Erratum: Repairing Broken Locators in Web Test Scripts using Tree Matching

Submission #356 (10+1 pages)

ABSTRACT

Web applications are constantly evolving to integrate new features and/or fix reported bugs. All these web applications are relying on the *Document Object Model* (DOM) to render content from client's browsers. Element locators are identifiers used to navigate across the DOM and are the keystones to automate tasks on web applications. Unfortunately, element locators tend to become fragile when the underlying web pages evolve over time. Robust locators have been introduced to overcome this issue, but they fail to repair broken locators on complex and dynamic web applications. 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, which is often challenging.

This paper adopts a different perspective on this problem by introducing a new locator repair solution that leverages tree matching 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 legacy web applications. Our empirical results demonstrate that Erratum outperforms the accuracy of WATER, a state-of-the-art solution, by 67%.

KEYWORDS

Element locator, Tree matching, Web testing

1 INTRODUCTION

The implementation of automated tasks on web applications, 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 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 test cases that cover the modified web pages.

To address this issue, state-of-the-art approaches in this area 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 this area fail to relocate a broken locator with a high probability, thus leaving all the related web test scripts as broken [11].

In particular, state-of-the-art solutions to the locator repair problem, WATER [6] and VISTA [23], tend to rely on the intrinsic properties of the element whose locator needs repairing to locate its matching element on the new 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 paper, we adopt a more holistic approach to the locator repair problem: instead of focusing on the element whose locator is broken individually, we use tree matching 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 locator with better accuracy. Additionally, if more than one locator is broken on a given web page, our approach will repair all of them at once. The holistic solution we propose, named Erratum,[1] more specifically leverages an efficient flexible tree matching algorithm to repair all broken locators by matching all changes in a web page with high accuracy.

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

When evaluated on both datasets, our results show that Erratum outperforms the state-of-the-art solution, namely WATER [6], both in accuracy (67% improvement on average) and performances, when more than 2 locators require to be repaired in a web page.

Overall, the key contributions of this paper consist of:

(1) Applying flexible tree matching to solve the locator repair problem,
(2) Providing a novel, reproducible, large-scale benchmark dataset allowing to evaluate both the robust locator and locator repair problems,
(3) Reporting on an empirical evaluation of how our approach performs when solving the locator repair problem.

The remainder of this paper 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 paper. Section 4 introduces our approach, Erratum, which leverages a scalable tree matching solution that we identified. Section 5 describes the locator benchmark we designed and implemented.

[1] Erratum stands for "*rEpaiRing bRoken locATors Using tree Matching*"
[2] https://www.alexa.com/topsites

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 intend to deliver a novel contribution to the locator repair problem. In this section, we thus introduce the required background and we describe state-of-the-art approaches to web testing, and in particular 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 [4, 19, 22] 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:

```html
<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* [15].

In practice, automated tests are often subject to breakages [11]. It is important to understand that a test *breakage* is different from a test *failure* [23]: 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, [11] reports that 74 % of web tests break because one of the included locators fails to locate an element in a web page.

While we focus on web test repair, some previous work focused on GUI test repair for desktop applications [7–9, 18, 28]. This work most often uses platform-specific features hardly transposable to the web, as Zhang *et al.* [28] who rely on the methods associated to GUI elements.

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 [15], ROBULA+ [16], which are algorithms that apply successive refining transformations from a raw XPath query until it yields a locator that exclusively returns the desired element, and [26] where locators are based on contextual clues. While automatically generating locators can speed 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 [11].

2.3 Repairing Web Element Locators

While some solutions to the robust locator problem, as presented above, aim to prevent locators to break, others focus on repairing broken locators. In this context, the repairing 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 [6] 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.* [10] 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 [23] is a recent technique that adopts computer vision to repair locators. VISTA falls within the category of computer vision-aided web tests [1, 5, 17]. 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).

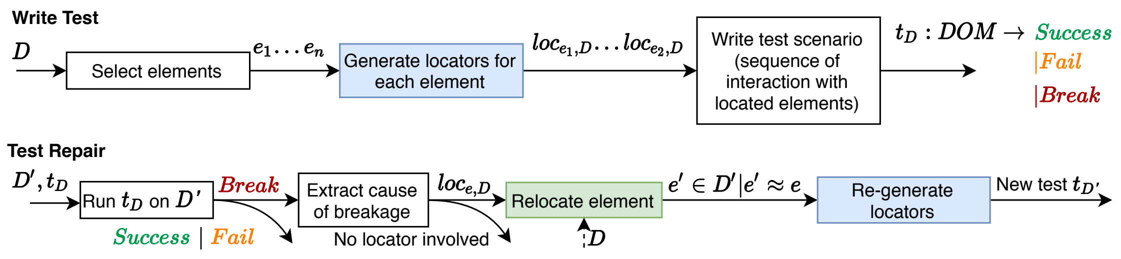
3 LOCATOR PROBLEM STATEMENT

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 is generated/written.

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.

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



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
No locator involved
$loc_{e,D}$
Relocate element
D
$e' \in D' | e' \approx e$
Re-generate locators
New test $t_{D'}$

Figure 1: Locator problem statement combining the robust locator (in blue) and the locator repair (in green) problems.

web page observed before and after a change, respectively. More specifically, we define the following similarity notations:

(1) $D \approx D'$ if tests 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 $E = e_1 ... e_n$ and $E' = e'_1 ... e'_{n'}$, $E \approx E'$ 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 couple $(loc_{E,D}, eval)$ is a **locator** of E with regard to D if:

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

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:

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

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).

4 REPAIRING LOCATORS WITH ERRATUM

The previous section formalized both *robust locator* and *locator repair* problems. The approach we report in this paper, ERRATUM, therefore matches the DOM trees of 2 versions of a webpage to solve the *locator repair* problem. Several tree matching solutions exist in the literature, such as *Tree Edit Distance* (TED) [24] or tree alignment [12]. 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

Adopting 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 on a small example the benefits of a more holistic approach using tree matching. In the example, the locator of the 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 approach) is often challenging and can easily lead to a mismatch. Therefore, we privilege a more holistic approach based on tree matching. By using tree matching (cf. right-bottom of Figure 2), matching the parent of the element to locate (div#menu) brings a strong contextual clue to accurately relocate the element a_1' whose locator was broken (in blue).

Formally, given a couple 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;

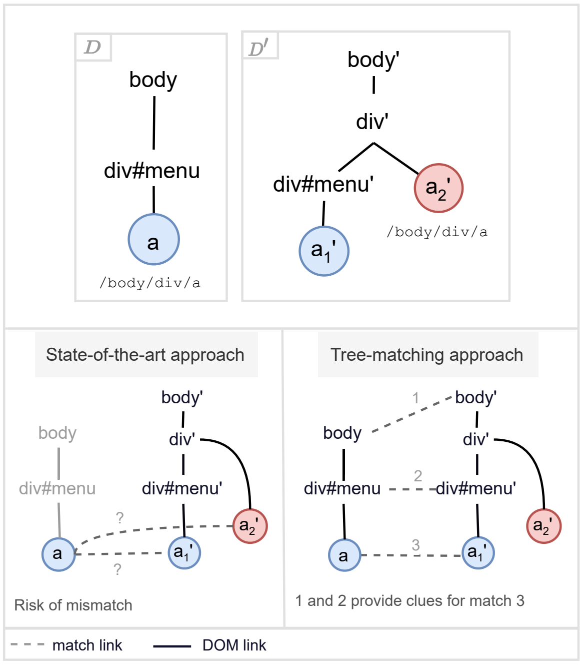


Figure 2: State-of-the-art Vs. tree matching locator repair.

(2) apply tree matching to T_1, T_2 yielding a matching $M \subset nodes(T_1) \times nodes(T_2)$. If the resulting matching M is accurate, then $\forall (e, e') \in M, e \approx e'$;
(3) use the resulting matching M to relocate the locator(s).

Regarding the test repair process illustrated in Figure 1, our approach thus fits in the green block 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 [15], and therefore considered as out of the scope of this paper.

4.2 Using a Scalable Tree Matching Algorithm

The state-of-the-art approach to match two trees is *Tree Edit Distance* (TED) [24]. When comparing two trees T_1 and T_2, TED-based approaches rely on finding the optimal sequence of relabels, insertions and deletions that transforms T_1 into T_2. Unfortunately, TED might be unsuitable to match real-life web pages due to its restrictions [14]: (1) if two nodes n and m are matched, the descendants of n can only match with the descendants of m, and (2) the order of siblings must be preserved. Furthermore, TED is computationally expensive ($O(n^3)$ for the worst-case complexity [2]) and, more practically, Brisset *et al.* [3] pointed out that applying the state-of-the-art implementation of TED, named APTED [21], on the *YouTube* page takes more than 4 minutes. We believe that, in addition to qualitative restrictions, such computation times are not acceptable when repairing locator on real websites.

Further studies of TED proposed to improve computation times [13, 25, 27], but at the cost of even more restrictive constraints on the produced matching (*e.g.*, the tree alignment problem [13] 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 [14]. FTM models tree matching as an optimization problem: given two trees T_1 and T_2 how to build a set of tuples $(n, m) \in T_1 \times T_2$ such that the similarity between all selected node tuples is maximal. The similarity used by FTM combines both the labels and the topology of the tree. However, as shown in [3], 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 webpage observed in our dataset is 1,507. Consequently, we believe that such computation times make FTM unpractical in the context of locator repair.

Nonetheless, the *Similarity-based Flexible Tree Matching* (SFTM) algorithm is an extension to FTM aiming to improve the computation times of FTM without any restriction on the resulting matching [3]. In [3], SFTM is compared to the state-of-the-art TED implementation (APTED [20]) on webpage DOM trees. Their results show that SFTM yields similar qualitative results to APTED, while drastically improving computation times, by up to two orders of magnitude. For example, while APTED matches the *YouTube* homepage in 4 minutes, SFTM, takes less than 2 seconds.

Given the above review, we integrated SFTM as the reference tree-matching algorithm implementation to relocate elements in Erratum.

5 THE ROBUST LOCATOR BENCHMARK

In the context of this paper, we are interested in addressing 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, starts by describing the benchmark we developed to assess these questions.

5.1 Evaluated Locator Repair Solutions

We compare two solutions: (1) Erratum, our approach to repair broken locators by leveraging tree matching, and (2) WATER [6].

WATER was designed to repair web test cases. The original algorithm analyses a given test case, finds the origin of the test breakage, and suggests potential repairs to the developer. 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 implemented this part of the WATER algorithm to compare its performance to Erratum.

We initially considered VISTA [23] 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*), which is the worst-case scenario for Erratum. 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 paper, 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 tuples (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 [15, 16] and robust locator [6, 10, 23] 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** [3] 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 tuple (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 tuples 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 tuples**: 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 tuples	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 tuples. 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.

5.2.1 Mutation dataset. We borrow the technique used to generate the Mutation dataset from [3]. 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 extracted from [3] describes the set of selected mutations that can be possibly applied to an element of the DOM.

The original websites from which mutants were generated were randomly selected from the Top 1K Alexa. Figure 3 depicts the distribution of DOM sizes in this synthetic dataset.

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

Table 2: Mutations applied in the Mutation dataset [3].

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

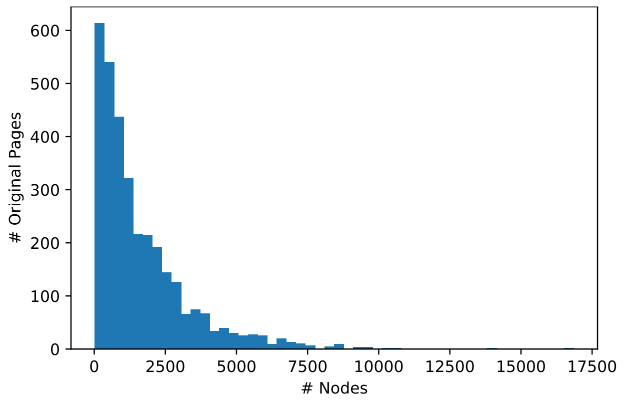


Figure 3: Distribution of DOM sizes (in number of nodes) in the MUTATION dataset.

5.2.2 WAYBACK *dataset.* This dataset encloses a list of (D, D') DOM couples 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:

$$versionsExplorer :: \quad (url, duration) \rightarrow \quad timestamp[\,]$$
$$versionResolver :: \quad (url, timestamp) \rightarrow \quad document$$

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 3).

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 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 tuple (D, D') from a given 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'. Instead, for each version tuple, we randomly select up to 15 clickable elements from D. Selecting clickable elements is indeed the most common use case for web UI testing (to trigger interactions), and WATER has special heuristics to enhance its accuracy on links. 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). 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 select 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 of located elements is 0.5 %.

5.3 Labeling of the Matched Elements

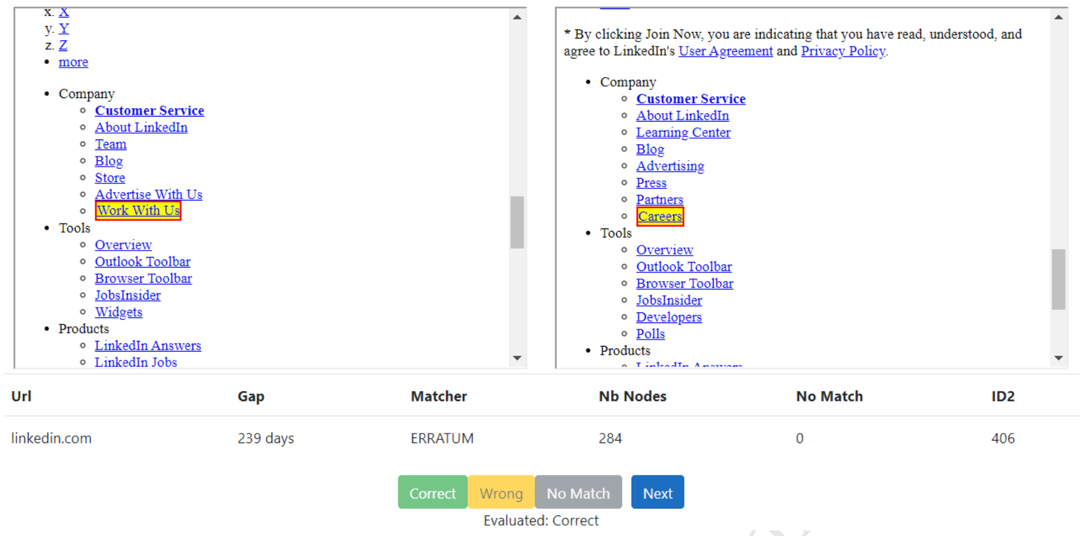
On the MUTATION dataset, the signatures attributes are preserved after mutations (but ignored when applying either locator repair solution), thus providing the ground truth matching between the DOMs of a version tuple.

For the WAYBACK dataset though, this information is not available. For each version tuple (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, $e'_{ERRATUM} = e'_{WATER}$. In this situation, while there is still the possibility that both solutions are wrong, we chose to focus our manual labeling effort on cases where WATER and ERRATUM disagree and assume that both solutions are right otherwise.

To manually label the disagreements between ERRATUM and WATER, we developed a web application (cf. Figure 4) to display the identified elements on both versions of the DOM version tuple 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 provide labeling as consistent as possible, we systematically followed the guidelines below:

(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;


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

Url	Gap	Matcher	Nb Nodes	No Match	ID2
linkedin.com	239 days	ERRATUM	284	0	406

Evaluated: Correct

Figure 4: 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.

(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;

(3) List of records is particularly challenging for repair solutions. For example, *YouTube* displays a list of videos. These videos are not the same from one day to another. Repair solutions often tend to match such videos according to their position in the grid. While this a valuable result, we decided to label such matches as mismatch since the matched elements do not strictly have the same semantic value.

The above list is an exhaustive list of non-trivial cases encountered during manual labeling. The last element—list of records—highlights the limits of state-of-the-art approach for locator repair. In the context of web testing, the systematic labeling policy that we applied consisted in favoring semantics (*e.g.*, the content of a video) over structure (its position on the grid, cf. similarity definition 3.1). Assuming that test data (*e.g.*, a list of videos) is stable over time, a semantic similarity matching should thus detect and follow the structural changes of a page over time by leveraging the semantic stability to repair the broken locator of a given element. This situation points towards the insufficiency of the underlying model used in existing locator related solutions and will be the object of further studies (see Section 7).

6 EMPIRICAL EVALUATION

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

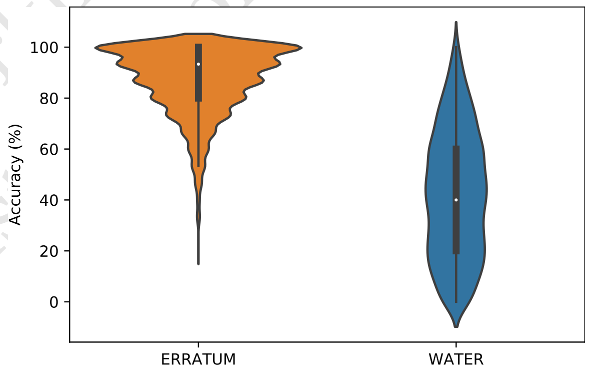


Figure 5: Accuracy distribution per version tuple on the Mutation dataset.

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 5 summarizes the distribution of the accuracy of Erratum and WATER over the 3, 291 version tuples of our Mutation dataset. For each version tuple (D, D'), the reported accuracy ratio thus corresponds to the ratio of the 15 selected elements from D that are accurately located in D'.

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.

Table 3: Errors distribution 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 6 depicts the sensitivity of both locator repair solutions on this alternative dataset. The figure shows that ERRATUM is almost exclusively sensitive to structural mutations. In particular, the accuracy of ERRATUM is not sensitive to content mutations on the page, which is expected since the algorithm ignores the content of the nodes. 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 7), 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 to ERRATUM, partially compensating the increase in possible combinations.

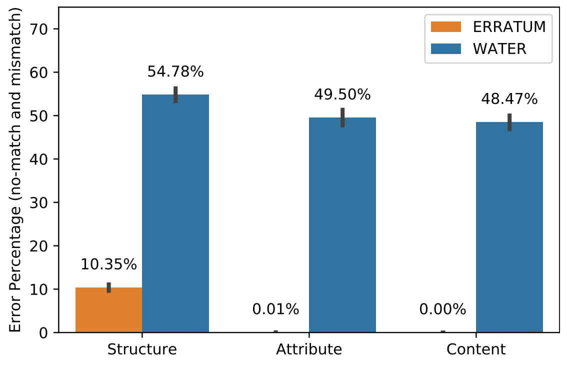


Figure 6: Error percentage according to the mutation type.

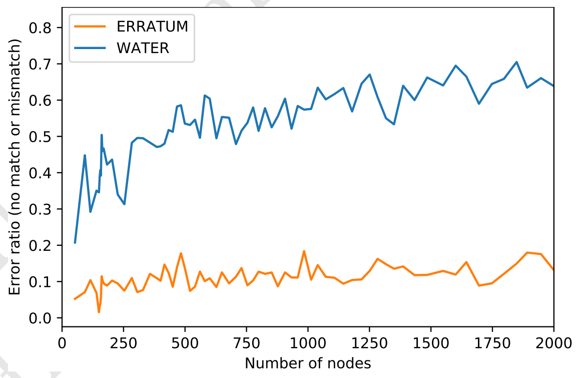


Figure 7: Errors rate evolution according to DOM size.

Finally, regarding the impact of the mutation ratio ($\frac{\#mutations}{\#nodes}$), Figure 8 reports on how ERRATUM and WATER's errors evolve when increasing the number of mutations ($\#mutation$) on the original page D. 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 $(e'_S, e'_W) \neq (\emptyset, \emptyset)$).

We manually labeled a sample of 366 matchings out of the 14, 784 disagreements, 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.

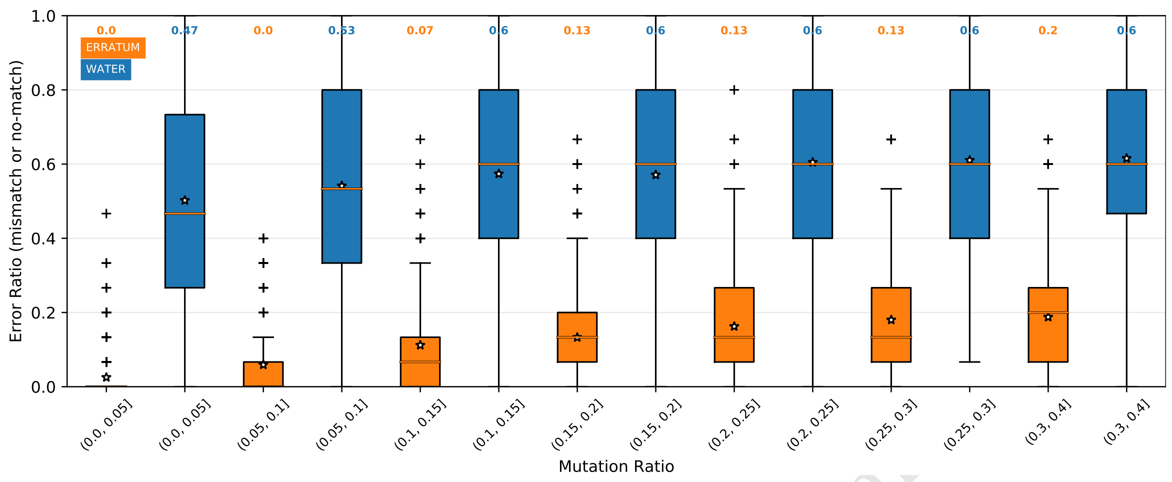


Figure 8: Errors rate evolution according to the mutation ratio.

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%	

Comparison of repair accuracy. Interestingly, as shown in Table 5, the 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 6), 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 Repair Time Evaluation

In terms of computation times, the comparison is not straightforward as we measure the time to repair a matching between all the elements in the tree for ERRATUM and the time to repair one and eight elements with WATER. In Figure 9, we compare the computation times of ERRATUM and WATER. It is interesting to note

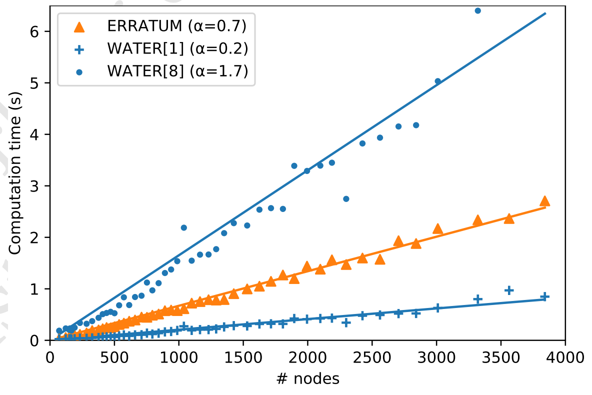


Figure 9: Repair time evolution with DOM size.

that, even though the theoretical worst-case complexity of SFTM is $O(N\sqrt{N})$ (where N, is the number of nodes), the observed computation times of ERRATUM rather seem to be linear with the number of nodes. As expected, WATER is faster at locating a single element than ERRATUM is at locating all elements. However, when the number of locators to repair grows elements, ERRATUM becomes more efficient and accurate.

More specifically, we compare the evolution of the performance coefficient (α) when increasing the number of locators to repair in a web page. Figure 10 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 2 locators to repair in a web page.

6.3 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 a representative sample of the matchings obtained on this dataset, which might have introduced some bias. To mitigate this

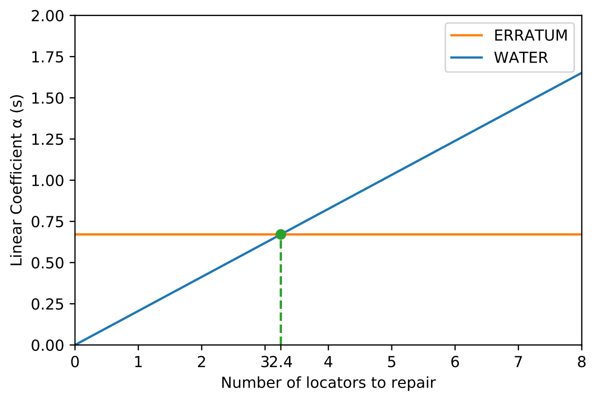


Figure 10: Performances of Erratum and WATER.

bias, we applied systematic and consistent decisions to label the data (cf. Section 5.3).

All our experiments with Erratum adopt the default SFTM parameters, as recommended by [3, 14]. 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 PERSPECTIVES

We have studied how Erratum can help in solving the existing locator repair problem within a traditional web development process, as described in Figure 1. In this section, we envision a more interactive development process made possible by Erratum.

When developing a web test script, a developer typically opens the page under test in her 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 test 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 particular, in this new layer, web test scripts do 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 test 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 test 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 paper. 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 test 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) [23]. 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 tests are even run, thus diminishing the chances for a "silent breakage" to occur.

8 CONCLUSION

In this paper, we considered the situation in which the evolution of a web page causes one of its associated test scenarios to break. This situation accounts for 74% of test breakages, according to past studies [11].

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. From all the steps involved, this paper focuses on the particularly challenging *locator repair* problem—*i.e.*, relocating an element taken from the original page in its new version.

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 then 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 2 locators in a web test script.

Finally, we demonstrate that Erratum is weakly impacted by the mutation rate and DOM size of web pages, which makes it a serious candidate for being deployed in web developments.

[4]Dataset available from https://zenodo.org/record/3800130#.XrQb02gzY20
[5]Benchmark available from https://zenodo.org/record/3817617#.XrWdqGgzaoQ

REFERENCES

[1] Emil Alegroth, Michel Nass, and Helena H Olsson. 2013. JAutomate: A tool for system-and acceptance-test automation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 439–446.

[2] Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 2018. Tree edit distance cannot be computed in strongly subcubic time (unless APSP can). In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 1190–1206.

[3] Sacha Brisset, Romain Rouvoy, Renaud Pawlak, and Lionel Seinturier. 2020. SFTM: Fast Comparison of Web Documents using Similarity-based Flexible Tree Matching. arXiv:cs.DB/2004.12821

[4] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 473–484.

[5] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. 2010. GUI testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1535–1544.

[6] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. Water: Web application test repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering*. ACM, 24–29.

[7] Brett Daniel, Danny Dig, Tihomir Gvero, Vilas Jagannath, Johnston Jiaa, Damion Mitchell, Jurand Nogiec, Shin Hwei Tan, and Darko Marinov. 2011. ReAssert: a tool for repairing broken unit tests. In *Proceedings of the 33rd International Conference on Software Engineering*. 1010–1012.

[8] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2015. Sitar: Gui test script repair. *Ieee transactions on software engineering* 42, 2 (2015), 170–186.

[9] Mark Grechanik, Qing Xie, and Chen Fu. 2009. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 408–418.

[10] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 2016. Waterfall: An incremental approach for repairing record-replay tests of web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 751–762.

[11] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. 2016. Why do record/replay tests of web applications break?. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 180–190.

[12] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. 1994. Alignment of trees—an alternative to tree edit. In *Annual Symposium on Combinatorial Pattern Matching*. Springer, 75–86.

[13] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. 1995. Alignment of trees—an alternative to tree edit. *Theoretical Computer Science* 143, 1 (1995), 137–148.

[14] Ranjitha Kumar, Jerry O. Talton, Salman Ahmad, Tim Roughgarden, and Scott R. Klemmer. 2011. Flexible tree matching. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (AAAI)*.

[15] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2014. Reducing web test cases aging by means of robust XPath locators. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. IEEE, 449–454.

[16] 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* 28, 3 (2016), 177–204.

[17] 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), e1665.

[18] Atif M Memon. 2008. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18, 2 (2008), 1–36.

[19] James W Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications.. In *NSDI*, Vol. 10. 159–174.

[20] Mateusz Pawlik and Nikolaus Augsten. 2011. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment* 5, 4 (2011), 334–345.

[21] Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Information Systems* 56 (2016), 157–173.

[22] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 488–498.

[23] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 503–514.

[24] Kuo-Chung Tai. 1979. The tree-to-tree correction problem. *Journal of the ACM (JACM)* 26, 3 (1979), 422–433.

[25] Gabriel Valiente. 2001. An Efficient Bottom-Up Distance between Trees.. In *spire*. 212–219.

[26] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust test automation using contextual clues. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 304–314.

[27] Kaizhong Zhang. 1996. A constrained edit distance between unordered labeled trees. *Algorithmica* 15, 3 (1996), 205–222.

[28] Sai Zhang, Hao Lü, and Michael D Ernst. 2013. Automatically repairing broken workflows for evolving GUI applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 45–55.