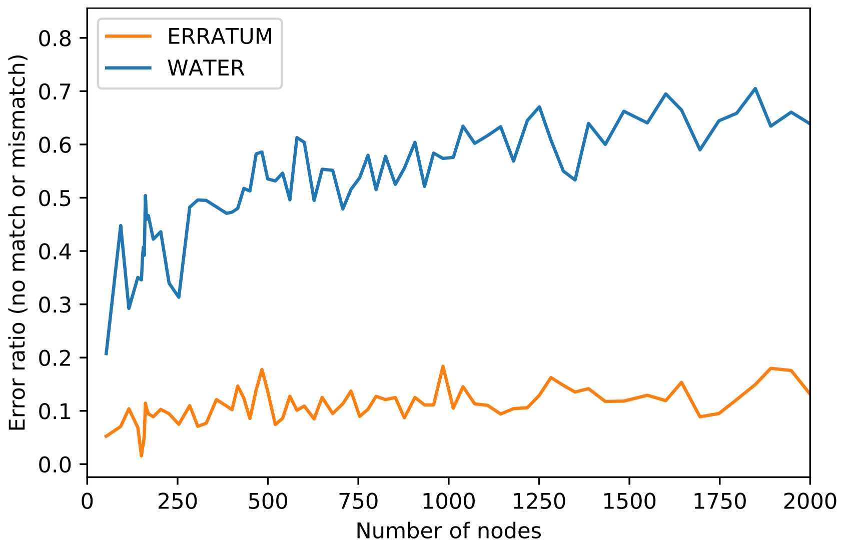


Figure 10: Errors rate evolution according to DOM size.

to ERRATUM, partially compensating the increase in possible combinations.

Finally, regarding the impact of the mutation ratio ($\frac{\#mutations}{\#nodes}$), Figure 11 reports on how ERRATUM and WATER's errors evolve when increasing the number of mutations ($\#mutation$) on the original page D. The figure contains more information than most common box plots, in particular: the stars indicate the average ratio, the horizontal orange lines, the medians whose values also appear above the boxes. As expected, both solutions lose accuracy when the mutation ratio increases, but one can still observe that ERRATUM demonstrates a significant advantage over WATER, no matter the mutation ratio, and exhibiting only 20% of errors on average (against 67% for WATER) when the ratio of mutation exceeds 20% of the nodes.

Repair accuracy on the WAYBACK dataset. Since the WAYBACK dataset does not provide any ground truth matching, we had to manually label the results of the evaluation. We ran both algorithms on the same 34,421 elements. For each element $e \in D$, ERRATUM and WATER returned e'_S and $e'_W \in D' \cup \emptyset$, respectively. In 49.0% of cases, ERRATUM and WATER agreed on a matching element ($e'_S = e'_W \neq \emptyset$). In 13.6% of cases, no solution found a matching element ($e'_S = e'_W = \emptyset$). In 37.4% of cases, ERRATUM and WATER disagreed on the matching element ($e'_S \neq e'_W$ and $(e'_S, e'_W) \neq (\emptyset, \emptyset)$).

A sample of 366 matchings out of the $14,784$ disagreements where labelled

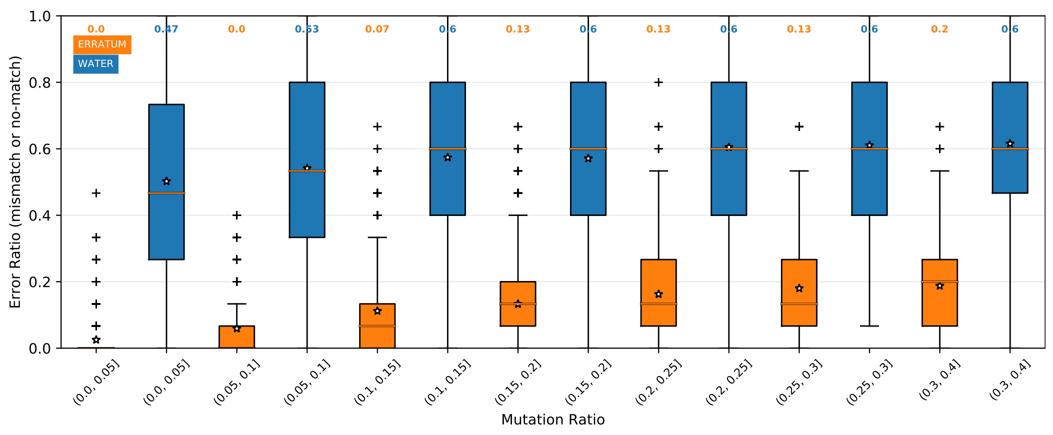


Figure 11: Errors rate evolution according to the mutation ratio.

by web testing experts, which corresponds to a 5% confidence interval at 95%. Table 4 reports on the results of the manual labeling (for disagreements), thus assuming that both WATER and Erratum are correct whenever they agree.

Table 4: Confusion matrix on the Wayback dataset.

		Erratum			
		correct	mismatch	no-match	
WATER	correct	49.0%	1.5%	1.4%	51.9%
	mismatch	26.5%	5.5%	3.3%	35.3%
	no-match	2.8%	1.9%	8.1%	12.8%
		78.3%	8.9%	12.8%	

We further investigated the causes of no-match cases reported by Erratum to assess if these specific cases could be matched by experts. As part of the Wayback experiment, we thus included Erratum's no-match cases in our labelling application (cf. Figure 7) and requested the participants to eventually propose a matching element if a no-match was reported by Erratum. The result of this evaluation, summarized in Figure 12, highlights that a majority of no-match are accurately labeled as such by Erratum, since the participants could not propose a matching element in the target web page. For the few cases where the participants proposed a matching element, we observed that the structure of the DOM tree where subject to strong changes, thus misleading Erratum as already observed in Figure 9.

Comparison of repair accuracy. Interestingly, as shown in Table 5, the

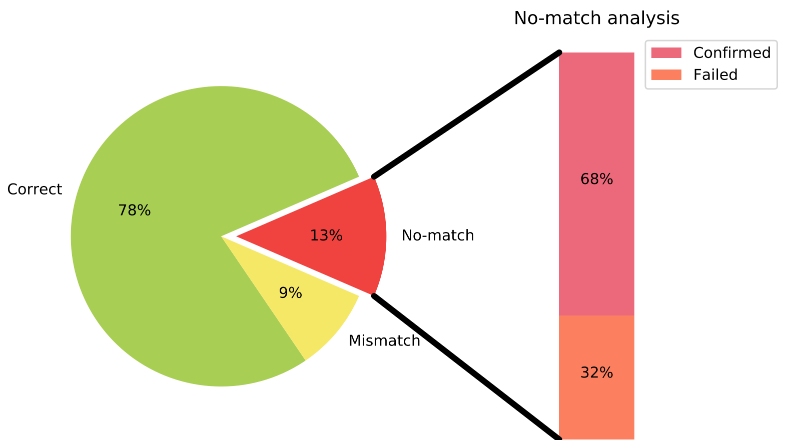


Figure 12: Analysis of matches labeled as no-match by ERRATUM.

accuracy of ERRATUM on the WAYBACK dataset (78.3 % ± 5 %) is 8.7% inferior to the accuracy obtained on the MUTATION dataset (87.0%), while the accuracy of the WATER algorithm is better on the WAYBACK dataset (51.9 % ± 5 %) than on the MUTATION dataset (42.1 %) by 9.8 %. We believe the difference observed between the two datasets is because real-life mutations might not be uniformly distributed. In particular, regarding our sensitivity analysis with regards to types of tree mutations (cf. Figure 9), one can guess that real-life websites are more subject to *content* and *attribute*-related mutations than *structure*-based mutations (cf. Table 2), as the former do not affect the accuracy of ERRATUM. However, since we miss the ground truth for the WAYBACK dataset, we cannot assess this hypothesis and the distribution of real-life mutations.

Table 5: Accuracy summary across datasets.

	MUTATION	WAYBACK
ERRATUM	87.0%	78.3 ± 5%
WATER	42.1%	51.9 ± 5%

6.2. Mutations in the Wayback Dataset

In order to further our understanding of the accuracy results on both datasets, we study the nature of the changes occurring between two versions of a page in the Wayback dataset.

Since the changes in pages from the Wayback dataset are not labeled, there is no way to know with certainty what are the ground truth mutations that were applied between two versions of a page which is the reason why we developed the Robust Locator Benchmark benchmark in the first place.

In the Robust Locator Benchmark, we used the datasets as ground truth to evaluate the matching algorithm on the locator repair problem. Conversely, in this section, we assume the matching algorithm (SFTM) to be correct and use it to provide insights on what mutations occur in the real-life Wayback dataset.

To do so, we estimate the amount and type of mutations between each couple of webpages (D, D') using the corresponding matching M and the following table that associates a mutation type $\forall e$ and $\forall e'$ in D and D' in the described cases:

	Mutation Type	Mutation Category
$(e, e') \in M$ but $\lambda(e) \neq \lambda(e')$	Relabel	Relabel
$\nexists e' \vert (e, e') \in M$	Removal	Structural
$\nexists e \vert (e, e') \in M$	Addition	Structural
$(e, e') \in M$ but $(p(e), p(e') \notin M$	Move	Structural

Where $p(e)$ is the parent of the element e and $\lambda(e)$ is the label of e.

Data Cleaning. In the case of the robust locator benchmark, we only used the subset of the Wayback dataset that had been manually labeled by users. For the manual labeling, problems related to the quality of the data scrapped on the Wayback time machine were then inconsequential as users would just ignore problematic pages and label the next version couple.

On the other hand, before analyzing the full Wayback dataset, it is important to filter out as many possible inconsistent data points as possible. Indeed, while scraping, several kind of failures can occur, in particular:

- one of the versions can be an error page
- one of the versions can be a redirection page
- the website may show "in construction"
- the website may have closed between the first and second snapshot

For this reason, we use two heuristics to prepare the dataset.

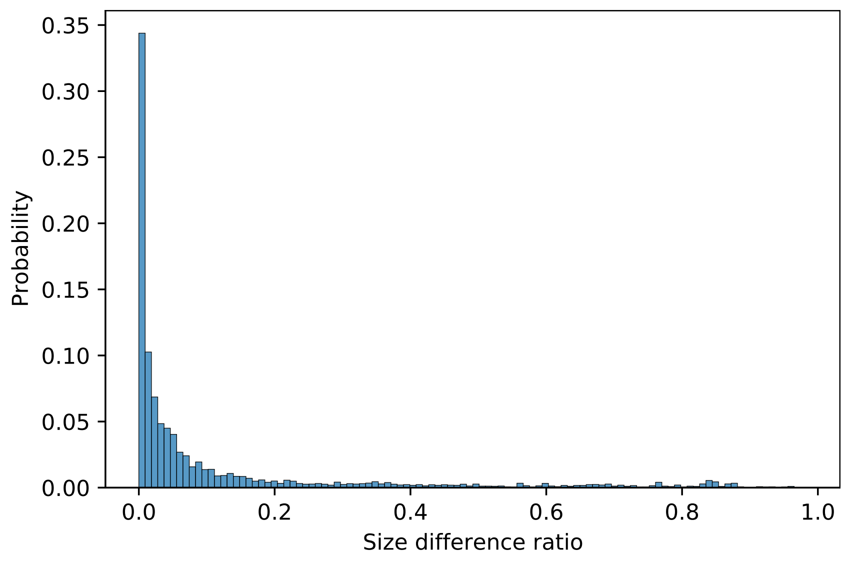


Figure 13: Distribution of the size difference between two versions

1. We filter out the couples where one the two versions have less than 30 nodes. Below 30 nodes, we are usually within one of the cases described above. For example, even a minimalistic webpage like the google homepage contains more than 250 nodes

2. We filter out the couples where the absolute difference in size between the two versions exceeds 50%. Indeed, when the size difference between two versions is that significant, comparing the two pages is likely to be conceptually irrelevant.

Fig 13 shows the distribution of size difference ratios of webpage versions in the Wayback Dataset.

Before filters, the number of couples in the Wayback dataset is 19,161. After the first filter it is 9,580 and after the second 8,848.

Amount of mutations. Fig 15 reports on the amount of mutations by category occurring between two versions of a page.

As expected, the amount of mutations increases with the gap between the versions. Most importantly, the average amount of mutations in the Wayback dataset is 60% which is significantly higher than the average amount of mutations in the Mutation dataset: 20% This difference is probably an

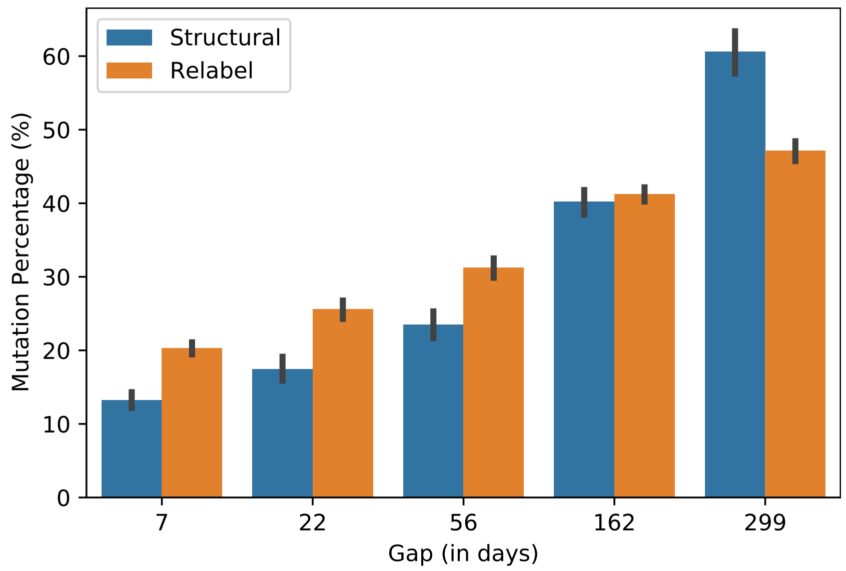


Figure 14: Amount of mutations between two Wayback snapshots according to the time gap between the snapshots.

important factor justifying the differences of accuracy measured on both datasets.

Comparing versions with high gaps is practical because it ensures there will be differences between the versions. In addition, it provides interesting insight about the frequency of change on highly visited webpages. However, in the test use case, the amount of mutations between two subsequent versions should rarely be as high as 60%.

Mutation Type. Using the table described above, we study in more details the different types of mutation occurring between versions. Figure **??** describes the relative ratio of mutation types in the Wayback dataset. The figure shows that relabels are the most common mutation while moves are quite rare. There are two main pieces of information to take into consideration when interpreting the results:

1. If a large block of the page is moved, it will be counted as one move even though the visual change will appear as important

2. The SFTM algorithm is particularly robust to relabels which could also be a factor contributing to explain the observed ratio.

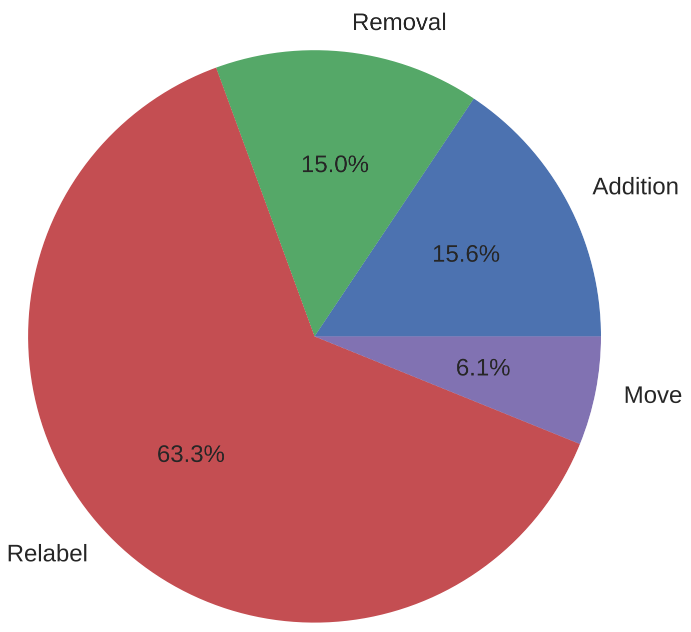


Figure 15: Average ratio of mutations occurring between two versions of a page in the Wayback Dataset

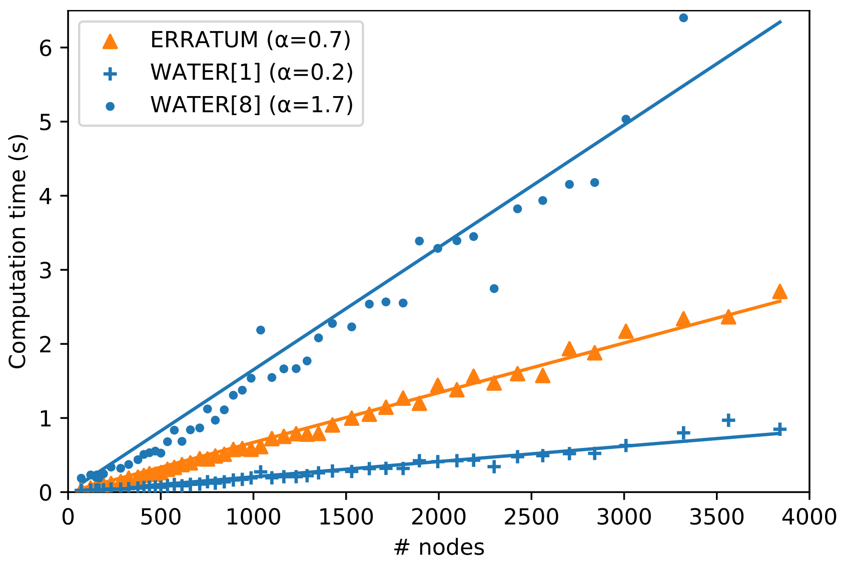


Figure 16: Repair time evolution according to DOM size.

6.3. *Repair Time Evaluation*

In Figure 16, we compare the computation times of Erratum and WATER. The results have been obtained by running both algorithms on the same server containing 252 GB of RAM and an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz.

ERRATUM works differently than WATER: while WATER matches one single element at a time, ERRATUM matches all elements at once. One can observe that WATER is thus faster at locating a single element than Erratum is at locating all elements. However, when the number of locators to repair grows, the computation time of WATER evolves proportionally while the computation time of Erratum remains the same.

More specifically, we compare the evolution of the performance coefficient (α) when increasing the number of locators to repair in a web page. Figure 17 plots these coefficients for Erratum and WATER, so that we can establish that Erratum becomes more efficient than WATER as soon as there are more than 3 locators to repair in a web page.

6.4. *Threats to Validity*

As described in Section 5.3, the Wayback dataset does not include a ground truth (perfect matching). This is why we had to manually label

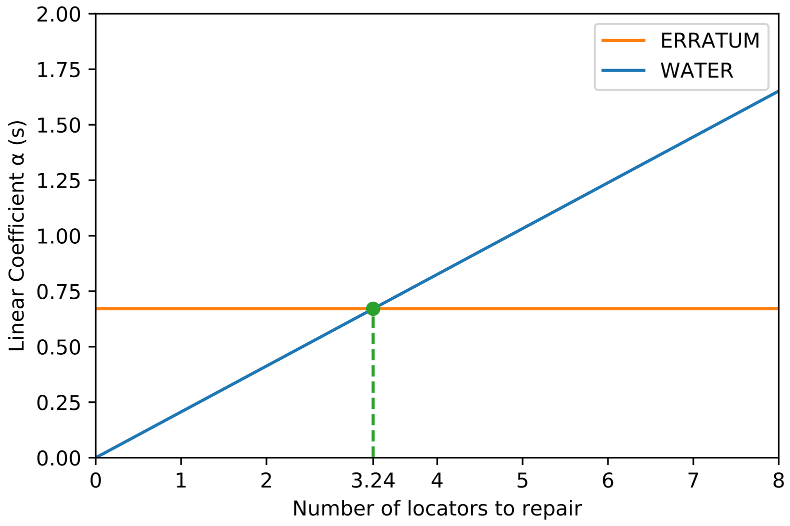


Figure 17: Performances of Erratum and WATER.

a representative sample of the matchings obtained on this dataset, which might have introduced some bias. To mitigate this bias, we recommended systematic and consistent decisions to label the data (cf. Section 5.3).

All our experiments with Erratum adopt the default FTM parameters, as recommended by [23]. Nonetheless, a thorough parameter sensitivity study would probably result in further improving the accuracy of Erratum. Given the results we obtain on a wide diversity of web pages evolutions, we believe that this parameter tuning would only positively and marginally impact the accuracy of Erratum.

In terms of repair time, we discussed the absolute value of repair time for both solutions. However, these values highly depend on the way each tool was implemented and the machines on which the simulations were run. To limit this bias, both solutions were executed on the same Node.js runtime version deployed in the same environment to ensure a proper comparison.

7. Applying Erratum

We have studied how Erratum can help in solving the existing locator repair problem, which is a common problem in web automation scripts. In this section, we envision a more interactive development process made possible by Erratum.

When developing a web automation script, a developer typically opens the page under test in the browser, visually locates the element to interact with and then encodes (or generates) a locator for this element. The locator is then used in the web automation script to select the target element and interact with it. Based on the results achieved by ERRATUM, the perspectives for this work include a new layer of abstraction to the target selection. In this new layer, web automation scripts no longer need to explicitly locate elements on the page directly, but only locate elements using a back-end service H:

1. each web page D under automation is registered in H,
2. for each registered web page D, H exposes a visual interface allowing the developer to visually select an element e and retrieve its unique identifier e_{id},
3. in the web automation script, instead of using a manually encoded (or generated) XPath or CSS locator to select the target element e, the developer sends (D, e_{id}) to H's API, which returns an absolute XPath selecting the target element e,
4. when a web page D registered in H evolves into a new version D', ERRATUM is automatically used to relocate all registered elements e_{id} in D with their matching elements in D',
5. whenever ERRATUM fails (or lacks confidence) to relocate an element, the developer is notified and invited to visually relocate the broken locator.

This approach differs from the test repair approach described in the original WATER article. In the test repair approach, the locator repair is triggered by the failure of one of the test scripts. Once such a test script fails, the test repair solution attempts to determine the cause of the breakage and if it is a locator, repair the locator. The approach we suggest in this section does not include the analysis of any automation script, as locators are updated whenever the page changes.

In many cases, the locator breakage occurs silently (the locator is mismatched and the consequences happen only later in the test script) [4]. In these situations, it is harder to locate the origin of the breakage from the test script. In addition to the obvious gain in time that having a visual-based breakage and repair solution provides, the ERRATUM-based notification process described above would help to detect such possible breakage before the scripts are even run, thus diminishing the chances for a "silent breakage" to occur.

8. Conclusion

In this article, we considered the situation in which the evolution of a web page causes one of its associated automation scripts to break. In the domain of automated web testing, this situation accounts for 74% of test breakages, according to past studies [2]. Our analysis of the state-of-the-art approaches on this topic contributed to formalize the key steps involved in preventing or fixing such a kind of test breakage.

While existing solutions to the locator repair problem treats broken locators individually, we rather propose to apply a holistic approach to the problem, by leveraging an efficient tree matching algorithm. This tree matching approach thus allows Erratum, our solution to repair all broken locators by mapping all the elements contained in an original page, to accurately relocate each of them in its new version at once. To assess Erratum, we created and shared the first reproducible, large-scale datasets of web page locators, combining synthetic and real instances,[4] which has been incorporated in a comprehensive benchmark of Erratum and WATER, a state-of-the-art competitor.[5] Our in-depth evaluation highlights that Erratum outperforms WATER, both in accuracy—by fixing twice more broken than WATER—and performances—by providing faster computation time than WATER when repairing more than 3 locators in a web test script.

References

[1] Imtiaz J, Sherin S, Khan MU, Iqbal MZ. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology* 2019; **113**:1–19.

[2] Hammoudi M, Rothermel G, Tonella P. Why do record/replay tests of web applications break? *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2016; 180–190.

[3] Choudhary SR, Zhao D, Versee H, Orso A. Water: Web application test repair. *Proceedings of the First International Workshop on End-to-End Test Script Engineering*, ACM, 2011; 24–29.

[4]Dataset available from `https://zenodo.org/record/3800130#.XrQb02gzY20`

[5]Benchmark available from `https://zenodo.org/record/3817617#.XrWdqGgzaoQ`

[4] Stocco A, Yandrapally R, Mesbah A. Visual web test repair. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2018; 503–514.

[5] Burg B, Bailey R, Ko AJ, Ernst MD. Interactive record/replay for web application debugging. *Proceedings of the 26th annual ACM symposium on User interface software and technology*, 2013; 473–484.

[6] Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: a selective record-replay and dynamic analysis framework for javascript. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013; 488–498.

[7] Mickens JW, Elson J, Howell J. Mugshot: Deterministic capture and replay for javascript applications. *NSDI*, vol. 10, 2010; 159–174.

[8] Leotta M, Stocco A, Ricca F, Tonella P. Reducing web test cases aging by means of robust xpath locators. *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, IEEE, 2014; 449–454.

[9] Leotta M, Stocco A, Ricca F, Tonella P. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process* 2016; **28**(3):177–204.

[10] Leotta M, Stocco A, Ricca F, Tonella P. Meta-heuristic generation of robust xpath locators for web testing. *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*, IEEE, 2015; 36–39.

[11] Yandrapally R, Thummalapenta S, Sinha S, Chandra S. Robust test automation using contextual clues. *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ACM, 2014; 304–314.

[12] Bajaj K, Pattabiraman K, Mesbah A. Synthesizing web element locators (t). *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015; 331–341.

[13] Leotta M, Stocco A, Ricca F, Tonella P. Using multi-locators to increase the robustness of web test cases. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2015; 1–10.

[14] Zheng Y, Huang S, Hui Zw, Wu YN. A method of optimizing multi-locators based on machine learning. *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, IEEE, 2018; 172–174.

[15] Long Z, Wu G, Chen X, Chen W, Wei J. Webrr: self-replay enhanced robust record/replay for web application testing. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020; 1498–1508.

[16] Hammoudi M, Rothermel G, Stocco A. Waterfall: An incremental approach for repairing record-replay tests of web applications. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, 2016; 751–762.

[17] Chang TH, Yeh T, Miller RC. Gui testing using computer vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2010; 1535–1544.

[18] Leotta M, Stocco A, Ricca F, Tonella P. Pesto: Automated migration of dom-based web tests towards the visual approach. *Software Testing, Verification And Reliability* 2018; **28**(4):e1665.

[19] Alegroth E, Nass M, Olsson HH. Jautomate: A tool for system-and acceptance-test automation. *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, IEEE, 2013; 439–446.

[20] Tai KC. The tree-to-tree correction problem. *Journal of the ACM (JACM)* 1979; **26**(3):422–433.

[21] Jiang T, Wang L, Zhang K. Alignment of trees—an alternative to tree edit. *Annual Symposium on Combinatorial Pattern Matching*, Springer, 1994; 75–86.

[22] Brisset S, Rouvoy R, Pawlak R, Seinturier L. SFTM: Fast Comparison of Web Documents using Similarity-based Flexible Tree Matching 2020.

[23] Kumar R, Talton JO, Ahmad S, Roughgarden T, Klemmer SR. Flexible tree matching. *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (AAAI)*, 2011.

[24] Bringmann K, Gawrychowski P, Mozes S, Weimann O. Tree edit distance cannot be computed in strongly subcubic time (unless apsp can). *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 2018; 1190–1206.

[25] Pawlik M, Augsten N. Tree edit distance: Robust and memory-efficient. *Information Systems* 2016; **56**:157–173.

[26] Jiang T, Wang L, Zhang K. Alignment of trees—an alternative to tree edit. *Theoretical Computer Science* 1995; **143**(1):137–148.

[27] Valiente G. An efficient bottom-up distance between trees. *spire*, 2001; 212–219.

[28] Zhang K. A constrained edit distance between unordered labeled trees. *Algorithmica* 1996; **15**(3):205–222.

[29] Jones KS. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 1972; .

[30] Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E. Equation of state calculations by fast computing machines. *The journal of chemical physics* 1953; **21**(6):1087–1092.