# Towards Understanding Web Applications: Automated Abstraction Inference and its Applications

Sacha Brisset

▶ **To cite this version:**

Sacha Brisset. Towards Understanding Web Applications: Automated Abstraction Inference and its Applications. Computational Engineering, Finance, and Science [cs.CE]. Université de lille; inria lille, 2022. English. NNT : . tel-04079897v1

**HAL Id: tel-04079897**

**https://theses.hal.science/tel-04079897v1**

Submitted on 11 Dec 2022 (v1), last revised 24 Apr 2023 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

| *Université* | *École doctorale* | *Centre de recherche* | *Entreprise partenaire* |
| --- | --- | --- | --- |
| Université de Lille | Madis | Inria, Spirals | Mantu |


Thèse de doctorat pour obtenir le grade de docteur en Informatique

# Towards Understanding Web Applications: Automated Abstraction Inference and its Applications

*Vers une comprehension des applications web: inference non-supervisée d'abstraction et ses applications*


**Sacha Brisset**

Directeurs:
*Lionel* Seinturier
*Renaud* Pawlak

Encadrant:
*Romain* Rouvoy


Soutenue le **05/12/2022** devant le jury composé de:

*Jean Christophe* Routier - Examinateur - Professeur des universités - Université de Lille
*Philippe* Collet - Examinateur - Professeur des universités - Université de Nice
*Xavier* Blanc - Rapporteur - Professeur des universités - Université de Bordeaux
*Dalila* Tamzalit - Rapporteur - Maitresse de conférences - Université de Nantes

,

# Abstract

Web applications are at every corner of modern society. The largest web applications can serve millions of people. These applications are expected to be strongly reliable and stable yet capable to evolve to adapt to its users. At such a scale, these expectations can only be met through huge resources and time. For this reason, it is critical to further our ability to understand the structure of web applications to ease their maintenance and evolution.

In this thesis, we explore web application structure through a variety of lenses: web testing, data extraction and web analytics. Our study shows that many web-related research, regardless of the research domain, suffer greatly from the lack of a generic fully unsupervised web application abstraction inference solution. We attempt to develop such a solution iteratively leading to three main contributions:

**SFTM** Similarity-based Tree Matching, an algorithm allowing to match two web pages. Compared to traditional, generic Tree Matching algorithms, SFTM produces better matchings for computation times several orders of magnitude smaller.

**ERRATUM** an approach allowing to repair locators on web applications. ERRATUM strongly improves the quality of repairs for little to no overhead. We integrated ERRATUM to a widely used open-source testing framework.

**APPSTRACT** an approach to automatically generate an abstraction of a web application. APPSTRACT combines intra-page abstraction and inter-page abstraction using SFTM to generate robust and semantically-rich application-wide locator identifiers for each element of a webpage.

We believe our work opens up many new possibilities in a variety of research domains, in particular: the computation speed of SFTM enables approaches that were previously unpractical with generic tree matching and the approach we describe in APPSTRACT could pioneer new web analytics or web testing generation solutions based on web application abstraction.

# Resumé

Les applications Web sont omniprésentes dans la société moderne. Certaines applications Web peuvent servir des millions de personnes. Ces applications se doivent d'etre fiables et stables tout en étant capables d'évoluer pour s'adapter à ses utilisateurs. À une telle échelle, ces attentes ne peuvent être satisfaites qu'avec d'importantes ressources. Pour cette raison, il est essentiel d'approfondir notre capacité à comprendre la structure des applications Web pour faciliter leur maintenance et leur évolution.

Dans cette thèse, nous explorons la structure des applications Web à travers plusieurs perspectives : les tests Web, l'extraction de données et l'analyse Web. Notre étude montre que de nombreuses recherches liées au Web, quel que soit le domaine de recherche, souffrent grandement de l'absence d'une solution générique d'inférence d'abstraction d'applications Web entièrement non supervisée. Nous tentons de développer une telle solution de manière itérative aboutissant à trois contributions principales :

**SFTM** Similarity-based Tree Matching, un algorithme permettant de faire correspondre deux pages web. Comparé aux algorithmes de correspondance d'arbres génériques traditionnels, SFTM produit de meilleures correspondances plus rapidement.

**ERRATUM** une approche permettant de réparer les localisateurs sur les applications web. ERRATUM améliore fortement la qualité des réparations pour peu ou pas de frais généraux. Nous avons intégré ERRATUM à logiciel de test open source largement utilisé.

**APPSTRACT** une approche pour générer automatiquement une abstraction d'une application web. APPSTRACT combine l'abstraction intra-page et l'abstraction inter-page à l'aide de SFTM pour générer des identifiants de localisation robustes et sémantiquement riches à l'échelle de l'application pour chaque élément d'une page Web.

Nous pensons que notre travail ouvre de nombreuses nouvelles possibilités dans une variété de domaines de recherche, en particulier : la vitesse de calcul de SFTM permet des approches qui n'étaient auparavant pas possibles avec l'appariement d'arbres génériques et l'approche que nous décrivons dans APPSTRACT pourrait ouvrir la voie à de nouvelles analyses Web ou à des solutions de génération de tests Web basées sur l'abstraction d'applications Web.

# Acknowledgements

I would like to thank all the people that helped me through this thesis. In particular, I would like to thank Romain Rouvoy who simply taught me how to be a researcher. Thank you for being available at literally any time. Thank you for the many days and nights writing, correcting and revising our various papers.

I would like to thank Renaud Pawlak who trusted me in the first place and with whom I shared hours and hours of endless discussions in front of an office, a whiteboard or even a meal. Thank you Renaud for your boundless ambition and imagination.

I would like to thank Lionel Seinturier for being the rock in this thesis. Thank you for keeping me on track and never giving up on me even on the most delicate times.

I would like to strongly thank Mantu who funded this research. I had the most wonderful time within Mantu lab team.

Finally, I would like to thank my partner and my family for supporting me through the sometimes difficult moments of the thesis.

# Contents

**1 Introduction** **1**
- 1.1 Motivation . . . . . 1
- 1.2 Objectives . . . . . 3
  - 1.2.1 Tree Matching . . . . . 3
  - 1.2.2 Robust and Semantically Rich Locators . . . . . 4
  - 1.2.3 Web Page Abstraction . . . . . 4
- 1.3 Contributions . . . . . 5
  - 1.3.1 SFTM . . . . . 5
  - 1.3.2 Erratum . . . . . 6
  - 1.3.3 Erratum Industrial Case Study . . . . . 7
  - 1.3.4 Appstract . . . . . 7

**2 Tree Matching** **9**
- 2.1 Introduction . . . . . 9
- 2.2 Related Work . . . . . 12
- 2.3 Flexible Tree Matching (FTM) . . . . . 14
  - 2.3.1 FTM Notations and Overview . . . . . 14
  - 2.3.2 Cost Estimation . . . . . 16
  - 2.3.3 Metropolis Algorithm . . . . . 16
  - 2.3.4 Complexity Analysis . . . . . 17
- 2.4 Similarity-based FTM (SFTM) . . . . . 18
  - 2.4.1 Overview of Similarity-based Matching . . . . . 19
  - 2.4.2 Implementation Details . . . . . 22
  - 2.4.3 Complexity Analysis . . . . . 25
- 2.5 Empirical Evaluation . . . . . 26
  - 2.5.1 Input Web Document Dataset . . . . . 26
  - 2.5.2 Baseline algorithms . . . . . 29
  - 2.5.3 Experimental Results . . . . . 29

2.6 Threats to Validity . . . . . . . . . . 34
2.7 Conclusion & Perspectives . . . . . . . . . . 34

**3 Erratum** **36**
3.1 Introduction . . . . . . . . . . 37
3.2 Background & Related Work . . . . . . . . . . 39
3.2.1 Introducing Web Element Locators . . . . . . . . . . 39
3.2.2 Generating Web Element Locators . . . . . . . . . . 40
3.2.3 Repairing Web Element Locators . . . . . . . . . . 41
3.3 Locator Problem Statement . . . . . . . . . . 42
3.3.1 Problem Notations . . . . . . . . . . 42
3.3.2 Problem Statement . . . . . . . . . . 43
3.4 Repairing Locators with Erratum . . . . . . . . . . 44
3.4.1 Applying Tree Matching to Locator Repair . . . . . . . . . . 44
3.4.2 Integrating a Scalable Tree Matching Algorithm . . . . . . . . . . 46
3.4.3 Matching DOM Trees by Similarity . . . . . . . . . . 47
3.5 The Robust Locator Benchmark . . . . . . . . . . 51
3.5.1 Evaluated Locator Repair Solutions . . . . . . . . . . 51
3.5.2 Versioned Web Pages Datasets . . . . . . . . . . 52
3.5.3 Evaluating of the Matched Elements . . . . . . . . . . 56
3.6 Empirical Evaluation . . . . . . . . . . 58
3.6.1 Evaluation of Repair Accuracy . . . . . . . . . . 58
3.6.2 Mutations in the Wayback Dataset . . . . . . . . . . 63
3.6.3 Repair Time Evaluation . . . . . . . . . . 66
3.6.4 Threats to Validity . . . . . . . . . . 68
3.7 Applying Erratum . . . . . . . . . . 68
3.8 Conclusion . . . . . . . . . . 69

**4 Integrating Erratum into Cerberus** **71**
4.1 Introduction . . . . . . . . . . 72
4.2 Background . . . . . . . . . . 73
4.2.1 Web Element Locators . . . . . . . . . . 73
4.2.2 Web Locator Terminology . . . . . . . . . . 73
4.3 Repairing web locators with Erratum . . . . . . . . . . 74
4.4 Building test cases with Cerberus . . . . . . . . . . 75
4.5 Integrating Erratum into Cerberus . . . . . . . . . . 77
4.5.1 Preliminary Demonstration of Erratum . . . . . . . . . . 77

4.5.2 Integration Strategies in CERBERUS . . . 78
4.5.3 The ERRATUM Robust Locators . . . 79
4.5.4 Usage . . . 80
4.6 Industrial Impact . . . 80
4.7 Perspectives . . . 82
4.8 Conclusion . . . 83

**5 APPSTRACT** **84**
5.1 Introduction . . . 84
5.2 Background & Related Works . . . 86
5.2.1 Data Extraction . . . 86
5.2.2 Web Testing . . . 88
5.2.3 Web Analytics . . . 90
5.3 APPSTRACT . . . 90
5.3.1 Abstracting a Web Application . . . 90
5.3.2 Building an Abstraction . . . 92
5.3.3 Intra-Page Abstraction . . . 94
5.3.4 Inter-Page Abstraction . . . 103
5.4 Limits . . . 104
5.5 Evaluation . . . 104
5.5.1 Experiment . . . 104
5.6 Conclusion . . . 109

**6 Conclusion** **111**

**Bibliography** **124**

# List of Tables


3.1 Description of the MUTATION & WAYBACK datasets. . . . . . . . . . . 53
3.2 Mutations applied in the MUTATION dataset 2. . . . . . . . . . . . . 54
3.3 Errors distribution of ERRATUM and WATER on the MUTATION dataset. 59
3.4 Confusion matrix on the WAYBACK dataset. . . . . . . . . . . . . . . 62
3.5 Accuracy summary across datasets. . . . . . . . . . . . . . . . . . . . . 63

4.1 Results of the third testing phase that lasted one month. . . . . . . . 81

# List of Figures


2.1 Example of matching biased by the TED (as computed by APTED). 11

2.2 Building a bipartite graph $G$ representing the set of all possible matchings (left) and then computing the optimal full matching (right). . . . 15

2.3 Steps to compute a full matching between two trees $T$ and $T'$. The upper part covers FTM, while the lower part is SFTM. . . . . . . . . 15

2.4 Creating the bipartite graph $G$ from two example DOMs $T, T'$. (1a,b) are the input DOMs, (2a,b) the extracted tokens, (3) the inverted index $TM$, (4) the neighbors' dictionaries, and (5) the resulting bipartite graph $G$. For simplicity, the figure shows a matching where $IDF(tk) = 1$, $P = 0$, and no-match nodes are not displayed. . . . . . 22

2.5 Distribution of DOM sizes (in terms of nodes) in the dataset. . . . . . 28

2.6 Precision, Recall, and F1 Score of SFTM, APTED, and XyDiff. . . 31

2.7 Computation times when matching trees of different sizes. . . . . . . 32

2.8 Matching efficiency of SFTM, APTED, and XyDiff. . . . . . . . . . 33

2.9 Performance of SFTM given $f(N) = N^{\alpha}$ according to $\alpha$. . . . . . . . 33

3.1 Illustration of the locator problem statement in automated tests combining the *robust locator* (in blue) and the *locator repair* (in green) problems. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.2 State-of-the-art Vs. tree matching locator repair. . . . . . . . . . . . 45

3.3 Two versions of an HTML snippet extracted from the homepage of *duckduckgo.com*. . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

3.4 Key steps followed by our *Similarity-based Flexible Tree Matching* (SFTM) algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

3.5 Distribution of DOM sizes (in number of nodes) in the Mutation dataset. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

3.6 Distribution of DOM sizes (in number of nodes) in the Wayback dataset. 55

3.7 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. . . . . 57
3.8 Accuracy distribution of Erratum and WATER on the Mutation dataset. . . . . 58
3.9 Error percentage according to the mutation type. . . . . 60
3.10 Errors rate evolution according to DOM size. . . . . 61
3.11 Errors rate evolution according to the mutation ratio. . . . . 62
3.12 Analysis of matches labeled as no-match by Erratum. . . . . 63
3.13 Cumulative Distribution of ratios between two versions of web pages. The orange dotted lines show the threshold used in this experiment . 64
3.14 Mutation ratio between two Wayback snapshots depending on gap duration. . . . . 65
3.15 Distribution of mutation labels in the Wayback dataset. . . . . 66
3.16 Repair time evolution according to DOM size. . . . . 67
3.17 Performances of Erratum and WATER. . . . . 67

4.1 A screenshot of the Cerberus web interface to define a test case. . . 76
4.2 The demo application was developed to test Erratum on real use cases before starting integration. In this example, the description of a book hovers on the left-side webpage, which automatically highlights the matched element on the right-side webpage. . . . . 78
4.3 Describing how Erratum is integrated as a robust locator in Cerberus. Both locator types originally locate an element $e \in D$. The figure illustrates the ways Cerberus evaluates XPath or Erratum locators on a new DOM $D'$. . . . . 79

5.1 Screenshots illustrating some elements $\hat{e}_\lambda$ included in the template pages $\hat{p}_{list}$ and $\hat{p}_{product}$ of 2 distinct clusters. . . . . 86
5.2 Overview of the two stages of the *appstraction* process: *learning* and *prediction*. The identifier map of $p$ noted $T_p$ associates an identifier to each element of $p$. The identifiers are first randomly generated, then merged to existing maps during inter-page abstraction. The prediction phase produces an identifier map $T_{\hat{p}}$ in which each original element of $p$ is associated with an application-wide identifier. . . . . 92

5.3 Illustration of Intra-Page abstraction. DOM leaves at the end of repeating branches are tagged and then recursively merged. . . . . . . . 94
5.4 Web page example to illustrate intra-page abstraction with no nested records . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.5 Output of the node-tagging algorithm. Each node is assigned a *tags* map keeping track of the number of leaf groups in its offspring. . . . . 97
5.6 Illustrating two important cases our merging algorithm must cover: *nested records* and *optional elements*. . . . . . . . . . . . . . . . . . . . 98
5.7 Key functions involved in the *abstractTree* algorithm. A directed arrow from function $f$ to $g$ indicates that $f$ calls $g$. . . . . . . . . . . . . . . 102
5.8 An example application of the *abstractTree* function. For each figure, the stack of the current step is shown below. . . . . . . . . . . . . . . 102

# Chapter 1

# Introduction

## 1.1 Motivation

Nowadays, software bugs are an unavoidable concern along all the stages of a software development process. From specification to development and production, software requires to be continuously tested and monitored to detect potential symptoms of dysfunction. Overall maintenance activities account for over two-thirds of the life-cycle cost of a software system, summing up to a total of $70 billion per year in the United States [70].

To support the process of fixing or preventing bugs, a considerable amount of research focuses on static analysis, software testing, or formal methods to help to detect bugs before deployment and bug localization or triaging to help to deal with identified bugs after deployment. However, less attention has been focused on the detection of bugs in deployed systems. In some cases, detecting a bug is indeed trivial, for example, when the application crashes, it usually produces an exception that should be visible in the logs. However, some bugs (like functional bugs) do not produce any exception and are thus far harder to detect using server logs. In 2006, Zhenmin Li *et al.* [54] studied over 29K bug reports and showed that more than 75 % were functional bugs. How to detect such bugs? Surprisingly, this topic has been very sparsely explored given its importance.

How comes the detection of functional bugs gets so little attention then? We think this is due to the perception of bugs within the software engineering community. Analyzing different papers that focus on software defects, we often find classes of bugs based on the type of the bug (multi-threading, arithmetic, logic…) or on the way it manifests (Heisenbug, Bohrbug…). These classifications denote a **system-centric** perception of bugs. In a system-centric paradigm, a bug is a divergence from the

specifications. This definition does not include nor needs to include the end-user of the application. Users may experience a bug, but the bug always preexisted this experience. **In our work, I tried to shift toward a more user-centric definition of a software defect.**

The motivation for our work can be illustrated with one simple idea: a human observing a user navigating a web application can easily understand various information about the user's state and intent. Is the user focused on a single task or browsing without any clear purpose? Is she frustrated? How effectively is the user achieving various tasks on the application? Automatically inferring such information requires a very deep understanding of the application under study. Several existing works studied human behavior in controlled experiments where the application is well known. In our work, we wonder if a systematic analysis of interactions can be applied to any web application without human supervision—i.e. with no manual description of the application under study.

Building tools generic enough to apply to any application poses a certain amount of challenges. As an example, let us try to analyze the behavior of a user on an e-commerce web application. The navigation of the user produces a sequence of actions. As a human, if I see a replay of this sequence, I understand that the user considered several items before buying one. On the other hand, for a machine, analyzing such a sequence appears more challenging. Firstly, how do we define an "action"? Let us only consider clicks, then an action is a tuple $\langle locator, time \rangle$ where the *locator* describes the location of the click (e.g. link, button) and *time* is the timestamp when the click occurred. Then, the next question is: how do we define such a *locator*? A naive approach would be to use the absolute path of the element that was clicked on, but again, this is not a viable option: what if the website changes? What if the web pages are slightly different depending on the country or item category (e.g. electronics, books)? The problem goes even further. When clicking on two different items from a list, a naively constructed *locator* would be different for the two links even though, from a human point of view, both actions have a semantically identical value: the user clicked on an item.

The locator problem [28] is a very profound one. It directly relates to our ability to create an abstract model of an application from a huge amount of elements and pages in much the same way as computer vision requires an underlying understanding of the world to make sense out of millions of pixels. This abstract model can be very subtle and involve visual or contextual clues, it can rely on our experience with a given application or different similar web applications (UI/UX conventions).

Surprisingly, while the locator generation problem has been extensively studied in the context of web application testing, we have never seen it formulated as an instance of the more general web application abstraction inference problem. Concretely, it means existing work focus on building *robust* locators (i.e. locators that do not break when the page changes). While robustness is a fundamental property of a locator, we seek to build locators that are not only robust but also carry semantic information about the nature of the element with respects to other elements in the application. We say that such locators are *semantically rich*.

**The overall objective of our work is thus to provide a generic approach to automatically build robust and semantically rich locators.** Such locators can constitute a powerful web page abstraction which is the cornerstone of many kinds of web page interaction analysis.

Since the locator objective as formulated above is ambitious, in our work, we tried to make progress on the locator problem and the more general web page abstraction problem while applying each iterative progress to improve state-of-the-art on real-life solutions to bug prevention and detection (in particular testing).

## 1.2 Objectives

My objective is thus to explore the idea of general web application abstraction. To do so, we explore three different ideas:

**RQ.1** Tree Matching: how to compare two web pages?

**RQ.2** Robust Locators: how to use tree matching to build robust locators on web pages?

**RQ.3** Appstract: how to leverage tree matching and intra-page abstraction to infer a whole web application abstraction?

### 1.2.1 Tree Matching

One of the core components of humans' ability to create abstract models of the world is the ability to compare. For example, recent significant progress in image generation was obtained by training a neural network to discriminate between real and synthetic images. As humans, one of the elements that enable us to build an abstraction of a web page is our ability to compare several parts of one web page and detect patterns (e.g. list of items), compare several pages from a given application and

locate invariant elements (e.g. menus, logo) compare several pages from a given page template (e.g. product page or news page) and extract what they have in common (e.g. a buy button, a share icon), and even compare several different web applications to understand common patterns and conventions.

The internal representation of a web page is the Document Object Model (DOM). The DOM is a tree of elements that the browser is capable to render into a web page. Comparing two web pages $D$ and $D'$ thus means being able to match every element in $D$ with its corresponding element in $D'$ (if there is one). Such a comparison is called *matching*. Thus the first research question that I propose to investigate is:

**RQ.1.1:** What solutions can be used to efficiently produce a matching between two web page DOMs?

Modern web pages can contain several thousand elements and certain applications contain hundreds of thousands of pages. It means that the efficiency (i.e memory and time complexity) of the studied solutions is crucial to be able to apply such a solution to real-life web applications.

**RQ.1.2:** How to benchmark the accuracy of a given matching solution?

When considering a matching between two web pages, asserting the quality of this matching manually can be hard or even impossible for large web pages containing thousands of elements, and to our knowledge, there is no existing large-scale benchmark for web page matching.

### 1.2.2 Robust and Semantically Rich Locators

Our final goal is to build robust, semantically rich, and application-wide locators.

We first focus on the robustness of our locators. Several works studying the robustness of locators exist.

**RQ.2.1:** How to apply DOM matching solutions to improve the robustness of locators? How does such an approach compare to existing robust-locators solutions?

### 1.2.3 Web Page Abstraction

The last part of our objective is to study how to leverage tree-matching to infer the abstraction of any application as a whole in the form of a set of robust and semantically rich locators.

Such locators must allow the identification of several instances of an element as semantically equivalent (e.g. the price of each item in a list) throughout the whole web application.

Our ability to generate such locators depends on our ability to separate the content from the container on a given application. For example, on an item list, each item has a different price (content), but the same container is instantiated several times.

We formulate the problem as a template extraction one. In particular, we separate this objective into two parts:

**RQ.3.1:** How to infer a template that can be used to generate a given web page?

Such an inferred template is optimal when it abstracts away a maximum of the web page variability while remaining as simple (i.e. few optional nodes) as possible.

Secondly, we wonder how to generalize our results on one page to the full application:

**RQ.3.2:** How to infer the set of templates that generates a given sample of web pages from an application?

Once we have a set of templates that generates a given sample of web pages, we can easily match any web page against this set of templates to generate locators.

## 1.3 Contributions

To achieve the aforementioned objectives, we made four distinct contributions.

1. SFTM: an algorithm allowing to match DOM trees several orders of magnitude faster and more accurately than generic tree matching algorithms,
2. Erratum: a solution that leverages tree-matching to improve the robustness of locators,
3. Integration of Erratum to an open-source testing framework and empirical analysis of its impact in partnership with a major online retailer,
4. Appstract: a solution allowing to infer a set of robust and semantically rich application-wide locators with almost no human supervision.

### 1.3.1 SFTM

A critical part of our objective relies on our ability to compare two web pages. Unfortunately, none of the state-of-the-art solutions we managed to test allowed us to produce an accurate matching between real-life web pages (containing up to several thousands of nodes) within acceptable time constraints (in the order of milliseconds).

That is why we developed SFTM. SFTM is a tree-matching solution specializing in web pages. It takes advantage of the fact that web page DOM nodes naturally contain very rich labels (e.g. element tags and attributes). Our algorithm thus takes a different approach from traditional tree-matching algorithms by using statistical tools to, first, match labels before taking into account the topology of the trees.

We also developed the first large-scale synthetic benchmark to compare SFTM to other state-of-the-art solutions. Our benchmark shows significant gains in terms of both computation times and accuracy.

We describe SFTM in Chapter 2

### 1.3.2 Erratum

While we believe tree-matching is a crucial step towards our objective, the SFTM contribution does not describe how to use tree-matching to progress towards the inference of web application abstraction through locator generation.

For this reason, we developed Erratum, a solution that leverages tree-matching to repair broken locators. Broken locators are the main reason web page test scripts break along the development of a web application. As an example let us consider a test scenario for an e-commerce website consisting of three actions:

1. On a product list page, click on a product title,
2. Once on the product page, click on the buy button,
3. Ensure we get to the checkout page.

This scenario contains a minimum of two locators locating the product title and the buy button. These locators will function as long as the page remains strictly the same but any change on the page could break them. In this case, the script will have to be updated. In practice, broken locators can be a major pain for testing teams.

Several approaches to repairing broken locators exist. These approaches all take as input a locator on an element $e$ on a page $D$ and try to find this element on another version of the page $D'$. Unlike existing approaches, Erratum uses a holistic approach to locator repair: instead of trying to repair one isolated locator, it matches all nodes between $D$ and $D'$ and uses the produced matching to fix any broken locators.

We then developed a benchmark for Erratum combining synthetic and real-life mutations to compare its performance with existing locator repair approaches. Results show the Erratum outperforms other locator repair solutions with little to no overhead depending on the number of locators to repair.

We describe ERRATUM in Chapter 3

### 1.3.3 ERRATUM Industrial Case Study

Following our work on ERRATUM, we were approached by the engineering department of a major online french retailer interested in the potential benefits of ERRATUM. Indeed, the company was struggling a lot with the broken locator problem causing them to spend a considerable amount of resources on fixing past testing campaigns. We thus released SFTM as an open-source maven library and followed ERRATUM's approach to integrating it into CERBERUS, the open-source testing framework used by the company.

We then followed up with the company to track the usage and impact of ERRATUM on their workflow. While executing rigorous and meaningful empirical experiences in this industrial context proved challenging, we found out that ERRATUM worked flawlessly as a replacement for manually written locators.

We describe the industrial application of ERRATUM in Chapter 4

### 1.3.4 APPSTRACT

A consequent body of research deals with the study of web applications, in particular in the domains of testing, web analytics, and information extraction. A large part of the work on web applications in these domains must resort to innovative heuristics to make sense of the thousands of pages and nodes in modern applications. While we originally tackled this idea from the analytics point of view—i.e. creating some kind of rich encoding for user actions—this idea is more general. I strongly believe most of the aforementioned research domains linked with the study of web applications would highly benefit from having a unified objective: unsupervised inference of an abstract model for any real-life web application. To my knowledge, such an objective has never been formulated as such in the literature. Thus, we need to note that beyond the actual algorithms, the first contribution of APPSTRACT is to make a step in this direction.

APPSTRACT is an approach to inferring an abstract model from a web application. Concretely, given a set of web pages from a single web application, APPSTRACT is capable to infer an abstract model. Once the model is built, APPSTRACT allows the assignment of an identifier to any node of a provided (previously unseen) page from this application. The assigned identifier is both robust and semantically rich—i.e. two buy buttons from different products will have the same identifier.

Our approach has two main components:

1. **Intra-page** abstraction: detect patterns within one page,
2. **Inter-page** abstraction: detect patterns between two pages.

APPSTRACT uses these components at several stages for both the creation of the abstraction model and its use.

Finally, we designed an experiment to evaluate the quality of APPSTRACT.

We describe APPSTRACT in Chapter 5

# Chapter 2

# Tree Matching

## Summary

Tree matching techniques have been investigated in many fields, including web data mining and extraction, as a key component to analyze the content of web pages. However, when applied to existing web pages, traditional tree matching approaches, covered by algorithms like *Tree-Edit Distance* (TED) or *XyDiff*, either fail to scale beyond a few hundred nodes or exhibit a relatively low accuracy.

In this section, we, therefore, propose a novel algorithm, named *Similarity-based Flexible Tree Matching* (SFTM), which enables high accuracy tree matching on real-life web pages, with practical computation times. We approach tree matching as an optimization problem and leverage node labels and local topology similarity to avoid any combinatorial explosion. Our practical evaluation demonstrates that SFTM significantly improves the state of the art in terms of accuracy while allowing computation times significantly lower than the most accurate solutions. By gaining on these two dimensions, SFTM, therefore, offers an affordable solution to match complex trees in practice.

## 2.1 Introduction

The success of the Internet has led to the publication and delivery of a deluge of structured content. Nowadays, web services and applications are heavily adopting tree-based documents to structure and transfer online content. However, these web pages keep evolving, and keeping track of such changes remains a critical issue for the ecosystem and the research community. Examples of usages that require to detect or

track changes in web pages include web extraction [71, 89, 91], web testing [15, 76], comparison of web service versions [22], web schema matching [30], and automatic re-organization of websites [42].

To date, few solutions are specifically designed or tested to match and compare two web pages. However, the more general question of tree matching has been extensively studied by two families of solutions applicable to the problem of web page matching: 1. *Tree Edit Distance* (TED) [79] and TED-related solutions, and 2. XML differentiation (diff) solutions.

TED is the first and most widely known approach to match trees. The matchings computed by TED solutions are optimal and there has been much effort into developing openly available efficient implementations of the algorithm [67, 68, 69]. Despite these efforts, TED remains costly to compute. A recent study [9] theoretically showed that no algorithm could compute the optimal TED in less than $O(N^3)$ worst time complexity. To address TED's limitations, several restrictions to TED have been developed. These TED-related algorithms add constraints to the produced matching allowing to trade accuracy for speed.

XML diff solutions aim to find the sequence of editions between two XML trees. The approach is similar to TED, but solutions sometimes make use of XML specificities. For example, the most widely-known XML diff solution—XyDiff [16]—is extremely fast, but makes heavy use of XSD schemas and XML primary keys, which cannot be assumed on any web page. Without such additional information, the algorithm unfortunately yields low-accuracy results.

Overall, when matching two web pages, even the most efficient TED implementation [69] offers far from optimal accuracy (69% of precision on average in our empirical evaluation) for computation times often reaching several seconds. The lack of accuracy may be due to the restrictions TED solutions impose on the produced matching: ancestors' and siblings' orders must be preserved. However, such restrictions do not hold for web pages and Figure 2.1 illustrates how TED can report biased matchings, even on simple trees.

To address these restrictions when attempting to match two web documents, [22] extended TED with some additional move operations executed *a posteriori* to address the ancestry restriction and [43, 42] developed her own *Flexible Tree Matching* (FTM) algorithm to address the ancestry restriction problem. Unfortunately, while FTM provides a truly restriction-free matching, its high complexity does not allow FTM to scale beyond more than a few dozen of nodes, which is far below the average size of real-life web pages.

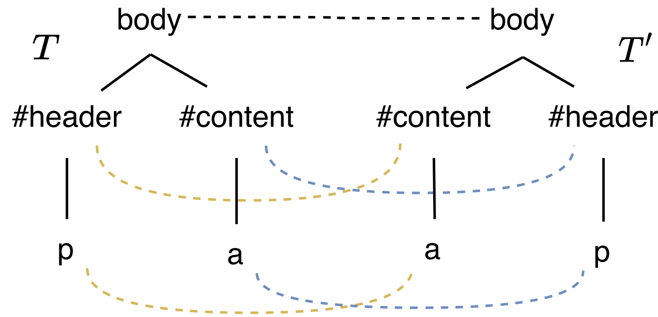



Figure 2.1: Example of matching biased by the TED (as computed by APTED).

In line with the aforementioned work, this chapter, therefore, aims at enabling the fast and non-restricted comparison of complex web pages. In particular, we propose an alternative to the state-of-the-art FTM algorithm, named *Similarity-based Flexible Tree Matching* (SFTM), that leverages similarity metrics to speed up the comparison of complex trees. SFTM shares the properties of FTM to offer a non-restricted tree matching, while offering computation times much lower than FTM, even on restricted versions of the problem. To match two web page trees, the approach taken by SFTM strongly differs from traditional techniques. In particular, existing matching algorithms are *structure-centric*: they leverage the structure of both trees to select the nodes to visit and compare. SFTM instead relies on a *label-centric* approach: it prunes the space of possible matchings using nodes' *label* and considers the tree topology *a posteriori* to propagate information contained in the nodes.

We compared SFTM to other state-of-the-art solutions on a large dataset of popular web pages. SFTM showed almost twice more efficiency as the best existing solution. Overall, our algorithm SFTM allows us to consistently match real-life web pages with high precision (89% precision on average) in a reasonable time (182 ms on average).

The code for both SFTM and its benchmark is available openly.[1]

The remainder of this chapter is organized as follows. Section 2.2 and 2.3 cover related work, with section 2.3 focusing in details on the *Flexible Tree Matching* (FTM) original algorithm. Section 2.4 presents *Similarity-based Flexible Tree Matching* (SFTM), our extension of FTM that leverages the node labels and local topology similarity to guide the comparison. Section 2.5 thoroughly evaluates our solution against the state of the art on a realistic dataset of web documents. Section 2.6 discusses the threats to validity of our contribution. Section 2.7 concludes and overviews

[1] https://anonymous.4open.science/r/7ae57bd7-3b29-463a-88a4-d31c04ecfcd2/

some perspectives for this work.

## 2.2 Related Work

***Tree Edit Distance* (TED)** Comparing two trees is a problem that has been at the center of a significant amount of research. In 1979, Tai [79] introduced the *Tree Edit Distance* (TED) as a generalization of the standard *edit distance* problem applied to strings. Given two ordered labeled trees $T$ and $T'$, the TED is defined as the minimal amount of node insertion, removal or relabel to transform $T$ into $T'$, while different cost coefficients can be assigned to each type of operation. By following an optimal sequence of operations applied to $T$, it is possible to match the nodes between $T$ and $T'$. This problem has been extensively studied since then to reduce the space and time complexity of the algorithm that computes the TED. To the best of our knowledge, the reference implementation available today is the *All-Path Tree Edit Distance* (APTED) [67, 68, 69] with a complexity of $O(n^2)$ in space and $O(n^3)$ in time in the worst case, where $n$ is the total number of nodes ($n = |T_1| + |T_2|$). In our work, we consider APTED as one of the baselines to evaluate our contribution.

[9] showed that TED cannot be computed in worst-case complexity lower than $O(n^3)$. To circumvent this limitation, several restricted versions of the TED problem have been formulated. The *Constrained Edit Distance* [93, 94] is an edit distance where disjoint subtrees can only be mapped to disjoint subtrees. The *Tree Alignment Distance* [36] is a TED where all insertions must be performed before any deletion. The *Top-Down* distance [74] is computable in $O(|T| \times |T'|)$, but imposes as a restriction that the parents of nodes in a mapping must be in the mapping. The *Bottom-Up* distance [81] between trees builds a mapping in linear time, but such mapping must respect the following constraint: if two nodes have been mapped, their respective children must also be part of the mapping. [71] proposes a variation of the *Top-Down* mapping, called *Restricted Top-Down Mapping* (RTDM), where replacement operations are restricted to the leaves of the trees, which delivers considerable speed gains, despite a theoretical worst-case time complexity still in $O(N^2)$. By definition TED already sets strong restrictions on produced matchings: sibling order and ancestry relationships must be preserved [93]. These restrictions are particularly problematic when matching two full web pages together [42]. While the above solutions improve computation times, they answer a restricted version of the TED problem leading to an even more restricted set of possible matchings.

**XML diff** While TED-related approaches focus on computing a distance between trees, another part of the scientific literature focuses on inferring the set of edit operations between two XML documents. Most XML diff solutions use an intermediary matching step to compute the diff. Computing the set of diff from a given matching is quite straightforward, which means that most works on the subject focus on the matching part. XyDiff [16] matches and computes the diff of two XML documents very quickly. To do so, XyDiff hashes subtrees from both documents and prunes the space of matching possibilities by matching subtrees with identical hashes. The algorithm can also make use of id attributes and XSD schemas if they exist. On the other end of the spectrum, X-Diff [85] favors accuracy over speed and computes an optimal matching using hashing of path signatures. XKeyDiff [21] builds on XyDiff and adds matching logic based on XML primary keys, `XML_SIM_CHANGE` [82] and `XREL_CHANGE_SQL` [78] match XMLs stored in relational databases using SQL. Phoenix [65] interestingly uses a more flexible similarity metric between nodes (*e.g.*, to compare the content of two nodes, they use the *Longest Common Sequence*) and choose how to match each subtree by recursively applying the Hungarian algorithm [41]. Unfortunately, Phoenix runs in $O(n^4)$ and yields less accurate results than X-Diff. In our empirical evaluation, we evaluated our solution along with XyDiff, which is widely known and used for XML diff, has an efficient implementation openly available, and runs in scalable computation times.

***Flexible Tree Matching* (FTM)** In [42], TED is found to be unpractical when applied on DOM, as the resulting matching enforces ancestry relationship—*i.e.*, once two nodes $n, n'$ have been matched, the descendants of $n$ can only be matched with the descendants of $n'$, and *vice versa*. Consequently, Kumar *et al.* [42, 43] introduced the notion of *Flexible Tree Matching* (FTM), which relaxes the ancestry relationship constraint at the price of a stronger complexity. It restricts its use to small HTML trees composed of less than a hundred nodes, thus making it unpractical for modern web documents, often including more than a thousand nodes. Furthermore, to the best of our knowledge, there is no public implementation of FTM that can be considered for comparison.

We, therefore, aim at reducing the complexity of the FTM algorithm to scale on complex web pages without enforcing restrictions on produced tree-matching solutions. While all the above contributions are structure-based, we build on FTM's approach and rather offer a flexible, label-based matching where labels are used to match nodes and structure is only used *a posteriori* to improve the matching.

Our contributions, therefore, read as follows:

1. we develop an algorithm inspired by FTM, coined as *Similarity-based Flexible Tree Matching* (SFTM), by leveraging the notion of label similarity, and similarity propagation to reduce the computation time, and
2. we apply mutations on real-life web documents to provide a thorough evaluation of our implementation of SFTM, showing that it outperforms state-of-the-art approaches in terms of efficiency.

## 2.3 Flexible Tree Matching (FTM)

The *Similarity-based Flexible Tree Matching* (SFTM) we introduce in this chapter can be considered as an extension of the *Flexible Tree Matching* (FTM) algorithm. This section, therefore, introduces the FTM algorithm, as originally proposed by Kumar *et al.* [42]. We first describe the notations used throughout the rest of the chapter and then describe the main steps of the algorithm.

### 2.3.1 FTM Notations and Overview

We define an ordered tree $T$ as a directed graph $(N, \prec)$ where $N$ is the non-empty set of nodes and $\prec$ a total order relation that can relate a *child* node $c \in N$ to its *parent* $p \in N$, as $c \prec p$, or *siblings*, as $s \in N$, as $c \prec s$.

In particular, we choose a total order rather than a partial one as the order of siblings has a strong semantic value for a webpage (e.g. the order of paragraphs).

In this chapter, we always consider *matchings* between two trees $T = (N, \prec)$ and $T' = (N', \prec')$.

Given two trees $T$ and $T'$, the FTM algorithm relies on the complete bipartite graph $G$ between $N^* = N \cup \Theta$ and $N'^* = N' \cup \Theta'$, where $\Theta$ and $\Theta'$ are *no-match* nodes. The fact that $G$ is complete means that every node of $T^*$ shares exactly one edge with every node of $T'^*$. Formally, we thus have $E(G) = N^* \times N'^*$ where $E(G)$ are the edges of the graph $G$. An edge $e = (n, n') \in E(G)$ between $n \in N^*$ and $n' \in N'^*$ represents the matching of $n$ with $n'$. Each edge linking a tuple $(n, n')$ is called a *match*. So, intuitively, $G$ represents all possible matchings between nodes of $T^*$ and $T'^*$ (cf. Figure 2.2).

Formally, we call *matching* and note $M \subset E(G)$, a subset of edges selected from $G$. A matching $M$ is said to be *full iff* each node in $N$ has exactly one edge in $M$ that links it to a node in $N'^*$ and, inversely, each node in $N'$ has exactly one edge

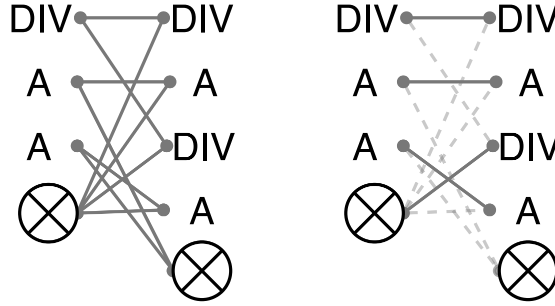



Figure 2.2: Building a bipartite graph $G$ representing the set of all possible matchings (left) and then computing the optimal full matching (right).

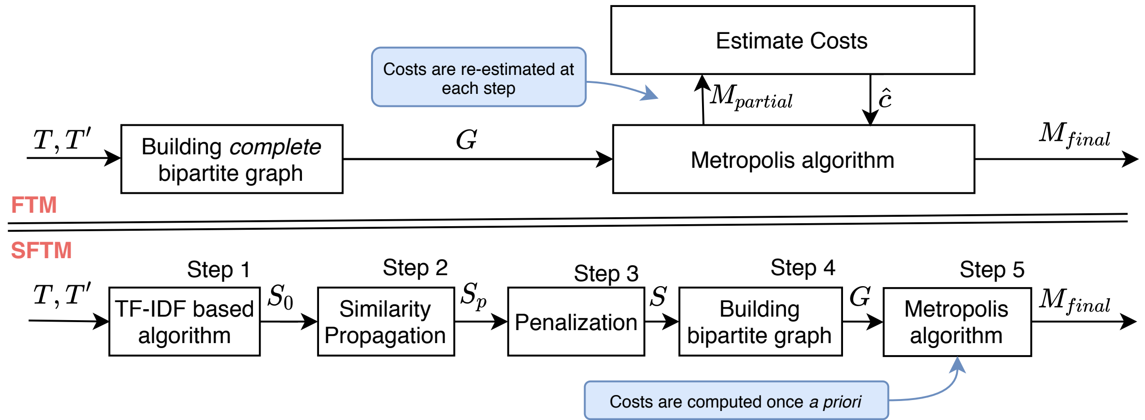



Figure 2.3: Steps to compute a full matching between two trees $T$ and $T'$. The upper part covers FTM, while the lower part is SFTM.

in $M$ that links it to a node in $N^*$. Since matchings need to be *full*, the auxiliary *no-match* nodes $\Theta_1, \Theta_2$ are required to cope with insertion and deletion operations. The set of possible *full* matchings is restricted to the set of matchings satisfying that every node in $N \cup N'$ is covered by exactly one edge. *No-match* nodes are the only nodes allowed to be involved in multiple edges.

Given an edge $e = (n, n') \in E(G)$ linking $n$ to $n'$, FTM defines the cost $c(e)$ to quantify how different $n$ and $n'$ are, considering both their labels and the topology of the tree. Starting from the bipartite graph $G$ describing all possible matchings, the idea behind FTM is to compute the costs $c(e)$ of each edge $e \in E(G)$ and to find the optimal matching with respect to these estimated costs—*i.e.*, to select the set of edges $M \subset E(G)$, such that $M$ is *full* and $c(M)$ is minimal (where $c(M) = \sum_{e \in M} c(e)$).

The upper part of Figure 2.3 describes the main steps involved in computing the final full matching between $T$ and $T'$.

### 2.3.2 Cost Estimation

As FTM provides wide flexibility regarding possible matchings, the design of the cost function $c$ is a key parameter to obtain a matching that takes into account both the labels and the topology of the trees. Typically, the cost $c(e)$ of an edge $e$ between two nodes $n$ and $n'$ is estimated by FTM as follows:

$$c(e) = \begin{cases} w_n & \text{if } n \text{ or } n' \in \{\Theta, \Theta'\} \\ w_r c_r(e) + w_a c_a(e) + w_s c_s(e) & \text{otherwise} \end{cases} \quad (2.1)$$

where $\Theta, \Theta'$ are *no-match* nodes, $w_n$ is the penalty when failing to match one of the edge ends, $c_r(e)$, $c_a(e)$ and $c_s(e)$ are the cost of *relabeling*, *violating ancestry relationship* and *violating sibling group*, respectively, and $w_r$, $w_r$ and $w_r$ their associated weight in the cost function. $w_n, w_r, c_r, w_a$ and $w_s$ are parameters of the cost function that depend on the kind of matching the user requires. By extension, we note $c(M) = \sum_{e \in M} c(e)$ the cost of a matching $M$.

Given $e = (n, n')$, the ancestry and sibling costs, $c_a(e)$ and $c_s(e)$, model the changes in topology that matching $n$ with $n'$ entails. Unfortunately, we can only estimate the costs $c_a$ and $c_s$ if we have access to a full matching, as both costs require knowledge of how other nodes in the tree were matched (*e.g.*, $c_a$ involves counting the number of children of $n$ matched with nodes that are not children of $n'$). To circumvent the problem, FTM rather considers the approximate costs $\hat{c_a}, \hat{c_s}$ that can be estimated from bounds on the different components of the cost $c$. Practically, to generate one possible full matching, FTM iteratively selects edges in $G$ and, each time an edge is selected, the bounds of $c$ are tightened (we can approximate $c$ more precisely), which means that the costs $\hat{c_a}, \hat{c_s}$ keep being re-estimated along iterations (cf. upper part of Figure 2.3). The need to re-estimate the approximated costs after each edge selection imposes some critical limitations on the scalability of the algorithm.

### 2.3.3 Metropolis Algorithm

Finding the optimal matching, given the graph $G$ and the cost function $c$ is a challenging problem, the authors even proved in [43] that this problem is NP-hard. Consequently, the authors described how to use the Metropolis algorithm [58] to approximate the optimal matching. The Metropolis algorithm provides a way to explore a probability distribution by random walking through samples. FTM uses this algorithm to walk randomly through several full matchings, and select the least

costly. The METROPOLIS algorithm requires to be configured with:

1. An initial sample (full matching) $M_0$,
2. A suggestion function (alternative matching) $M_t \mapsto M_{t+1}$,
3. An objective function to maximize: $f : M \mapsto$ quality of $M$,
4. The number of random walks before returning the best value.

Kumar *et al.* defines the objective function $f$ by:

$$f_{FTM}(M) = \exp(-\beta\ c(M)) \tag{2.2}$$

To suggest a matching $M_{t+1}$ from a previously accepted one $M_t$, FTM selects a random number of edges from $M_t$ to keep, sorts remaining edges by increasing costs, and iterates through the ordered edges with a probability $\gamma$ to select it. Once an edge $e = (n, n')$ is selected, all edges connected to $n$ and $n'$ are removed from $G$, and approximate costs need to be re-estimated for all remaining edges, and then sorted so we can select another edge. The process is repeated until an alternative full matching is obtained. Therefore, despite using the METROPOLIS algorithm to reduce the time complexity of the problem, the overall algorithm remains prohibitively costly to compute (cf. Section 2.5), notably due to the continuous re-estimation of the approximated costs at each step of the full matching generation.

### 2.3.4 Complexity Analysis

The original FTM article [43] does not report on the complexity of the computation time of the algorithm. We, therefore, provide an analysis of FTM's theoretical complexity to compare it to the one of SFTM (cf. Section 2.4.3).

When discussing complexity, to simplify the notations, we consider the matching of two trees with the same number of nodes and we note $N$ the number of nodes of both trees.

**Complete bipartite graph $G$** Building the complete bipartite graph requires matching each node from $T$ to one node from $T'$, which requires $O(N^2)$ operations.

**METROPOLIS algorithm** For each iteration of the METROPOLIS algorithm, FTM has to suggest a new matching. In the worst case, the algorithm should choose among all $N^2$ edges. Each time an edge between $e_1$ and $e_2$ is selected, all other edges connected to $e_1$ and $e_2$ are pruned and remaining costs requires to be re-estimated. It means that costs have to be re-estimated and sorted for $N^2$ edges, then $(N-1)^2$

edges (after selection and pruning), and so on, until all edges have been selected or pruned. This implies that the total number of times the costs are re-estimated and sorted is in $O(\sum_{n=0}^{N} n^2) = O(N^3)$. Estimating the cost for a given edge linking $e_1$ and $e_2$ involves counting the number of potential ancestry and sibling violations, which requires going through all edges connected to siblings and children of $e_1$ and $e_2$. Even if we assume the number of siblings and children is independent of $N$, it still means that estimating the cost of one edge requires $O(N)$ operations. Thus, in the worst case, the amount of operations required by FTM for each iteration of the Metropolis algorithm is in $O(\sum_{n=0}^{N} n^3) = O(N^4)$ (using Faulhaber's Formula).

Overall, the Metropolis step is the one with the highest complexity, which means that the complexity of the FTM algorithm is in $O(N^4)$ where $N$ is the number of nodes to match.

## 2.4 Similarity-based FTM (SFTM)

Based on the above complexity analysis, *Similarity-based Flexible Tree Matching* (SFTM) replaces the cost system of FTM by a similarity-based cost that can be computed once *a priori* (cf. Figure 2.3). This approach drastically improves computation times and rather exposes a parameter that can be tuned to find the desired trade-off between computation time and matching accuracy (cf. Section 2.5).

Given two trees $T = (N, \prec)$ and $T' = (N', \prec')$, SFTM relies on the specification of a *similarity metric* between nodes $n \in N$ and $n' \in N'$. We compute this similarity metric for all pairs of nodes $(n, n')$ using *i)* inverted indices for labels and *ii)* label propagation and some penalization heuristics for the topology. We build a bipartite graph $G$ between nodes of $T$ and $T'$ using this similarity metric to compute the costs and apply the Metropolis algorithm to approximate the optimal full matching from $G$. This new similarity measure allows SFTM to improve the FTM algorithm in two key aspects:

1. when building $G$, we do not create all $|N| \times |N'|$ possible edges. We only consider edges linking two nodes with a non-null similarity; and
2. when generating a full matching, costs do not need to be updated as these costs solely depend on our similarity measure.

In this section, we therefore (a) introduce our new similarity metric, and (b) describes how we leverage it to approximate the optimal full matching.

### 2.4.1 Overview of Similarity-based Matching

The similarity metric between nodes $N$ and $N'$ is computed in two main steps: 1. we compute $s_0$, the initial similarity function using only *labels* of the trees individually, and then 2. we transform $s_0$ to take into account the topology of the tree and compute our final similarity function $s$. The computation of $s_0$ leverages inverted index techniques traditionally used to query text in a large document database. In our case, the documents we query against are $N$, while queries are extracted from $N'$. Figure 2.3 illustrates the different steps described in this section.

#### Initial Similarity (step 1)

To compute the initial similarity $s_0$ between $N$ and $N'$ (cf. *step 1* in Figure 2.3), we independently compare the labels of $N$ and $N'$ using the *Term Frequency-Inverse Document Frequency* (TF-IDF). The resulting initial node similarity $s_0$ does not take the topology of the trees into account.

In order to take into account relabeling cost between nodes, some existing solutions (*e.g.*, APTED) allow the user to input a pairwise comparison function $label(n), label(m) \mapsto$ *similarity score*. However, computing this similarity score for all the pairs of nodes requires $O(N^2)$ operations. Thus, to reduce the number of operations, SFTM uses—instead—inverted indices: given a tokenize function $tokenize : n \mapsto$ *token list*, SFTM 1. decomposes each node $n$ from $N$ into a set of tokens (as defined by the *tokenize* function), and then 2. iterates through tokens of nodes $n'$ from $N'$ to increase the value of $s_0(n, n')$ for each token $n$ and $n'$ have in common. Section 2.4.2 provides a detailed description of the function *tokenize* we use in our evaluation.

Decomposing nodes from $N$ into tokens allows SFTM to build an inverted index $TM$ (*Token Map*), which maps every token $tk$ with the list of nodes of $N$ that contains $tk$. The idea behind the inverted index $TM$ is to use the information that a node $n \in N$ contains a token as a differentiating feature of $n$ allowing us to quickly compare it to nodes in $N'$. If a token $tk$ appears in all nodes $N$, this token has no differentiating power. In general, the rarest a token, the more differentiating it is. This idea is very common in *Natural Language Processing* (NLP) and a common tool to measure how rare is a token in TF-IDF [38] and more precisely, the *Inverted Document Frequency* (IDF) part of the formula. Applying TF-IDF to our similarity yields the following

definition:

$$IDF(tk) = log(|N|/|TM[tk]|) \tag{2.3}$$

$$s_0(n, n') = \sum_{tk \in TK} IDF(tk) \tag{2.4}$$

Where $TK = tokens(n) \cap tokens(n')$. The function IDF is a measure of how rare a token is, $|TM[tk]|$ is the number of nodes containing the token $tk$ and $tokens$ refers to the user input tokenize function. Intuitively, we retrieve the tokens shared between nodes $n$ and $n'$ and, for each common token $tk$, we increase $s(n, n')$ by a high value if $tk$ is rare and a low value if $tk$ is common. In Section 2.4.2, we provide a detailed implementation of how to compute the initial similarity $s_0$.

Tokens that appear in many nodes have little impact on the final score—*i.e.*, low IDF—yet have a very negative impact on the computation time. In our algorithm, we expose the sublinear threshold function $f : N \mapsto f(N) < N$ as a parameter of the algorithm. We use $f$ to filter out all the tokens appearing in more than $f(N)$ nodes. Therefore, $f$ provides a balance between computation time and matching quality: when $N - f(N)$ decreases, computation times and matching quality increase. In Section 2.4.3, we discuss how $f(N)$ influences the worst-case theoretical complexity.

### Local Topology (step 2)

$s_0$ represents the similarity between node labels but does not take into account the topology of the trees. To weigh in local topology similarities, we propagate the score of each node pair to their offspring and siblings. This idea of propagation is inspired by recent *Graph Convolutional Network* (GCN) techniques [39].

The original FTM algorithm includes two terms in the cost function, $c_a$ (ancestry cost) and $c_s$ (sibling cost), which reflect the topology of the trees. Since we do not use these terms (as they require too much computation time), we need our similarity score to reflect both the similarity of node labels and the similarity of the local topology. Therefore, we first compute the score matrix $s_0$, based on the label similarity we described above, and then we update this score to take into account the matching score of the parents of $n$ and $n'$. By doing so, $n$ has a higher similarity score with $n'$ if their respective parents or children are also similar.

Beginning at $s_0$, at each step $i$ and for all pairs that have a non-null initial score $\{(n, n') \in N \times N' | s_0 \neq 0\}$, we first compute:

$$s_i(n, n') \leftarrow s_{i-1}(n, n') + w_i \times s_{i-1}(p(n), p(n')) \tag{2.5}$$

where $p(n) \in N$ refers to the parent of node $n$.

Similarly, we then increase the score of the parents of $n, n'$:

$$s_i(p(n), p(m)) \leftarrow s_{i-1}(p(n), p(n')) + v_i \times s_{i-1}(n, n') \tag{2.6}$$

where $w_0, w_1 \dots w_P$ and $v_0, v_1 ... v_P$ are topology weights. We repeat the process $P$ times ($P$ for propagation) where $P$ is a parameter of SFTM. The resulting function $s_P$ then reflects both label similarity and local topology similarity.

Intuitively, at each iteration, we propagate information further up in the tree. This is why the weight sequences $w$ and $v$ should be decreasing so that close kinship among nodes prevails. From our experiments, we advice the following values for the $P = 3$ weights: $w_0 = 0.4, w_1 = 0.04, w_2 = 0.004$ and $w_0 = 0.8, w_1 = 0.08, w_2 = 0.008$. These values were used and unchanged for all results presented in the empirical evaluation 2.5, leading to high accuracy on a large variety of web documents.

### Penalization (step 3)

There are two main drawbacks to the way we propagate the scores in step 2: 1. the scores are still almost exclusively based on labels, 2. nodes with many children may get an unfair score boost from the propagation.

While (2) can be fixed by normalizing the propagation according to the number of children, the normalization would also potentially remove valuable information. Instead, for each pair $(n, n')$, we rather apply a penalization proportional to the difference between the number of children of $n$ and $n'$:

$$s(n, n') = s_P \times (1 - penalty(n, n')) \tag{2.7}$$

where $penalty(n, n') \mapsto [0, 1]$ is the children penalization defined by:

$$penalty(n, n') = \frac{|(|children(n)| - |children(n')|)|}{max(|children(n)|, |children(n')|)} \tag{2.8}$$

where $|ch(n)|$ is the number of children nodes of $n$. This step yields the final score function $s$, defined for each couple $(n, n')$.

### Building the bipartite graph $G$ (step 4)

Using our final score function $s$, we can now build the bipartite graph $G$: we iterate over all nodes $n \in N$ and we create an edge $e = (n, n')$ for each pair of nodes such

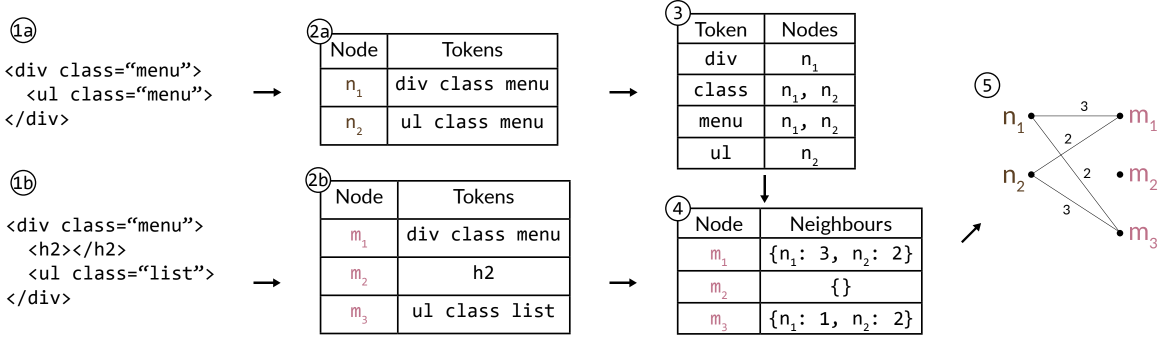



Figure 2.4: Creating the bipartite graph $G$ from two example DOMs $T, T'$. (1a,b) are the input DOMs, (2a,b) the extracted tokens, (3) the inverted index $TM$, (4) the neighbors' dictionaries, and (5) the resulting bipartite graph $G$. For simplicity, the figure shows a matching where $IDF(tk) = 1$, $P = 0$, and no-match nodes are not displayed.

that $s_P(n, n') \neq 0$ and associate it with the cost $c(n, n') = 1/(1 + s_P(n, n'))$. Our resulting cost function is thus defined as follows:

$$c_{SFTM}(e) = \begin{cases} w_n, & \text{if } n \text{ or } n' \text{ is a no-match node} \\ \frac{1}{1+s_p(n,n')}, & \text{otherwise} \end{cases} \tag{2.9}$$

Importantly, unlike the bipartite graph built in the FTM algorithm, the resulting bipartite graph $G_{SFTM}$ is *not complete* as only edges, such that $s_p(n, n') \neq 0$ are considered. This is one of the key differences allowing SFTM to drastically improve computation times.

## 2.4.2 Implementation Details

In the previous section, we introduced the SFTM algorithm and described how it compares to FTM. In this section, we describe more precisely how we implement the different steps of SFTM.

### Node Similarity (step 1, 2 and 3)

Let us consider two trees $T$ and $T'$. We first build the dictionary $TM$, an inverted index—*i.e.*, each entry of $TM$ is a tuple $(token, nodes)$ where *token* is a token (usually a string) and *nodes* is a *set* of all $n \in N$ that contains *token*. Figure 2.4 (2a,b) depicts two examples of inverted index. We note $TMmap[key]$ the set of *nodes* whose key in $TM$ is *key*. In Section 2.4.2, we further describe how we sort HTML nodes into tokens.

Given the inverted index $TM$, we define the function $IDF : tk \mapsto log(|N|/|TM[tk]|)$. To limit the complexity of our algorithm, we remove every token $tk \in TM$ that is contained by more than $f(N) = \sqrt{N}$ nodes, where $f$ is the chosen sub-linear threshold function. This is equivalent to putting a threshold on $IDF$ to only keep tokens $\{tk \in TM | IDF(tk) > log(\sqrt{N})\}$. Removing the most common tokens has a limited impact on matching quality since these are exactly the tokens that provide the least information on the nodes they appear in.

**Algorithm 1** For a given node $n' \in N'$, compute similarity score $s_0(n, n')$ with all $n \in N$, such that $s_0 > 0$

**Inputs** $n'$: a node in $N$, $TM$: token map, dictionary of nodes from $T$ per token
**Output** $neighbors$: a dictionary of scores per node in $T$

```
neighbors ← new Dictionary()
for all tk in tokens(n') do
    for all n in TM[tk] do
        neighbors[n]+ = IDF(tk)
    end for
end for
return neighbors
```

Once we have the token index $TM$ and the function $IDF$, we apply Algorithm 1 on each node $n' \in N'$. In Algorithm 1, we first compute the tokens of node $n'$ and, for each token $tk$, we use $TM$ to retrieve the nodes $n \in N$ that contain the token $tk$. Each node $n$ thus retrieved is considered as a *neighbor* of $n'$—*i.e.*, $s_0(n, n') \neq 0$. Finally, for each neighbour $n$ of $n'$, we add $IDF(tk)$ to the current score $s_0(n, n')$. At this point, we have a $neighbors(n')$ dictionary for each node $n' \in N'$. Each $neighbors(n')$ dictionary contains all non-null matching scores: $\forall n \in keys(neighbors(n')), neighbors(n')[n] = s_0(n, n')$. Using the Equation 2.5, we can now easily compute $s_p$ and $s$.

### Building the Token Vector

The actual labels are never directly used by SFTM. The algorithm only leverages the tokens extracted from these labels. The way we choose to extract the tokens contained in a node $n$ thus strongly influences the quality of our similarity score. We implemented the following function *tokens* to report all the tokens of a node $n$. Given $n$, an HTML node representing a `tag`:

```
<tag att_1="val_1" ... att_2="val_2">
  CONTENT
```

`</tag>`

where $l$ is the number of attributes, $(att_i, val_i), i \in [1, l]$ are the attribute/value pairs of $n$ and the absolute XPath of $n$ is $xPath(n)$. We decompose $n$ into the following tokens:

$$tokens(n) = \{xPath(n), \texttt{tag}, att_1..a_l, tok(val_1)..tok(val_l)\} \tag{2.10}$$

where $tok$ is a standard string tokenizing function that takes a string and splits it into a list of tokens on each non-Latin character. The absolute XPath of a node $n$ in a tree is the full path from the root to the element where ranks of the nodes are indicated when necessary—*e.g.*, `html/body/div[2]/p`.

SFTM does not include the text content of the nodes in the extracted token vectors. This decision allows to match pages in different languages or containing different content (e.g. news website) robustly.

#### Building G (step 4)

Using Equation 2.9, we compute the cost $c(n, n')$ for each couple $(n, n')$ where $s_p(n, n') \neq 0$. Then, for each node $n' \in N'$, we add one edge for all nodes $values(neighbours(n')) \subset N$.

#### Metropolis Algorithm (step 5)

Once we built the graph $G$ with its associated costs, we need to find the set of edges $M$ in $G$ that represents the best full matching between $T$ and $T'$. To do so, we apply the METROPOLIS algorithm in a different way than FTM does: 1. we adopt an alternative objective function, and 2. SFTM matching suggestion function is faster to compute, as costs never need to be re-estimated.

Typically, FTM uses the objective function $f_{FTM}(M) = \exp(-\beta\ c(M))$. In the original FTM article, the authors noted that the parameter $\beta$ seemed to depend on $|M|$. To avoid this dependency, we, therefore, normalize the total cost:

$$f_{SFTM}(M) = \exp(-\beta\ \frac{c(M)}{|M|}) \tag{2.11}$$

The function $suggestMatching : M_i \mapsto M_{i+1}$ takes a full matching $M_i$ and returns a full matching $M_{i+1}$ related to $M_i$. In Algorithm 2,

1. $selectEdgeFrom(edges)$ loops through $edges$ (in order) and, at each iteration $j$, has a probability $\gamma \in [0, 1]$ to stop and return $edges[j]$,

2. *connectedEdges*(*edge*), where *edge* connects $u$ and $v$, returns the set $E$ of all edges connected to $u$ or $v$ (note that $edge \in E$).

**Algorithm 2** Suggest a new matching

**Inputs** $G$ : The bipartite graph, $M_i$: A full matching
**Output** $M_{i+1}$: the suggested full matching

$M_{i+1} \leftarrow []$
$remainingEdges \leftarrow sortedEdges(g)$
$toKeep \leftarrow randomInt(0, |M_i|)$
**for** j = 0 ... $toKeep$ **do**
    edge ← remainingEdges.first
    $M_{i+1}$.add(edge)
    remainingEdges.removeAll(connectedEdges(edge))
**end for**
**while** $remainingEdges$ is not empty **do** $edge \leftarrow selectEdgeFrom(remainingEdges)$ $M_{i+1}$.add(edge) remainingEdges.removeAll(connectedEdges(edge))
**end while**
**return** $M_{t+1}$

In practice, we first compute all the connected nodes and edges before storing them as dictionaries, so that the function *connectedEdges* in Algorithm 2 can be computed in $O(1)$ time. It is worth noting that, to allow fast removal, the list *remainingEdges* is implemented as a double-linked list. The parameter $\gamma$ defines a trade-off between exploration (low $\gamma$) and exploitation (high $\gamma$). For the Metropolis related parameters, we used mostly the values advised in the original FTM article [43]: $\gamma = 0.8, \beta = 2.5$ and a number of iterations of 10.

### 2.4.3 Complexity Analysis

We are interested in evaluating the time complexity of the algorithm concerning the size of both trees $N$. In our analysis, we consider that $N_{tk} = max(|tokens(n)|, n \in nodes(T))$, the maximum number of tokens per node is a constant since it does not evolve with $N$.

When building $G$, we first compute the inverted index $TM$, which requires iterating through the tokens of all the nodes in $T$, and thus implies complexity in $O(N \cdot N_{tk}) = O(N)$.

To find the neighbors of nodes from $T'$ using $TM$, we iterate through all the nodes in $T'$, while each node in $T'$ has $N_{tk}$ tokens. The number of nodes containing

a token is artificially limited to $f(N)$. Thus, building the similarity function $s_0$ takes $O(N \cdot f(N))$ time.

For each node $n'$ in $T'$, we create an edge for each neighbor $n$ of $T$. Each token $tk \in tokens(n')$ adds up to $f(N)$ neighbors. It means that the total number of edges is in $O(N \cdot N_{tk} \cdot f(N)) = O(N \cdot f(N))$.

Before executing the METROPOLIS algorithm on $G$, we sort all the edges by cost, which takes $O(N \cdot f(N) \cdot log(N \cdot f(N))) = O(N \cdot f(N) \cdot log(N))$ (as $f(N) \leq N$). Finally, at each step of the METROPOLIS algorithm, we run the *suggestMatching* function, which prunes a maximum of $O(f(N))$ neighbors for each one of the $N$ edges it selects.

Overall, sorting all edges requires the highest theoretical complexity: $O(N \cdot f(N) \cdot log(N))$. If no threshold is set—*i.e.*, $f(N) = N$—then the worst-case overall complexity of SFTM is $O(N^2 \cdot log(N))$, which keeps outperforming TED ($O(N^3)$) and FTM ($O(N^4)$).

In this evaluation, we used $f(N) = \sqrt{N}$, which leads to a theoretical worst-case complexity in $O(N \cdot \sqrt{N} \cdot log(N))$.

## 2.5 Empirical Evaluation

The objective of this evaluation is to assess that:

1. the quality of the matchings reported by SFTM compares with the baselines we selected—APTED and XYDIFF—and
2. SFTM offers practical computation times on real-life web pages.

### 2.5.1 Input Web Document Dataset

We need to assess the ability of SFTM to match the nodes between two slightly different web pages $d$ and $d'$. To measure and compare the accuracy of all studied solutions, we must have access to the ground truth matching between $d$ and $d'$—*i.e.*, for each node $n$ in $d$, what is the true matching node $n'$ in $d'$.

To the best of our knowledge, there is no established and public benchmark that includes such pairs of trees, along with the ground truth matching of their nodes. Creating such a benchmark is challenging. Existing matching solutions usually do not provide any qualitative empirical benchmark [10, 20, 36, 81, 83, 93] and challenging matching problems involve thousands of nodes, which makes manual labeling error-prone for humans (both trees could not even be rendered on the same screen).

Therefore, we built a semi-synthetic dataset built from mutations applied to real-life web pages, thus obtaining a large-scale dataset in which the ground truth is known.

**DOM mutation** To build a grounded dataset of $(d, d')$ pairs—*i.e.*, where the ground truth (perfect matching) is known—we developed a mutation-based tool that operates as follows:

1. we construct the DOM $d$ from an input web document,
2. for each element of $d$, we generate a unique signature attribute,
3. for each original DOM $d$, we randomly generate a set of mutated versions: the *mutants*. Each *mutant* $d'$ is stored along with the precisely described set of mutations that were applied to $d$ to obtain $d'$. Importantly, the signature tags of the elements in $d$ are transferred to $d'$, which constitutes the perfect matching between $d$ and $d'$. These signatures are ignored when applying the matching algorithms.

In our tool, most attention has been dedicated to the choice of relevant mutations to apply. We, therefore, relied on the expertise of web developers to identify the most common changes that can apply to DOM. Their feedback led to the identification of the following list of mutation operations:

| Category | Mutation Operators |
|---|---|
| Structure | `Remove, duplicate, wrap, unwrap, swap` |
| Attribute | `Remove, remove words` |
| Content | `Replace, remove, remove words, change letters` |

where *Structure: remove* removes an element and its children (recursively), *duplicate* duplicates a subtree, applies *mutate* to duplicated subtree, and inserts the subtree anywhere in the tree, *wrap* wraps an element within a new *div* container, *unwrap* removes an element $e$ and attach the children of $e$ to the parent of $e$, *swap* swaps the position of two sibling elements, *Attribute: remove* removes an attribute with its value, *Attribute: remove words* removes a random number of tokens from the value of an attribute, *Content: replace* replaces the content of an element with a random text whose size is close to the original, *Content: change letters* replaces a few letters in the content of an element, *Content: remove* removes the content of an element, *Content: remove words* removes random tokens from the content of an element.

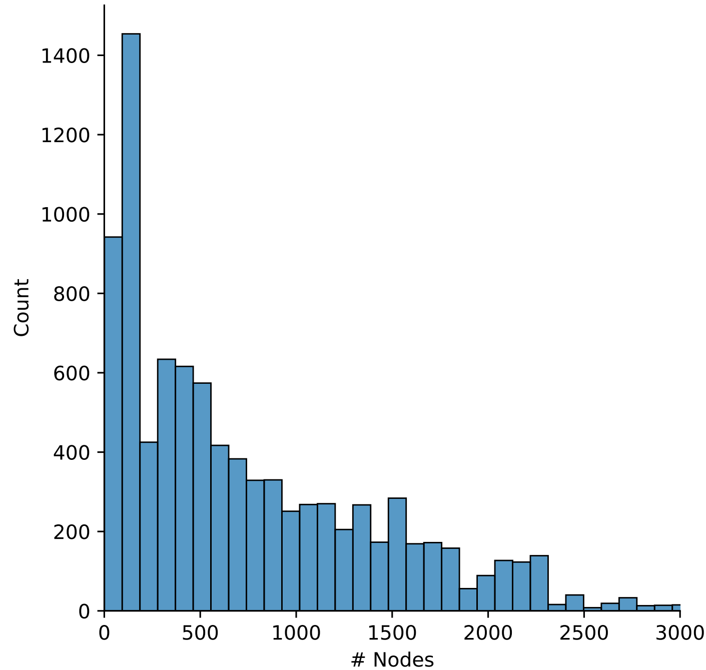



Figure 2.5: Distribution of DOM sizes (in terms of nodes) in the dataset.

We believe that the above mutations are representative of a wide range of changes that apply to web pages, although they may not perfectly cover all the cases encountered in practice. In particular, the distribution of these mutations might not be uniform in real life—*i.e.* some mutations might happen more than others. Yet, this evaluation intends to compare the sensitivity of studied matching algorithms to common mutations, which remains a relevant context to estimate and compare their quality.

**Input document sample** We fed our mutation tool with the home pages of the Top 1K Alexa websites.[2] Alexa provides a list of websites ordered by popularity, thus providing a representative list of web pages of variable complexities. For each original DOM $d$, we created 10 mutants $d'_0 \ldots d'_9$ with a ratio of mutated nodes ranging from 0 to 50% of the total number of nodes on the page, $|nodes(d)|$.

Overall, we obtained a dataset composed of $6,276$ document pairs $d, d'_n$ that could be correctly processed by the algorithms under study. Figure 2.5 reports on the size distribution, in the number of nodes, of original and mutated web documents included in this dataset.

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

### 2.5.2 Baseline algorithms

Given no implementation of the original FTM algorithm is available, we implemented and evaluated it, but the computation times and space complexity of this implementation were too high to run the algorithm on real-life web documents (*e.g.* for a tree of 58 nodes, the computation took 1 hour).

We thus mainly compare SFTM to APTED and XyDiff. APTED is the reference implementation of TED that reports on the best performance so far. The implementation of APTED used for this evaluation is the one provided by the authors of [69, 68]. Since APTED yields the optimal solution to the TED problem, TED is theoretically superior in accuracy to all more restricted solutions (see Section 2.2).

XyDiff is the most widely-known and used algorithm to efficiently match XML documents. Unlike APTED, XyDiff does not return an optimal result, it instead focuses on speed which makes it a complementary candidate to APTED as a baseline. In order to use XyDiff on HTML pages we had to convert the HTML into XHTML, which mostly means closing unclosed tags (*e.g.*, input tags). We used an existing open source implementation of XyDiff.[3] We consider the pairs $(d, d')$ taken from the above input dataset, and we systematically ran SFTM, APTED, and XyDiff algorithms with each pair to match $d$ with $d'$ on the same machine.

**Ground truth** When building the dataset, we keep track of nodes' signatures so that we always know which nodes from $d$ should match with nodes from $d'$. This ground truth is hidden from the evaluated algorithms but is used *a posteriori* to measure and compare the quality of the matchings computed by the algorithms under evaluation.

### 2.5.3 Experimental Results

All the results in this section have been obtained by running all three algorithms on the same server containing 252 GB of RAM and an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz.

**Matching quality** The signature tags injected in nodes from $d$ and $d'$ allow us to assess the quality of the matchings by comparing them to the ideal matching $M_{ideal}$. For the qualitative analysis, we model the tree matching algorithm as a binary classification problem: Given two trees $T$ and $T'$ containing the set of nodes

[3]https://github.com/fdintino/xydiff

$N$ and $N'$ respectively, $N \times N'$ is the set of all possible matches. We consider the matching $M_a \subset N \times N'$ produced by a tree matching solution $a$. Then, a match $e = (n, n') \in M$ is classified as positive by $a$ if $e \in M_a$. All matches that should be positive are in the ideal matching $M_{ideal}$. All possibilities are summarized in the following confusion matrix:

| | $e \in M_{ideal}$ | $e \notin M_{ideal}$ |
|---|---|---|
| $e \in M_a$ | **True Positive** | False Positive |
| $e \notin M_a$ | False Positive | **True Negative** |

Using the above confusion matrix, we can compute the *precision*, *recall*, metrics and the *F1 score*, which are very commonly considered for binary classification problems.

Figure 2.6 reports on the precision, recall and F1 score of the 3 tree matching solutions we compared, namely SFTM, APTED, and XyDiff. As expected, the accuracy of all solutions decreases when the mutation ratio increases. However, for all the reported metrics, SFTM outperforms both XyDiff and APTED. For both APTED and XyDiff, we believe the lack of accuracy stems from the lack of flexibility when matching labels. XyDiff relies entirely on hashing subtrees of the document and matching subtrees with identical hash. While this approach might be robust to small structural mutations, it is naturally very sensitive to large amounts of mutations when both structures and labels are mutated. Similarly, TED compares the labels of most pairs of nodes and generates an associated cost of `1` when the labels are identical and `0` when they are different (no gradual costs if the labels are similar).

**Completion time** For each couple ($d$, $d'$) retrieved from the dataset, we measured the time taken by SFTM, APTED, and XyDiff to compute a full matching. Figure 2.7 reports the average time to match DOM couples of increasing size (in terms of the number of nodes) for all three solutions. XyDiff exhibits a very fast computation speed and despite its numerous optimizations, APTED's computation times increase exponentially for large web documents. SFTM is not as fast as XyDiff, but seems to show reasonable growth when the size of web documents increases. Interestingly, APTED computation time varies greatly, which is due to the multiple heuristics used by this implementation to optimize the computation in certain situations.

Overall, one can observe that SFTM offers an interesting trade-off between two classes of tree matching algorithms: the ones maximizing accuracy at the cost of time, like APTED, and those minimizing the completion time at the cost of reduced accuracy, like XyDiff.

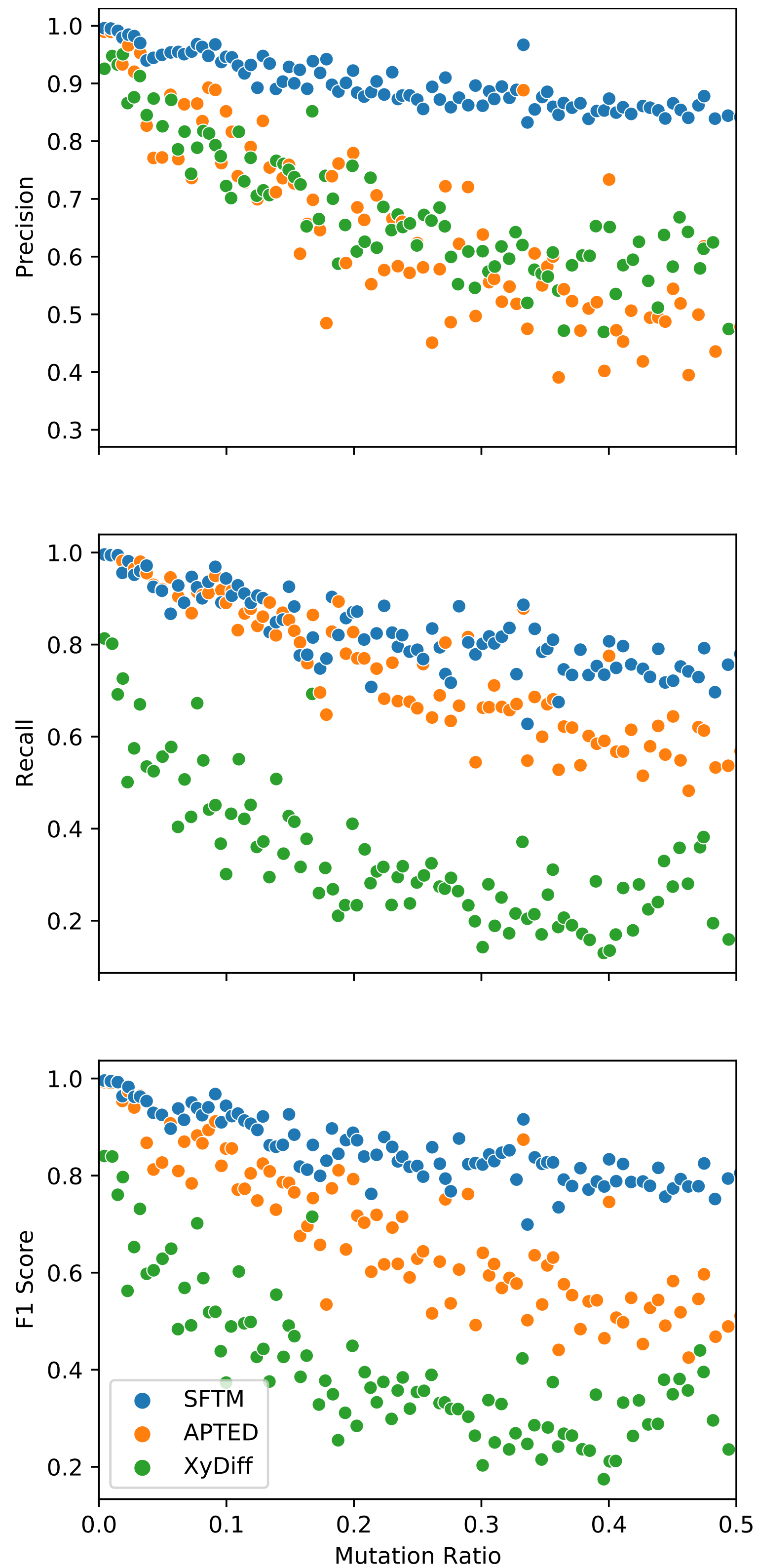



Figure 2.6: Precision, Recall, and F1 Score of SFTM, APTED, and XyDiff.

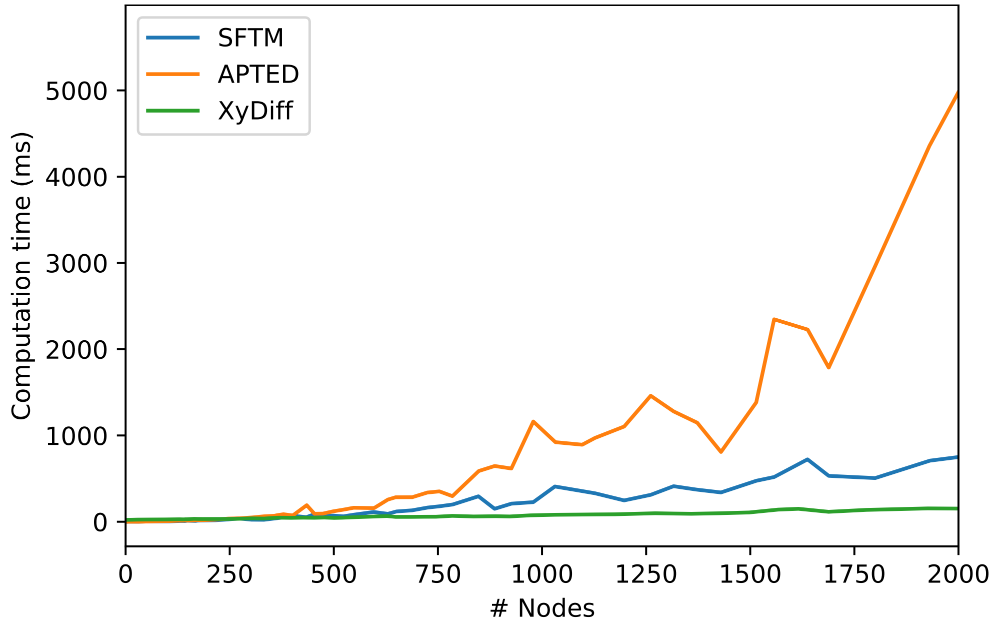



Figure 2.7: Computation times when matching trees of different sizes.

**Matching efficiency** The matching efficiency measures how fast a given solution can produce accurate results. Efficiency is a way to combine both accuracy and speed metrics into one that can be used to compare all solutions. In our case, we already showed that SFTM outperforms APTED in both accuracy and computation time. This efficiency measure is thus particularly interesting to compare SFTM to XyDiff, as SFTM outperforms XyDiff in terms of accuracy, but remains slower when it comes to speed. To compute this matching efficiency, we consider the same metric as [65]—*i.e.*, the number of good matches produced per millisecond. Figure 2.8 reports the matching efficiency of all three matching solutions. One can observe that SFTM produces 7.7 good matches per millisecond on average, which is far above APTED and XyDiff that produce 3.6 and 2.4 good matches per millisecond, respectively.

**Parameter sensitivity** Since we aim at improving the performances of FTM in terms of computation times, we study the sensitivity of the sub-linear threshold function $f$, which is a parameter that directly influences the computation time of the algorithm.

Figure 2.9, therefore, reports on the evolution of SFTM performances when $f$ varies. To study the sensitivity of $f$, we choose to use the power function $f(N) = N^\alpha$ as a threshold and display how the computation times and matching accuracy evolve with $\alpha$.

For this experiment, as we are interested in studying the sensitivity of the parame-

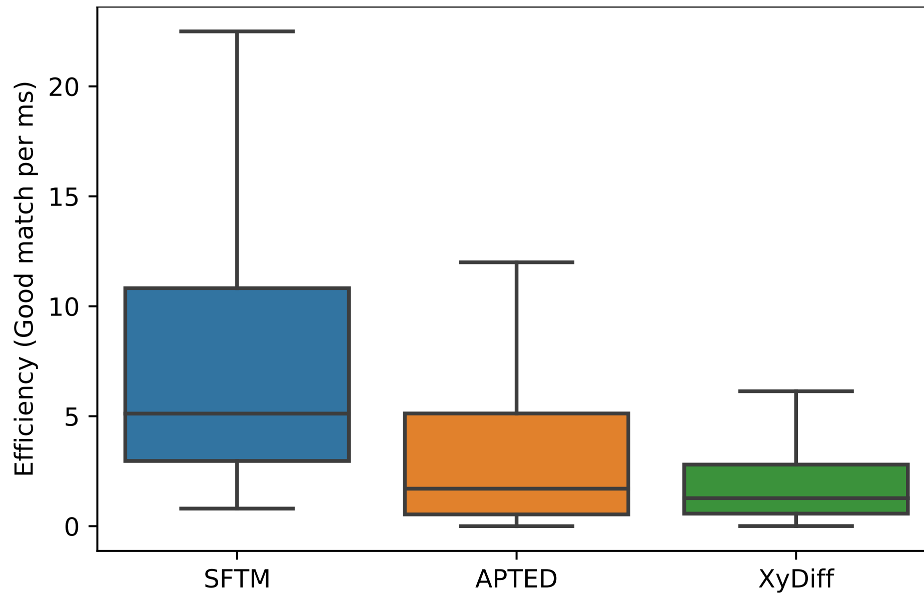



Figure 2.8: Matching efficiency of SFTM, APTED, and XyDiff.

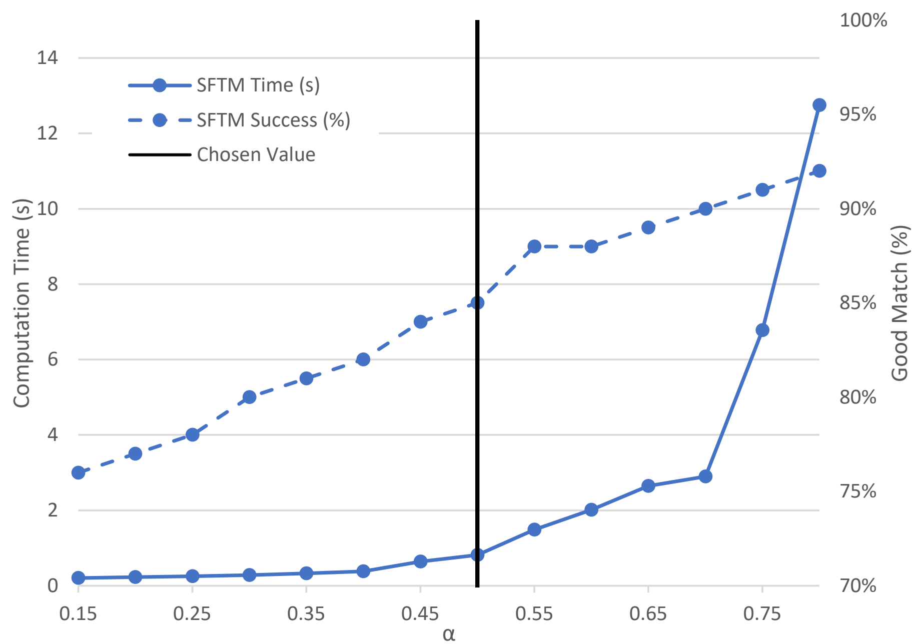



Figure 2.9: Performance of SFTM given $f(N) = N^{\alpha}$ according to $\alpha$.

ter $\alpha$ on the performances of SFTM, we, therefore, consider a subset of 243 pairs from the complete dataset used in previous sections (cf. Section 2.5.3), which represents a 6% error margin with 95% confidence.

As expected, when $\alpha$ increases, the quality of the matching and the computation times increase. However, beyond a certain value of $\alpha$, the increase of computation time is superior to the gain in accuracy: increasing $\alpha$ from 0.5 to 0.8 entails more than 10 times longer computation times for 8% gain in accuracy. Intuitively, this is because tokens contained in most nodes provide little relevant information (low IDF), but increase the complexity quadratically. In this chapter, we thus adopted $\alpha = 0.5$ (*i.e.*, $f(N) = \sqrt{N}$) as this value achieves good enough performances to demonstrate that SFTM can match two real-life web pages in practical time, with a minimum of compromise on quality.

## 2.6 Threats to Validity

The absolute values of completion times depend on the machine on which the algorithms were executed. As computations took time, we had to run both SFTM, APTED, and XyDiff on a server, which is shared among several users. Although we paid careful attention to isolating our benchmarks, the available resources of the server might have varied along execution thus impacting our results.

Our dataset contains the homepages of the Top 1k Alexa websites. The fact that our qualitative evaluation has only been conducted on homepages might have biased the results, as such pages might not be fully representative of the complexity of online documents. Yet, one can observe that the distribution of page sizes in our datasets offers a good diversity of situations.

The parameters used for SFTM and, in particular, the weights for the propagation may not be optimal. However, our evaluation shows that the adopted values succeed to report tree matchings that compete with the state-of-the-art accuracy in reasonable times and on a very large variety of web pages, which means the values we provided for the parameters do not require to be tuned for most web page matching cases.

## 2.7 Conclusion & Perspectives

Comparing modern real-life web pages is a challenge for which traditional *Tree Edit Distance* (TED) and XyDiff solutions are too restricted and computationally expensive. [43] introduced *Flexible Tree Matching* (FTM) to offer a restriction-free

matching but at the cost of prohibitive computational times.

This chapter thus introduced *Similarity-based Flexible Tree Matching* (SFTM), the first implementation of an advanced *Flexible Tree Matching* (FTM) algorithm with scalable computation times. We evaluated our solution using mutations on real-life web pages and we showed that SFTM outperforms XyDiff qualitatively and compares to TED, while significantly improving the computation time of the latter. Our proof of concept demonstrates that accurate matching of real-life web pages in practical time is possible.

Our *label-centric* approach to matching is significantly different than previous *structure-centric* techniques. In addition to providing a competitive solution to match web pages, we hope that our solution will encourage the development of solutions based on similar approaches. We also believe that having a robust algorithm to efficiently compare web pages will open up new perspectives within the web community.

In future work, we will further investigate how to improve the quality of the tree matchings by analyzing which situations cause SFTM to report mismatches and establishing guidelines to adjust the exposed parameters.

Finally, whether our work might apply to other trees than web DOMs remains to be demonstrated. Indeed, SFTM strongly relies on the fact that node labels in DOMs are highly differentiating (many specific attributes on each element), which is not the case for all kinds of trees.

The final objective of our work is to build a set of tools allowing us to create an abstraction of any web application. Being able to compare two web pages is the first step. In the next section, we will apply tree-matching to the robust locator problem.

# Chapter 3

# Erratum

## Summary

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 is *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 to either too long computation times or incorrect element repairs. We, therefore, adopt 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%.

## 3.1 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 fails 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 [34]. 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 [28]. More specifically, state-of-the-art solutions to the locator repair problem, WATER [15] and VISTA [77], 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.

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 the efficient *Similarity-based Flexible Tree Matching* (SFTM) algorithm we developed to repair all broken locators by matching all changes in a web page.

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

As described in chapter 2, SFTM is a tree matching algorithm providing fast computation times and high accuracy when compared to other generic tree matching solutions. To do so, SFTM builds on 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 building 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). 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 [15], both in accuracy (67% improvement on average) and performance, 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 resilient architecture centered around Erratum allowing us 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 (RPA)* [35]

**Summary** Overall, the key contributions in this chapter consist of:

1. proposing a solution to the locator repair problem leveraging the principles of *Flexible Tree Matching* (FTM),
2. implementing and integrating an efficient algorithm, named Erratum, to repair broken locators,
3. providing a novel, reproducible, large-scale benchmark dataset to evaluate both the robust locator and locator repair problems,

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

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.

**Outline** The remainder of this chapter is organized as follows. Section 3.2 introduces the state-of-the-art approaches in the domain of robust locators and locator repair, before highlighting their shortcomings. Section 3.3 formalizes the locator problem we address. Section 3.4 introduces our approach, ERRATUM, which leverages an efficient flexible tree matching solution that we identified. Section 3.5 describes the locator benchmark we designed and implemented. Section 3.6 reports on the performance of our approach compared to the state-of-the-art algorithms. Finally, Section 3.7 presents some perspectives for this work, while Section 3.8 concludes.

## 3.2 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 describe state-of-the-art approaches to repairing broken locators, and in particular, the literature published in the domain of web test repair.

### 3.2.1 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 with 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 [11, 75, 60] to visually record their scenarios. In both cases, the scenario requires identifying 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* [46].

In practice, automated tests are often subject to breakages [28]. It is important to understand that a test *breakage* is different from a test *failure* [77]: 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, [28] reports that 74 % of web tests break because one of the included locators fails to locate an element in a web page.

### 3.2.2 Generating Web Element Locators

The fragility of locators remains the root cause of test breakage, no matter whether 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 [46], ROBULA+ [49], which are algorithms that apply successive refining transformations from a raw XPath query until it yields an XPath locator that exclusively returns the desired element. Leotta *et al.* [47, 44] also propose Sideral, a graph-based algorithm to generate XPath locators. Sideral requires to specifically train on each application to learn what properties are most likely safe to rely on when building XPath locators. While these methods all use ancestors of a given element as an anchor to generate a locator, [64] uses siblings instead, arguing that they make more reliable anchors.

Another work by Yandrapally *et al.* leveraged contextual clues to generate locators [87]. These clues rely mostly on the content surrounding the element to locate which may be problematic in case the content changes. LED [5] 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 [48, 95, 56]. 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 [28].

### 3.2.3 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 ($D'$).

**Property Based** In this area, WATER [15] and COLOR [40] provide an algorithm to fix broken tests using intrinsic properties of the element to relocate. 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 and COLOR scan all elements in the new version and return 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.* [27] 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.

**Vision Based** Using a completely different approach, VISTA [77] is a recent technique that adopts computer vision to repair locators. VISTA falls within the category of computer vision-aided web tests [13, 51, 3]. However, while using computer vision succeeds in repairing most of the *invisible* changes, such solutions tend to fail when the content, language, or 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).

Finally, J.Imtiaz et al [33] developed a test repair solution that integrates several different capture replay tools. While we focus specifically on the locator repair problem, they used and evaluated a more comprehensive test-repair strategy involving the classification of the test script and detected breakages and the extension of the UML Testing Profile specifications to capture more interaction details.

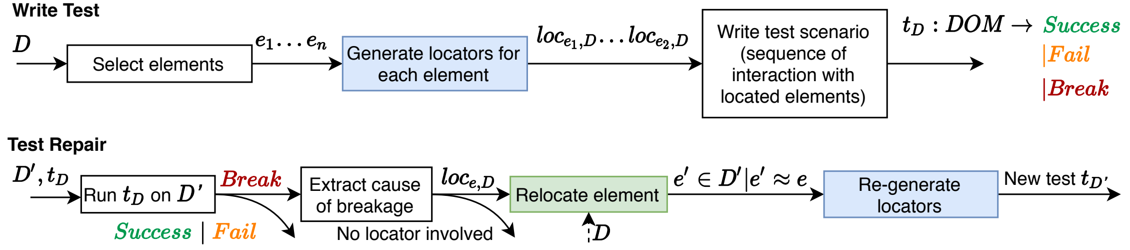



Figure 3.1: Illustration of the locator problem statement in automated tests combining the *robust locator* (in blue) and the *locator repair* (in green) problems.

## 3.3 Locator Problem Statement

Figure 3.1 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. 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 3.1, namely the *robust locator* (in blue) and *locator repair* (in green) problems for the general case of web automation scripts.

### 3.3.1 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 3.3.1.** *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:*

$$eval(loc_{E,D}, D) = E \tag{3.1}$$

*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 3.3.2.** *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 3.3.3.** *Given a locator $L = (loc_{E,D}, eval)$, $L$ is* ***robust*** *to a mutation function mut if:*

$$eval(loc_{E,D}, mut(D)) \approx E \tag{3.2}$$

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, their values, and the textual content. However, in the context of ERRATUM, we willingly ignore the content as described in section 3.4.3.

### 3.3.2 Problem Statement

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

**RQ 3.3.1.** ***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 3.3.1, this piece of information allows to locate the originally selected elements $eval(loc_{E,D}, D) = E$.

**RQ 3.3.2.** ***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. 3.3.1).

We thus present a novel approach to solving the locator repair problem.

## 3.4 Repairing Locators with Erratum

The previous section formalized both *robust locator* and *locator repair* problems. The approach we report, 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) [80] or tree alignment [36]. 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.

### 3.4.1 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 3.2 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 [15, 77]) is often challenging and can easily lead to a mismatch. By using tree matching (cf. right-bottom of Figure 3.2), 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 2.

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 $T$;

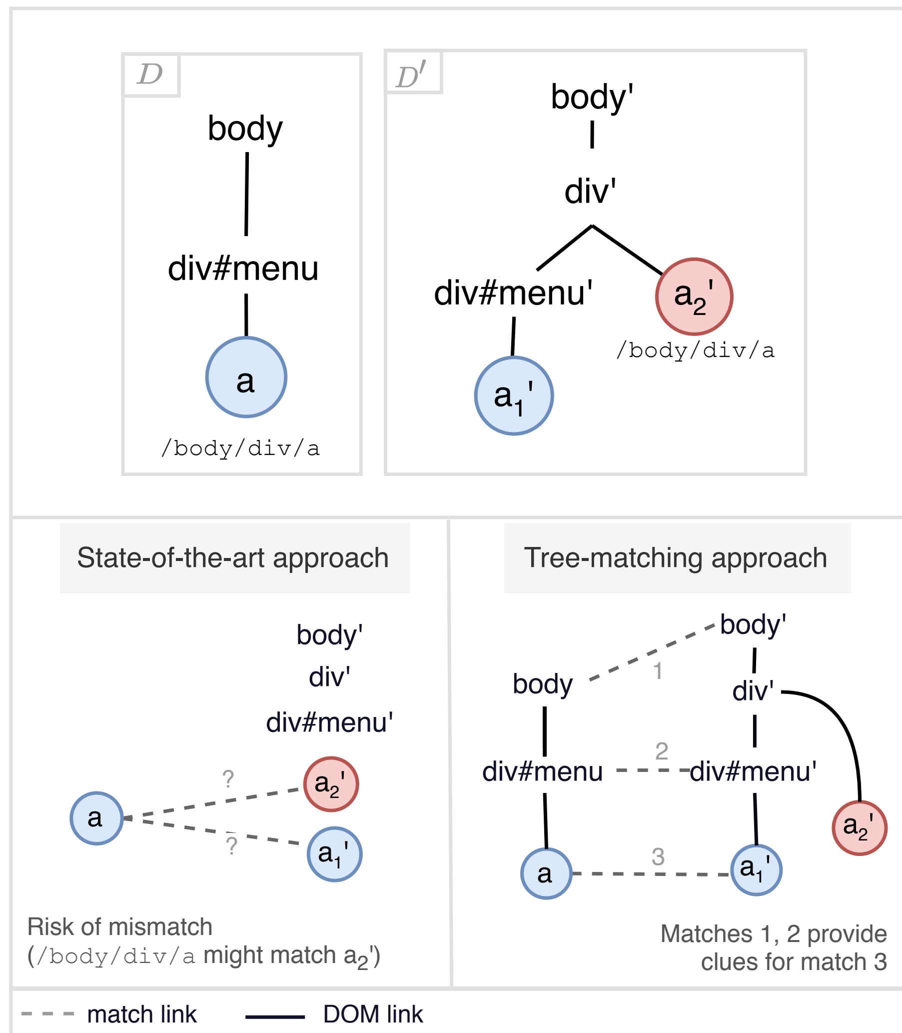



Figure 3.2: State-of-the-art Vs. tree matching locator repair.

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 3.1, 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 [46], and is therefore considered out of the scope of our study.

### 3.4.2 Integrating a Scalable Tree Matching Algorithm

The state-of-the-art approach to match two trees is *Tree Edit Distance* (TED) [80]. 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 [43]: 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 [9]) and, more practically, our preliminary experimentation has shown that applying the state-of-the-art implementation of TED, named APTED [69], 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 [37, 81, 94], but at the cost of even more restrictive constraints on the produced matching (*e.g.*, the tree alignment problem [37] 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 [43]. 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 the previous chapter (2), 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.

### 3.4.3 Matching DOM Trees by Similarity

Given the limitation of FTM, ERRATUM integrates the *Similarity-based Flexible Tree Matching* (SFTM) algorithm we developed in the previous chapter, 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 performance. 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.

In this section, we walk through the key steps of the SFTM algorithm (described in chapter 2) applied to ERRATUM's approach. We use HTML snippets reported in Figure 3.3. 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 3.4 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 3.1). The interested reader can refer to our technical report 2

(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 3.3: Two versions of an HTML snippet extracted from the homepage of *duckduckgo.com*.

for an exhaustive description of our SFTM algorithm, whose applications go beyond the context of repairing broken locators.

**Step 1: Node Similarity** Elements of DOMs $D$ and $D'$ are compared. The first step consists in computing an *initial similarity* $s_0 : D \times D' \rightarrow [0,1]$. 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 with this element—*i.e.*, thus ignoring the content from the child elements.

**Example 3.4.1.** *The* label *computed for $div_1$ (cf. Figure 3.3) 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) [38] formula to take into account how rare each token is.

**Example 3.4.2.** *Following our previous Example 3.3, 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 little information to the relevance of a given match, while they cause a significant increase of potential matches to consider. That is why, to reduce the computation times, the algorithm rules out the 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 3.4.3.** *In Example 3.3, 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*

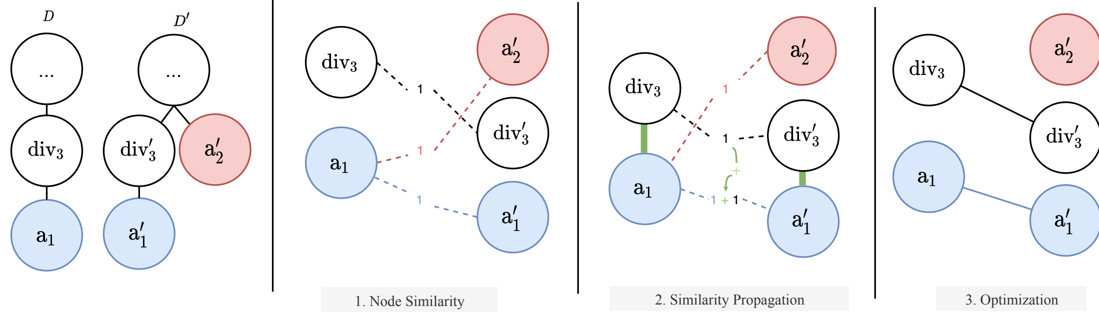



Figure 3.4: Key steps followed by our *Similarity-based Flexible Tree Matching* (SFTM) algorithm.

*the fact that the parents of* $a_1$ *and* $a'_1$ *are 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.

To approximate the optimal set of matches, SFTM implements the Metropolis algorithm [58]. 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 is to handle variations of labels—*e.g.*, renaming a class, adding an id, and removing an attribute. This is because of the fuzzy nature of the first similarity steps of SFTM: as long as two elements' labels share enough rare tokens, they will match. For elements that have little 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 labels containing little 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 a 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.

## 3.5 The Robust Locator Benchmark

In our context we are interested in covering the following research questions:

**RQ 3.5.1.** *How does* ERRATUM *perform in solving the locator repair problem (cf. RQ. 3.3.2) when compared to state-of-the-art solutions?*

**RQ 3.5.2.** *What are the factors influencing the accuracy of WATER and* ERRATUM*?*

**RQ 3.5.3.** *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.

### 3.5.1 Evaluated Locator Repair Solutions

We compare two solutions: 1. ERRATUM, our approach to repairing broken locators by leveraging flexible tree matching, and 2. WATER, the reference implementation of a locator repair technique applied to web test scripts [15].

The original algorithm of WATER analyses a given test case and finds the origin of the test breakage and suggests potential repairs to the developer. In our context 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 [77] 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.

### 3.5.2 Versioned Web Pages Datasets

In the remainder of this chapter, 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 [49, 46] and robust locator [77, 15, 27] 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** 2 generated by applying random mutations to a given set of web pages (see Section 3.5.2),
2. A **WAYBACK dataset** collects past versions of popular websites from the Wayback API (see Section 3.5.2).

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 3.1 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 because 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 3.5.2);
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 3.1: 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.

### Building the MUTATION dataset.

We extend the technique we introduced to generate a MUTATION dataset in 2. The mutation dataset is built by applying a random amount of random mutations to a set of original web pages: 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

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

Table 3.2: Mutations applied in the MUTATION dataset 2.

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

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 3.2 describes the set of DOM mutations that can be observed along the evolution of a web page.

The original websites from which mutants were generated were randomly selected from the Top 1K Alexa. Figure 3.5 depicts the distribution of DOM sizes in this synthetic dataset.

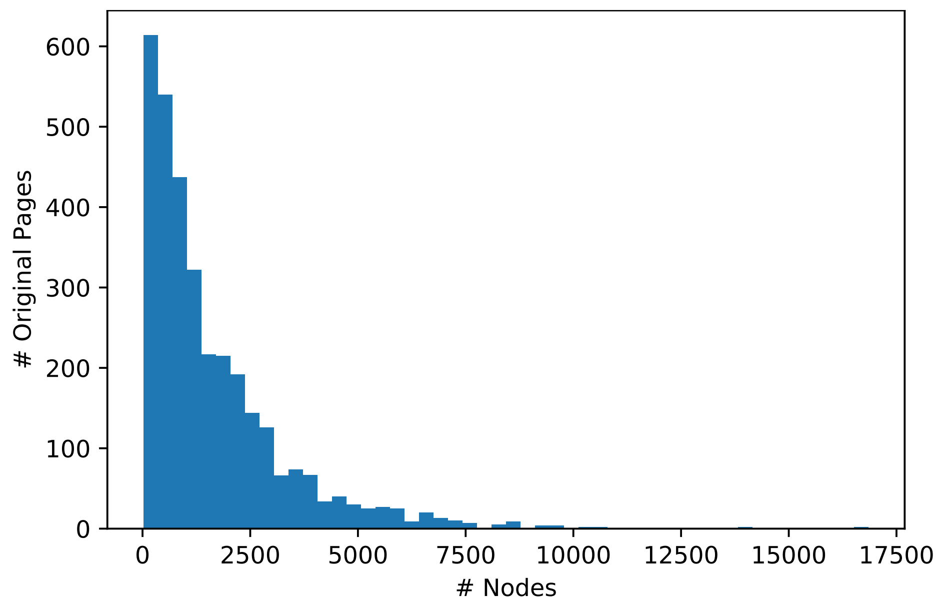



Figure 3.5: Distribution of DOM sizes (in number of nodes) in the MUTATION dataset.

This dataset was built with an automation tool that we made available along with its source code 3.8. From a given list or source URLs, our tool creates a dataset of randomly mutated web pages following the above-described approach.

**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 the 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 3.6).

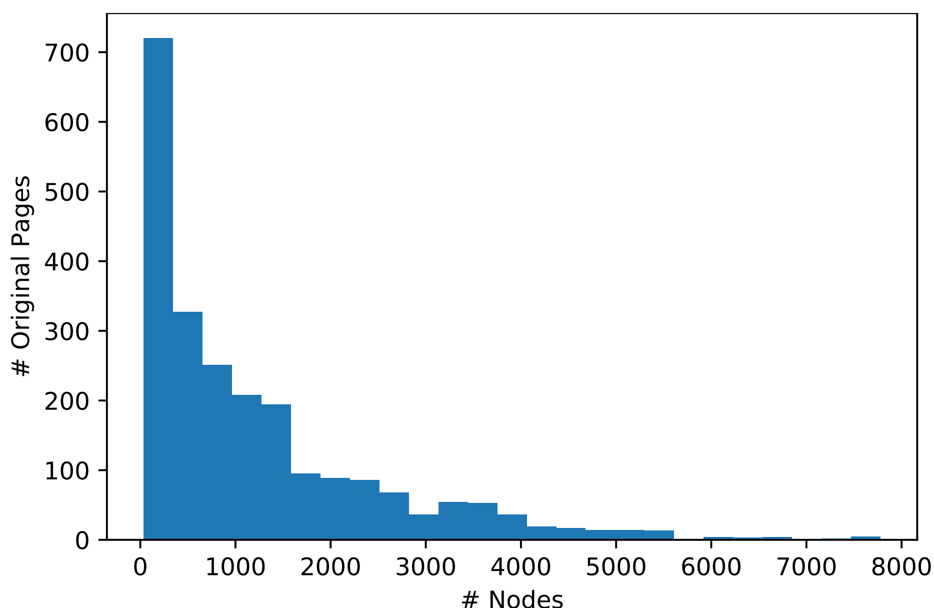



Figure 3.6: Distribution of DOM sizes (in number of nodes) in the WAYBACK dataset.

**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 3.3.2): 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 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 $D$ to $D'$. 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 of located elements is 0.5 %.

### 3.5.3 Evaluating of the Matched Elements

On the Mutation dataset, the signature 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 situa-

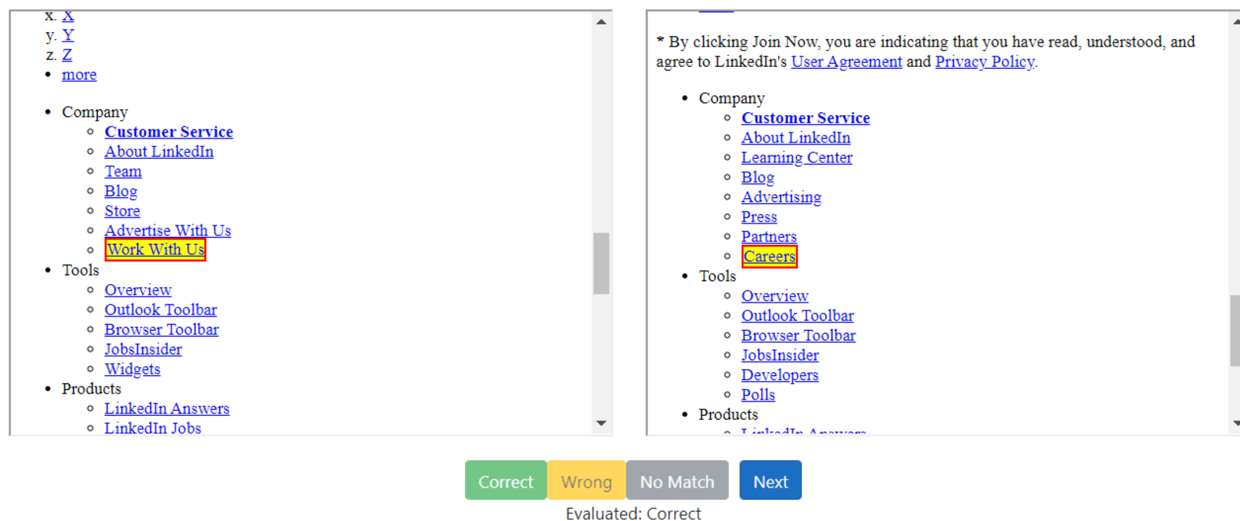



Figure 3.7: Labeling a given element matched by ERRATUM on two versions of the Linkedin homepage. The screenshot comes from the visual matching application we created to manually label disagreements between ERRATUM & WATER.

tion, 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 3.7) 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 3.3.1), 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.

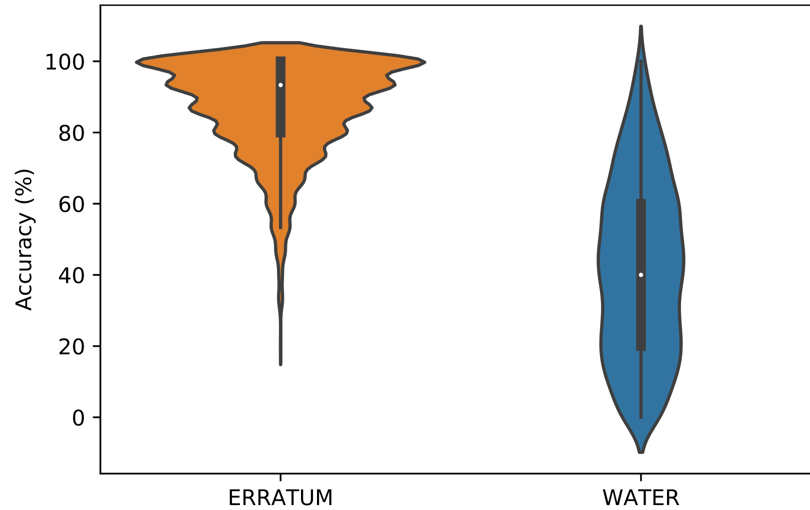



Figure 3.8: Accuracy distribution of ERRATUM and WATER on the MUTATION dataset.

## 3.6 Empirical Evaluation

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

### 3.6.1 Evaluation of Repair Accuracy

In this section we answer RQ.3.5.1: How does ERRATUM perform in solving the locator repair problem (RQ. 3.3.2) when compared to state-of-the-art solutions?

**Repair accuracy on the MUTATION dataset.** Figure 3.8 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 3.3 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 3.3: 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.3.5.2), 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 3.2) 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 3.9 depicts the sensitivity of both 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 3.10), while Erratum exhibits a more stable performance. The Spearman correlation coefficient between the error ratio of WATER and the size of the DOM is $\rho = 0.41$ com-

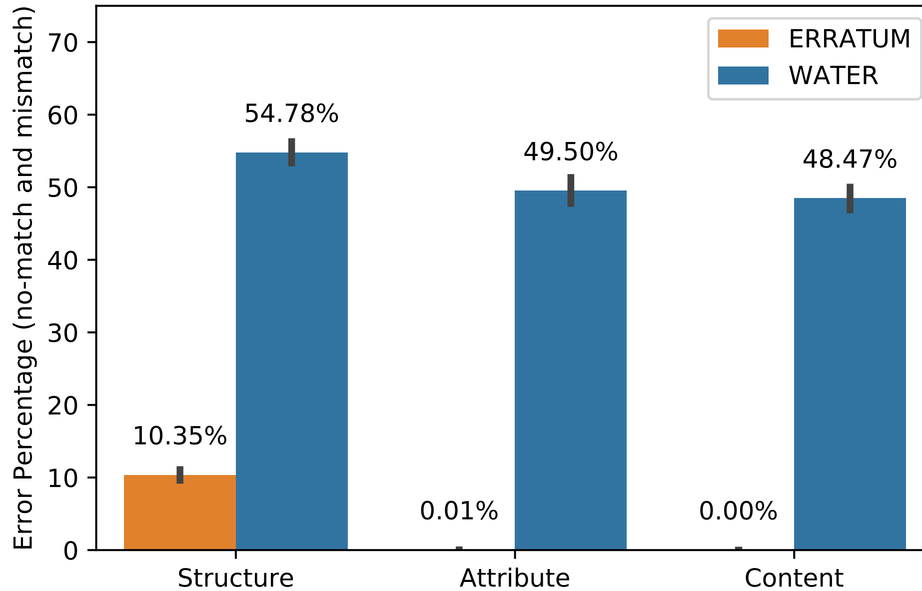



Figure 3.9: Error percentage according to the mutation type.

pared to 0.28 for Erratum. Interestingly, while Erratum correlates rather strongly ($\rho = 0.46$) with the percentage of mutation between the two DOM versions, WATER shows almost no correlation with the same variable ($\rho = 0.12$). It means WATER is surprisingly not impacted by the number of mutations between the versions. The dependency to the mutation ratio of Erratum is easily explainable: for each match, Erratum indirectly relies on structural and textual similarities in the whole DOM which means mutations anywhere in the DOM could theoretically impact the scores on which Erratum relies to compute a matching. Conversely, the WATER approach is fundamentally more local to the element to match. We believe these are key insights 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 $D$, concerning 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 $e$ tend 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 for the increase in possible combinations.

Finally, regarding the impact of the mutation ratio ($\frac{\#mutations}{\#nodes}$), Figure 3.11 reports 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

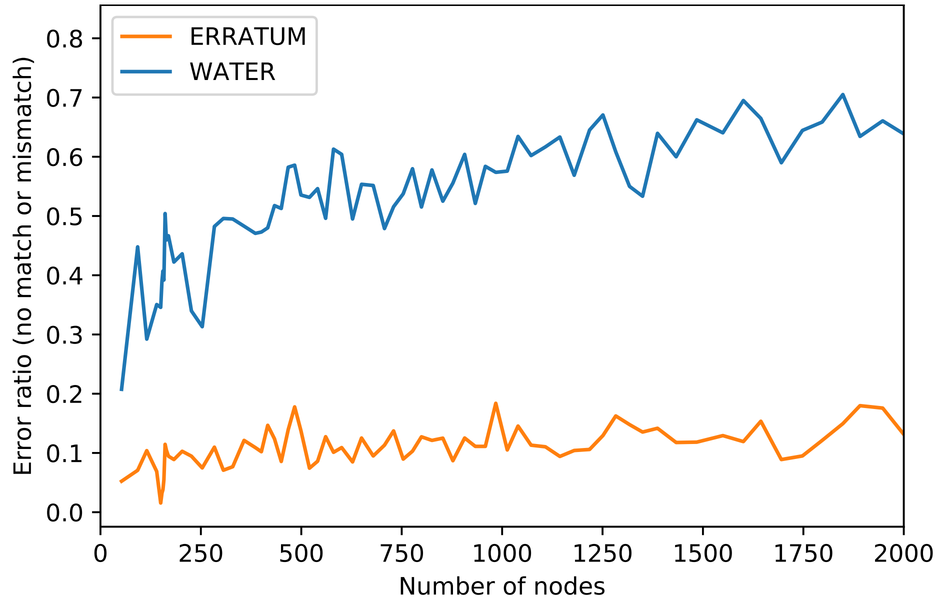



Figure 3.10: Errors rate evolution according to DOM size.

orange lines, and 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 exhibits 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 was labeled by web testing experts, which corresponds to a 5% confidence interval at 95%. Table 3.4 reports on the results of the manual labeling (for disagreements), thus assuming that both WATER and ERRATUM are correct whenever they agree.

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 3.7) and requested the participants to eventually propose a matching

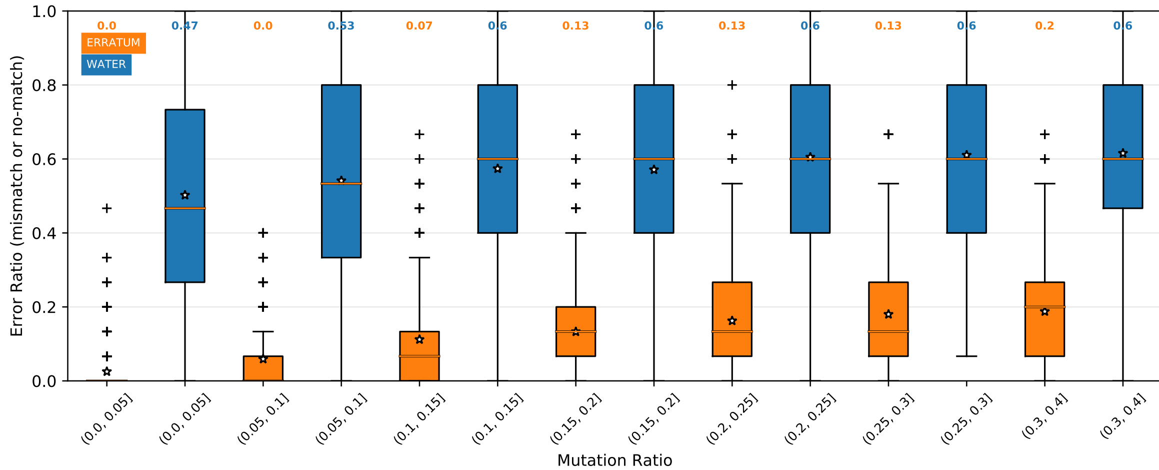



Figure 3.11: Errors rate evolution according to the mutation ratio.

Table 3.4: 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% | |

element if a no-match was reported by ERRATUM. The result of this evaluation, summarized in Figure 3.12, 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 was subject to many mutations, thus misleading ERRATUM as already observed in Figure 3.9.

**Comparison of repair accuracy.** Interestingly, as shown in Table 3.5, 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 3.9), one can guess that real-life websites are more subject to *content* and *attribute*-related mutations than *structure*-based mutations (cf. Table 3.2), 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

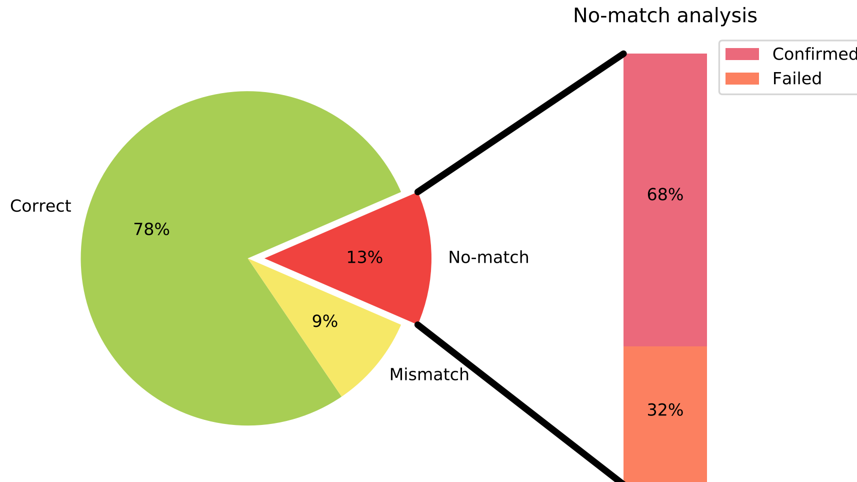



Figure 3.12: Analysis of matches labeled as no-match by ERRATUM.

of real-life mutations.

Table 3.5: Accuracy summary across datasets.

| | MUTATION | WAYBACK |
|---|---|---|
| **ERRATUM** | 87.0% | $78.3 \pm 5\%$ |
| **WATER** | 42.1% | $51.9 \pm 5\%$ |

### 3.6.2 Mutations in the WAYBACK Dataset

To assess the accuracy of ERRATUM on both datasets, we study the nature of the changes occurring between two versions of a given page in the WAYBACK dataset. The changes applied along versions of pages available in the WAYBACK dataset are not labeled, thus lacking a ground truth. The robust locator benchmark (cf. Section 3.5) assumes the input datasets as the ground truth to evaluate ERRATUM on the locator repair problem. Conversely, this section assumes the matching algorithm exploited by ERRATUM—*i.e.*, SFTM—to be correct and uses it to label the mutations observed in the WAYBACK dataset. To do so, we estimate the number and types of mutations between each pair of web pages $(D, D')$ by leveraging the resulting matching $M = sftm(D, D')$. The following table lists the considered mutation types, considering $\forall e \in D$ and $\forall e' \in D'$, where $p(e)$ is the parent of $e$ and $\lambda(e)$ is the label of $e$:

**Dataset Consolidation.** The robust locator benchmark considers a subset of the WAYBACK dataset, manually labeled by experts. However, during this process, potential inconsistencies observed in rendered web pages were ignored by experts. There-

| Label | Mutation | Category |
|---|---|---|
| addition | $\nexists e \mid (e, e') \in M$ | Structural |
| removal | $\nexists e' \mid (e, e') \in M$ | Structural |
| move | $(e, e') \in M$ and $(p(e), p(e')) \notin M$ | Structural |
| relabel | $(e, e') \in M$ and $\lambda(e) \neq \lambda(e')$ | Relabel |

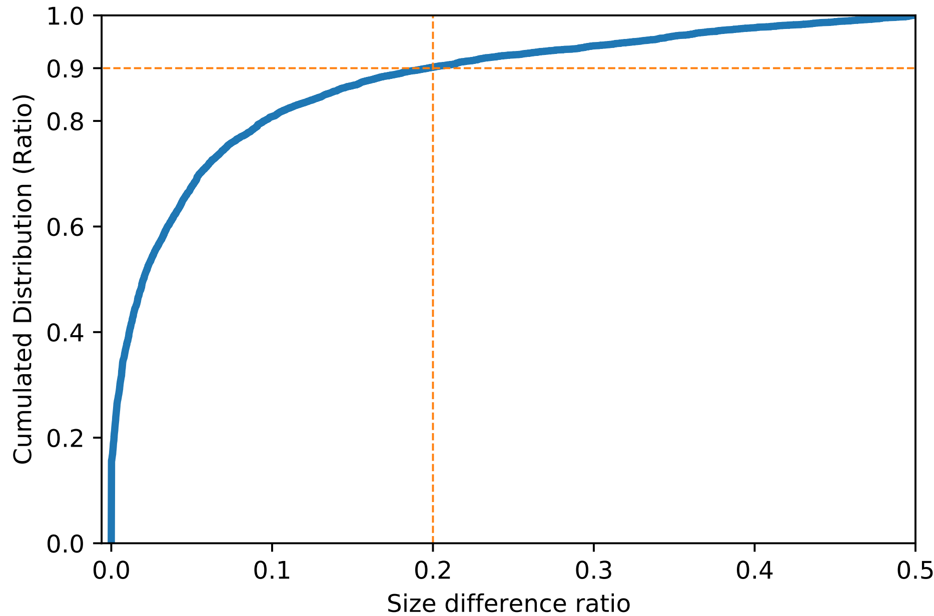



Figure 3.13: Cumulative Distribution of ratios between two versions of web pages. The orange dotted lines show the threshold used in this experiment

fore, before analyzing the full WAYBACK dataset, one should discard such inconsistencies between unrelated web pages, including cases where: *(a)* one of the versions can be an error page or a redirection page, *(b)* the website may show "in construction" or may have closed between the two snapshots. For this reason, we apply two heuristics to prepare the dataset by removing all the pairs where:

1. one of the two versions has less than 30 nodes, reflecting one of the above inconsistencies. For example, even a minimalist web page, like Google, contains more than 250 nodes;
2. the absolute ratio of sizes between the two versions exceeds 40% which selects approximately 90% of the dataset (see Figure 3.13). When this ratio is large, comparing the two pages is likely to be conceptually irrelevant.

While the initial WAYBACK dataset contains $19,161$ pairs of web pages, the application of the above rules leads to a consolidated dataset of $8,641$ pairs. Figure 3.13 reports on the distribution of ratios of web page versions in the consolidated WAYBACK dataset.

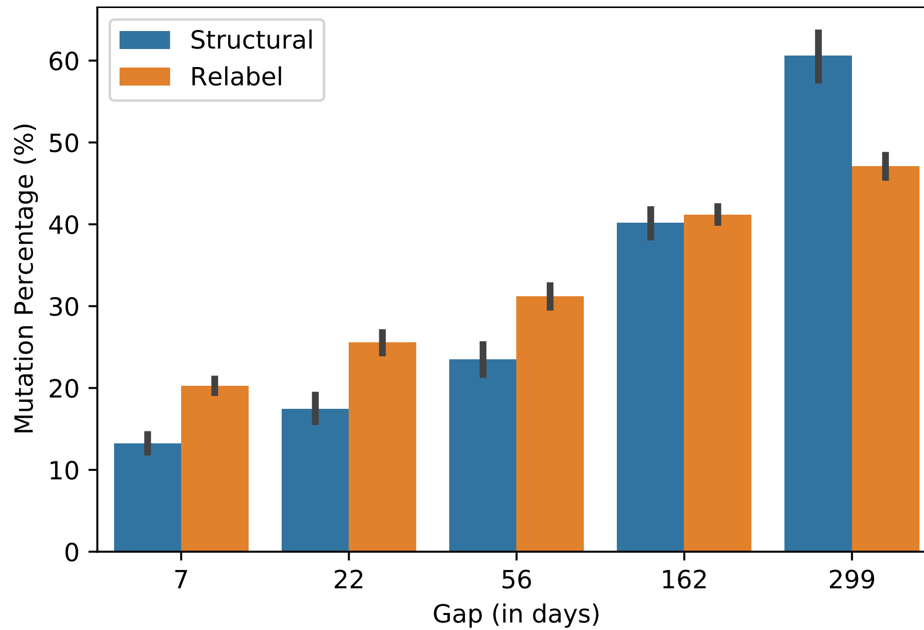



Figure 3.14: Mutation ratio between two WAYBACK snapshots depending on gap duration.

**Mutation Frequency.** Figure 3.14 further analyzes the above dataset by reporting on the ratio of mutations by category occurring between two versions of a web page.

As expected, the mutation ratio increases with the gap between the versions. Most importantly, the average mutation ratio in the WAYBACK dataset is 60%, i.e. on average, 60% of the nodes have mutated between two versions $D$ and $D'$ from the Wayback dataset. This mutation ratio is significantly higher than the average amount of mutations in the MUTATION dataset: 20%. This difference is probably an important factor justifying the differences of accuracy measured on both datasets. Comparing versions with large gaps is practical because it ensures there will be differences between the versions. In addition, it provides interesting insights into the frequency of changes on popular web pages. However, in the context of web testing, the mutation ratio between two consecutive versions is unlikely to reach 60%, in particular when adopting test-driven developments.

**Mutation Labels.** Figure 3.15 further describes the distribution of mutation labels in the WAYBACK dataset. The figure highlights that relabel is the most common mutation, while move is quite rare. This result can be explained by the following observations: 1. when a subtree is moved, only one move is accounted for even if the visual change may appear as important, and 2. the SFTM algorithm is particularly robust to relabels, which could also be a factor explaining the observed ratio.

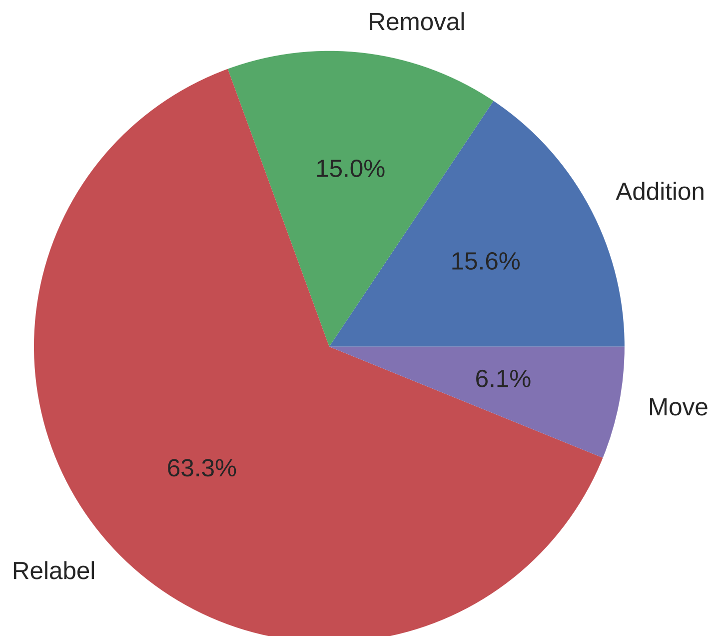



Figure 3.15: Distribution of mutation labels in the WAYBACK dataset.

### 3.6.3 Repair Time Evaluation

In Figure 3.16, 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 ($\alpha$) when increasing the number of locators to repair on a web page. Figure 3.17 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 on a web page.

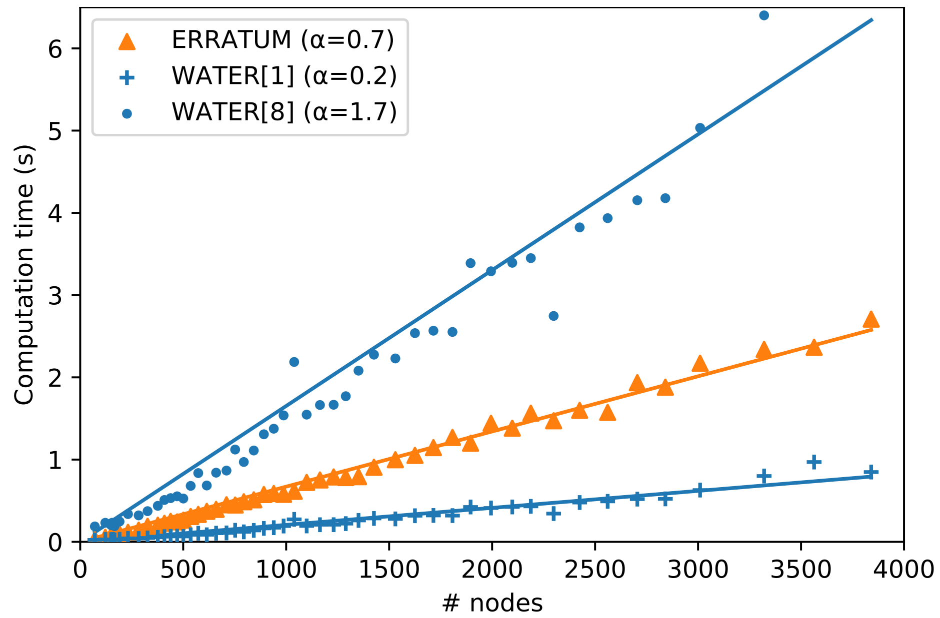



Figure 3.16: Repair time evolution according to DOM size.

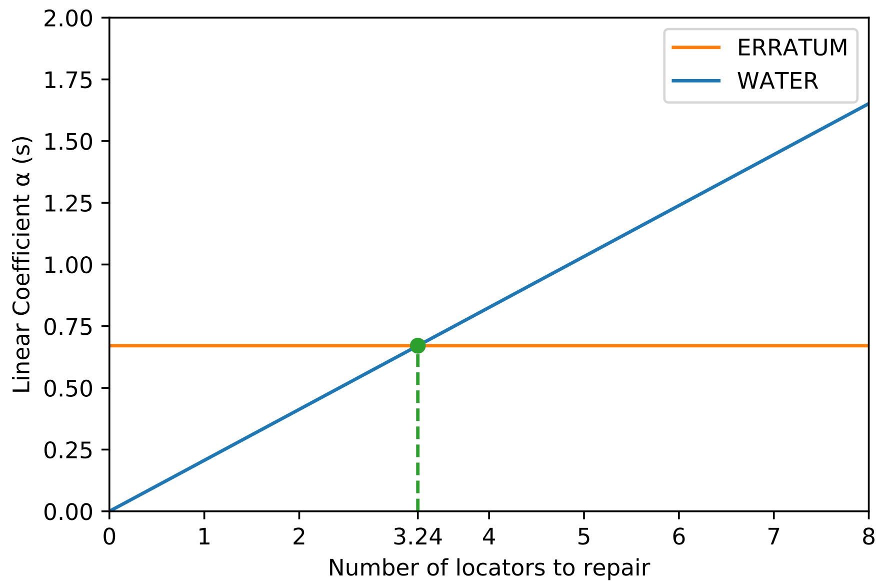



Figure 3.17: Performances of ERRATUM and WATER.

### 3.6.4 Threats to Validity

As described in Section 3.5.3, 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 bias, we recommended systematic and consistent decisions to label the data (cf. Section 3.5.3).

All our experiments with ERRATUM adopt the default FTM parameters, as recommended by [43]. 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 page 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.

## 3.7 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 generate a unique identifier $e_{id}$ (e.g UUID),
3. in the web automation script, instead of using a manually encoded (or gener-

ated) 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) [77]. In these situations, it is harder to locate the origin of the breakage from the test script. The silent breakage problem happens because when using XPath locators to relocate $e$ in $D'$, the XPath query either succeeds or fails. There is no indication of the confidence of the relocation that would help to detect a mismatch. On the opposite, Using ERRATUM, every individual match between two elements $e$ and $e'$ has an associated cost $s(e, e')$ that can be used as a confidence level to avoid mismatch.

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 thus help to detect possible breakage before the scripts are even run, thus diminishing the chances for a "silent breakage" to occur.

## 3.8 Conclusion

In this chapter, 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 [28]. Our analysis of the state-of-the-art approaches on this topic contributed to formalizing 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 performance —by providing faster computation time than WATER when repairing more than 3 locators in a web test script.

Finally, we worked with the development team of a widely used open source test framework called Cerberus [6] to integrate [7] the Erratum approach into the test creation part of the software. The next section describes the integration of Erratum to Cerberus and its impact on some of its users.

[4]Dataset available from `https://zenodo.org/record/3800130#.XrQb02gzY20`
[5]Benchmark available from `https://zenodo.org/record/3817617#.XrWdqGgzaoQ`
[6]https://cerberus-testing.com/
[7]https://github.com/cerberustesting/cerberus-source/commit/0a70d4cc0d70a797901652fd2b97d501bb7fa511

# Chapter 4

# Integrating Erratum into Cerberus

## Summary

Web applications are constantly evolving to integrate new features and fix reported bugs. However, 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 elements of a web application. In this context, the continuous evolution of web applications makes it extremely challenging to automate test scenarios of a web application robustly. More precisely, the major cause of breakages observed in integration tests is *element locators*, which are identifiers used by automated tests to navigate across a DOM. When this DOM evolves, these locators tend to break, thus causing the related tests to no longer locate the intended target elements.

So far, established test automation frameworks adopted by the industry, like Cerberus, only support CSS or XPath selectors to query web page elements. Such web locators require to be manually crafted and are often fragile with regards to DOM changes, hence often requiring testers to fix the broken test scenarios in priority. While several solutions to this problem have been proposed in the scientific literature, to our knowledge, no locator repair or locator generator has been integrated into open-source test-automation software before. We thus report on the seamless integration of a more robust web locator in Cerberus by leveraging a new locator repair solution, named Erratum. While Erratum was originally designed to repair broken locators, this chapter demonstrates how it can be extended to deliver an elegant robust locator solution that succeeds to reduce the maintenance cost of automated tests.

## 4.1 Introduction

The implementation of automated tests on web applications (apps) requires software engineers to locate specific elements in the DOM (*Document Object Model*) of a web page. To do so, software testers 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 fails in all the associated test scenarios that apply to the modified web pages.

Several existing works focus on repairing tests on GUI applications, but there are surprisingly very few test repair solutions targeting web interfaces [34]. 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 fix a broken locator, thus leaving all the related automation tests as broken [28].

While we already introduced ERRATUM,[1] as an holistic approach to the locator repair problem 3, this paper explores its extension to address the robust locator problem. More specifically, we report on the integration of ERRATUM in an open-source test automation solution called CERBERUS [17].[2] CERBERUS is commonly adopted by several companies to write, organize, and run their test campaigns. In particular, we worked in close collaboration with the web testing team of *La Redoute*,[3] a large (846M turnover, 1,700 employees) french online fashion retailer.

The teams we worked with acknowledged that locator breakage is a painful issue for them that even caused a whole test campaign to be canceled. Yet, to the best of our knowledge, none of the locator generator or locator repair solutions existing in the literature have ever been integrated into an open-source testing software. As a matter of assessment, we observed that the concept of locator repair was unknown to all professional testers we interviewed.

The remainder of this chapter is organized as follows. Section 4.2 introduces the necessary background on web locators. Then, Sections 4.3 and 4.4 overview ERRATUM and CERBERUS, respectively, while Section 4.5 reports on the integration of a robust locator solution in CERBERUS based on ERRATUM. Section 4.6 discusses the preliminary impact of this integration on the software development projects managed

[1] ERRATUM stands for "*rEpaiRing bRoken locATors Using tree Matching*"
[2] `https://cerberus-testing.com/`
[3] `https://www.laredoute.com/`

by *La Redoute*. Finally, Section 4.7 presents some perspectives for this work, while Section 4.8 concludes.

## 4.2 Background

### 4.2.1 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 with 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 build such test scenarios, a software engineer can 1. manually write web test scripts to interact with the application, 2. use record/replay tools [11, 75, 60] to visually record their scenarios, or 3. adopt low-code test libraries to leverage the description of actions for domain experts. In all these cases, the scenario requires identifying 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 web 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* [46].

In practice, automated tests are often subject to breakages [28]. While there can be many causes to test breakage, Hammoudi *et al.* [28] report that 74 % of web tests break because one of the included locators fails to locate an element in a web page.

### 4.2.2 Web Locator Terminology

When a locator is defined, manually or automatically, it is written from and for a given web page. The internal representation of a web page by the browser is a *Document*

*Object Model* (DOM). In this chapter, we adopt the following notations:

- we denote $D$ the original DOM of the web page for which the locator was written and $D'$ an evolution of $D$ on which the locator is expected to work,
- we note $e \in D$ an element of the DOM $D$ and $e' \in D'$ the corresponding element in $D'$,
- we note $loc_{e,D}$ a locator that identifies $e$ in $D$,
- then by construction, evaluating this locator on $D$ returns the original element $e$: $eval(loc_{e,D}, D) = e$,
- finally, we assume there is a locator breakage if $eval(loc_{e,D}, D') \neq e'$.

The robustness of a locator can be characterized by its capability to keep selecting the element $e$ on any mutation of $D$. In the following section, we, therefore, report on how we integrated Erratum—an effective solution to the locator repair problem—into the Cerberus web testing framework to implement an elegant solution to the robust locator problem, thus addressing a major concern of web testers.

## 4.3 Repairing web locators with Erratum

Erratum was developed as a locator repair solution. Given a locator $loc_{e,D}$ broken on a new page $D'$, Erratum allows to relocate $e$ on $D'$—*i.e.*, to find $e' \in D'$. Formally, a locator repair solution, like Erratum, is defined as a function: $(D', loc_{e,D}) \to e'$.

Erratum differs from other locator repair solutions by using a holistic approach: all the elements from $D$ and $D'$ are matched by an efficient tree matching algorithm, and the resulting matching is processed to relocate all elements from $D$ in $D'$ 3.

**Example 4.3.1.** *Given the Google search page $D$, the send button $e$ is located by the XPath locator $loc_{e,D} = /html/body/div[3]/button$. Sometime later, the page may evolve to $D'$. Evaluating $loc_{e,D}$ on $D'$ yields no result as the button is now in a wrapped inside a* `form` *tag. In such a situation,* Erratum *can automatically repair the broken locator $loc_{e,D}$ by detecting that the matched element $e \to e'$ is a now child of node* FORM*, hence located with $loc_{e',D'} = /html/body/div[3]/form/button$.*

The core of the Erratum approach the Similarity Based Flexible Tree Matching (SFTM) described in chapter 2. Given two trees $D$ and $D'$, the SFTM algorithm allows for the creation of a mapping between all elements $e \in D$ and $e' \in D'$.

Erratum then uses this mapping to relocate all locators defined in $D$ in $D'$.

While many tree-matching solutions have been developed, we designed SFTM to specifically match complex web pages with fast response times (below 100ms) and

accurately. To do so, SFTM relies on a distinctive characteristic of DOM trees: the *labels*, which usually contain a lot of information (attributes, attribute values, tags, and inner text).

SFTM thus relies mostly on the labels of the trees and only makes use of topology in a second step, to fine-tune the estimated matchings. Intuitively, matching two sets of labels is significantly easier than trying to match trees, which is the reason why SFTM achieves such competitive performance.

## 4.4 Building test cases with Cerberus

Cerberus is a low-code open source scalable test automation solution. It is used by several large companies to create, organize, and run their test suites. Cerberus contains many features, including parallel test execution, test requirement management, and reporting. We integrated Erratum in one of the core components of Cerberus: the test case creation.[4]

A test case in Cerberus describes a sequence of *actions* to be executed. Each action has a type and specific arguments that match this type. Figure 4.1 depicts the action editor of Cerberus. For most action types, the main required argument is the *locator* ("element path" in the screenshot)—*e.g.*, if the action is "click", the locator refers to the element on which Cerberus should click. The action "form" allows the tester to specify the locator using element properties, like *id*, *class*, *name*, or use more expressive query languages, like CSS or XPath. Nonetheless, there are mainly two drawbacks to such basic queries:

- defining the robust locator of an element is a tedious and often arbitrary task,
- tests tend to quickly break because the underlying locators break.

It appears that these drawbacks are more than mere inconvenience. When first meeting with one of the companies interested in Erratum, we were told that a whole test campaign had to be discarded because the locators included in the considered test cases were constantly breaking during the software development process.

To overcome these problems, Cerberus standardized the use of *data-cerberus* identifiers, which are simple element attributes aimed at uniquely identifying web elements to be used by automated tests. For example, if developers anticipate that the web testing team needs to interact with a specific application button, they should include such an identifier as follows:

[4] `https://github.com/cerberustesting/cerberus-source/issues/2252`

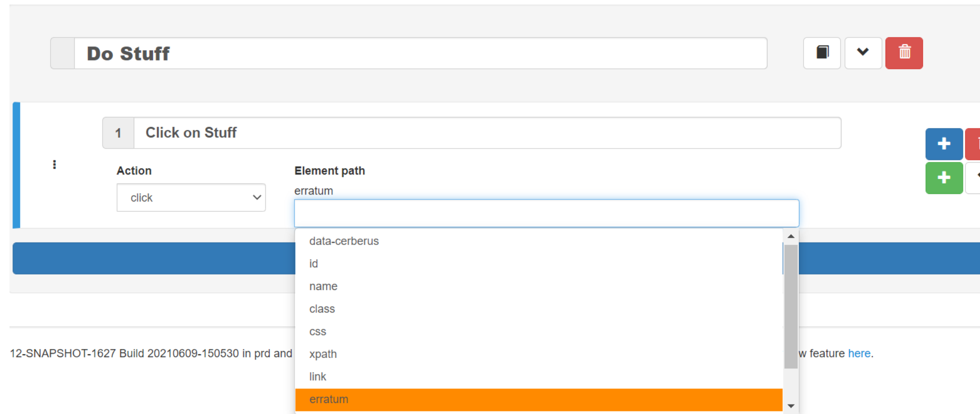



Figure 4.1: A screenshot of the CERBERUS web interface to define a test case.

```
<button data-cerberus="a_special_button" type="button"
    >Send</button>
```

Unfortunately, using such static identifiers entails other challenges, among which:

1. it introduces a—probably undesired—tight coupling between the development and testing teams,
2. if a `data-cerberus` attribute is missing, the development team should first fix the page under test and then re-deploy the web application, which may take time and several iterations to expose all the required identifiers,
3. for various reasons, the testing team might not have the possibility to change the source code of the page under test (*e.g.*, proprietary source code), and finally
4. it forces web developers to anticipate and maintain all these unique identifiers to ensure the absence of identifier collision.

The above observations shared by testing practitioners from *La Redoute* and CERBERUS demonstrate that none of the standard locators support a robust execution of their test campaigns. In the best case, they reported that broken locators causing the failure of a test campaign may require some hours to be fixed. In the worst case, they even mention that some test campaigns were canceled and discarded due to the fragility of web locators, thus causing recurrent breakages of test executions and a prohibitive maintenance cost. Unfortunately, *data-cerberus* identifiers failed to be applied in this specific case, due to the proprietary source code used by the web application under test. In this context, the integration of ERRATUM in CERBERUS offers a practical alternative to standard web locators, thus aiming to drastically reduce the cost of broken locators for web testing teams.

## 4.5 Integrating Erratum into Cerberus

Given the maturity of the Cerberus open-source project, the integration of Erratum has been achieved in several steps that we report in this section.

### 4.5.1 Preliminary Demonstration of Erratum

Since integrating a solution like Erratum requires a non-negligible amount of work, we decided to start by sharing a demonstration of Erratum's core matching algorithm to encourage testers to assess some of their typical test breakages before starting to integrate.

Given an element $e$ in a DOM $D$, the broken locator problem happens when the locator that located $e \in D$ does not return any results on $D'$. In this scenario, the key idea of Erratum is to match all elements from $D \rightarrow D'$ and use this matching to locate $e \in D'$.

Figure 4.2 reports on a screenshot of the demo application. To use the application, one simply needs to:

*i)* *Input* the URL of 2 web pages to match (*e.g.*, two different product pages or two versions of the same product page),

*ii)* *Hover* over one of the elements on either page. The corresponding element matched on the other page is automatically highlighted. If no element matches, the input element is highlighted in red,

*iii)* *Check* that the matched element exists and assesses it matches the expected one.

The application also allows users to input HTML files, which allowed them to test the matching algorithm on several alternative versions of pages. This demonstration step constituted a crucial step of the integration process by convincing the Cerberus community of the benefit brought by Erratum with regards to existing support for web locators. Practitioners quickly understood that the algorithms of Erratum 3 could automate most of the situations they were facing during the web testing campaigns. Furthermore, the fact that Erratum preserved the separation of concerns between the web application and the testing campaign emerged as another convincing argument in favor of Erratum. Once validated, we moved forward by studying the integration strategies to implement these research results into the open-source platform.

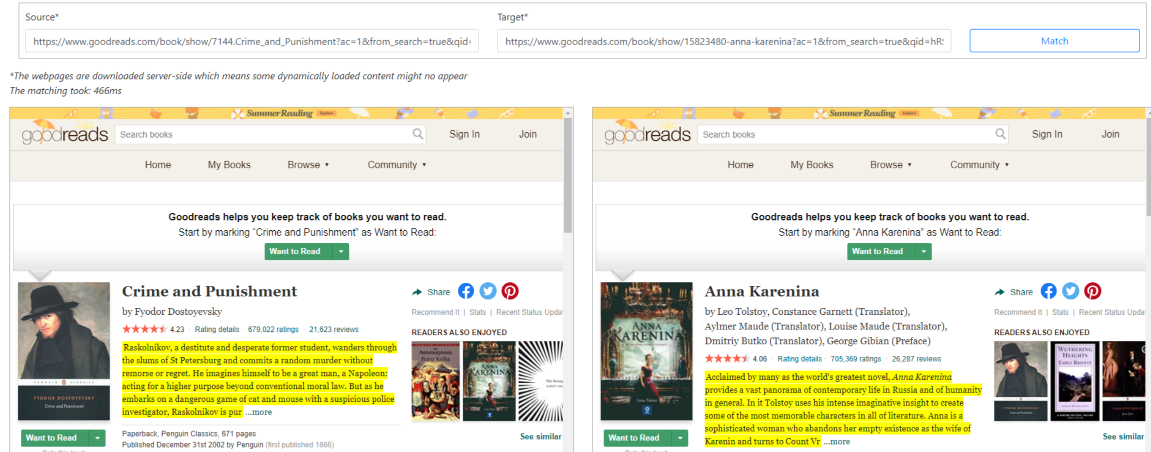



Figure 4.2: The demo application was developed to test ERRATUM on real use cases before starting integration. In this example, the description of a book hovers on the left-side webpage, which automatically highlights the matched element on the right-side webpage.

## 4.5.2 Integration Strategies in CERBERUS

Two main strategies were investigated: *(i)* use ERRATUM to automatically repair broken locators by fixing automated tests, or *(ii)* use ERRATUM directly in the CERBERUS testing environment, as an alternative way to locate elements.

The first approach reflects the case study considered in the original ERRATUM article 3 The main benefit of this approach is that it can be seamless for the tester: whenever a locator breaks, it is automatically repaired without requiring any action from the tester. However, integrating such a solution implies some significant changes in the CERBERUS architecture. Indeed, given a locator $l = loc_{e,D}$ that identifies an element $e \in D$, if $l$ breaks on $D'$, then ERRATUM requires to recover the original page $D$ on which the locator was initially working. Since the history of pages is not recorded by CERBERUS, the repair approach would require some major structural changes to keep track of web application histories.

As a first iteration, we, therefore, chose the second strategy. Instead of using ERRATUM as a locator repair method, we consider its extension as a robust locator method to replace standard locators, like *CSS* or *XPath*, as described in Section 4.3. Interestingly, the existing support for multiple classes of web locators in CERBERUS offers a more straightforward hook to integrate ERRATUM.

### 4.5.3 The ERRATUM Robust Locators

To leverage the integration of ERRATUM in CERBERUS, we, therefore, packaged ERRATUM as a robust locator. The key property that differentiates ERRATUM from other web locators is that ERRATUM requires the original DOM $D$ on which the element was successfully located. To build a robust locator with ERRATUM, we embedded the whole DOM $D$ in the locator—*i.e.*, the ERRATUM locator of an element $e \in D$ is the pair $(XPath(e), D)$, where $XPath(e)$ is the absolute XPath of $e \in D$. Figure 4.3 depicts how we extended ERRATUM to implement to robust locator support in CERBERUS:

1. from the original page $D$, the tester selects the element to locate $e$,
2. so far, with XPath:
   (a) the tester specifies the web locator as an XPath query that she thinks to be robust to select $e \in D$,
   (b) during the test execution, the XPath query is executed on the new DOM $D'$. If the locator is not broken, the query evaluation returns $e'$, but if the XPath query fails, then the test scenario is likely considered broken,
3. while, with ERRATUM:
   (a) the web locator is automatically generated, as a pair $(XPath(e), D)$, where $XPath(e)$ is the absolute XPath of $e$ and $D$ is the full DOM,
   (b) during the test execution, ERRATUM first matches all the elements of $D'$ with $D$, and then the resulting matching is used to relocate the matched element $e \in D$ into $e' \in D'$.

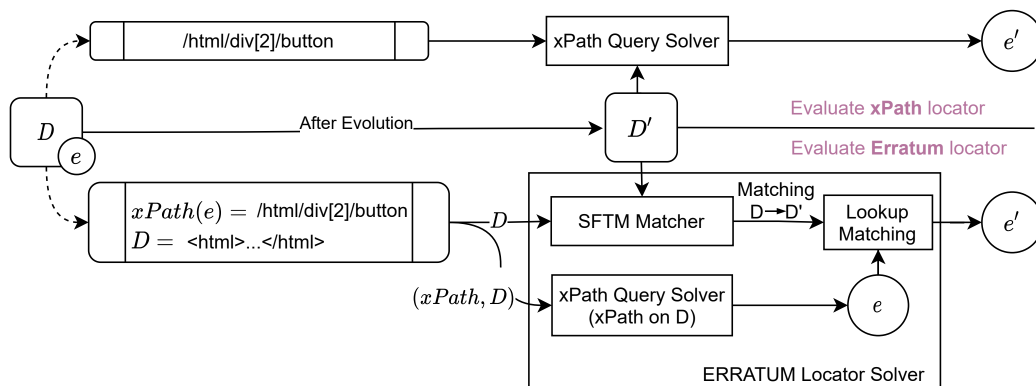



Figure 4.3: Describing how ERRATUM is integrated as a robust locator in CERBERUS. Both locator types originally locate an element $e \in D$. The figure illustrates the ways CERBERUS evaluates XPath or ERRATUM locators on a new DOM $D'$.

To allow this integration within CERBERUS, we rewrote the generic matching

algorithm of Erratum, namely SFTM 2, in a language that can compile to JVM. The code of the resulting library is open-source[5] and the library is published in Maven central.[6] Then, the integration of the Erratum library in the Cerberus platform represents only 127 lines of Java code.[7]

### 4.5.4 Usage

The usage of the new Erratum locator on Cerberus is very similar to that of other locators. Given an element $e$ that the tester wants to select on page $p$, she has to:

1. select the Erratum locator type on the test case select box (see figure 4.1), and then
2. copy-paste the absolute XPath of $e$ and the whole HTML of $p$ in the input form.

To ease this process, we developed a browser extension allowing you to simply click on the targeted element $e$ to copy the required data in the clipboard before pasting it into the input form.

In addition to providing more robust locators, writing a test scenario using Erratum along with its extension represents a significant time gain: instead of manually crafting a locator that the tester judges to be robust enough, she simply needs to click on the targeted element. It allows the tester to completely ignore the details of how a certain element should be located and instead focus on building the most relevant test scenarios.

## 4.6 Industrial Impact

Every year, the *La Redoute* web testing team estimated that the locator breakage entailing the failure of their test campaigns induced more than 11 K€ worth of load to their teams. For industrial projects that evolve quickly, some testing campaigns were even canceled because of the cost of continuously repairing the broken test scripts. In addition to that, repairing broken locators was considered a tedious low-value task, which may be harmful to the morale of testers. We collected the feedback of 2 testers from this team, mainly asking about the biggest pain points faced by the testing team and how a solution like Erratum could help. According to the most experienced tester we interviewed, most challenges faced by the test team stem from

[5] https://github.com/lssol/sftm_tree_matching

[6] https://mvnrepository.com/artifact/io.github.amaris/sftm-tree-matching

[7] https://github.com/cerberustesting/cerberus-source/commit/0a70d4cc0d70a797901652fd2b97d501bb7fa511

| | **Campaign A** | **Campaign B** |
|---|---:|---:|
| # Test cases | 16 | 3 |
| # Releases | 16 | 2 |
| # Locators / test (avg) | 18 | 22 |
| # Relocations | 4,608 | 132 |
| [XPath] # Locator Failure | 0 | 1 |
| [ERRATUM] # Locator Failures | 0 | 0 |

Table 4.1: Results of the third testing phase that lasted one month.

the separation between the development team and the testing team. The development team is encouraged to push features fast (pressure on quantity), while the testing team's role is to monitor the quality of the released application. The problem is all the more aggravated by the fragility of the developed test cases: when a new version is released, it takes several days to fix all tests which means that when the tests are ready, the developers are already working on different features. Hence, using a tool like ERRATUM helps tighten the feedback loop between the testing team and the development team, thus allowing more intricate collaboration and less friction.

*La Redoute* unfortunately does not conserve a history of failed tests (along with the web page versions), which makes it hard to quantitatively measure the exact percentage of successful relocation of ERRATUM on their industrial cases. We collaborated with 5 testing experts working on different campaigns to evaluate ERRATUM. The purpose of this evaluation was mainly to assess if ERRATUM could be used systematically in place of manually written XPATH to locate elements on new test cases, which would significantly fasten their development and maintenance. We tried to design evaluation strategies that would not require too much time from the testing experts. In particular, there were three stages of testing:

1. Each tester tested 2 to 3 typical pages from their testing campaigns on the demo described earlier (see section 4.5.1);

2. After ERRATUM was integrated to CERBERUS, the ERRATUM locator type was tested on the same pages within the CERBERUS framework;

3. For two campaigns, each new test case written by the testers was duplicated for one month: one version used an XPath locator, and the other version, the ERRATUM locator.

The evaluation of ERRATUM attempted to answer two questions:

1. Is ERRATUM a viable locator replacement of XPath?

2. Are ERRATUM locators more robust than manually written XPath locators?

Table 4.1 presents the results of the third testing phase. Campaign A applies to an application being completely rewritten, which explains the high frequency of releases.

During a month, there were $4,740$ locator relocations (# tests × # releases × average # locators / test) over the two campaigns under study. For ERRATUM, none of these relocations failed, while one failed for XPath.

The results thus tend to indicate that ERRATUM can be safely used as a replacement of manually written XPATHS. As for whether ERRATUM provides more robust locators than manually written XPath, there is still not enough data to conclude.

Following the evaluation, *La Redoute* now adopted ERRATUM as the preferred locator method, along with `data-cerberus` attributes, and have not yet reported any relocation failure at the time of writing this manuscript. Beyond the collaboration with *La Redoute*, we believe that these promising results will benefit the wider CERBERUS community.

## 4.7 Perspectives

**Adoption** While we specifically studied the integration of Erratum to Cerberus for *La Redoute*, Cerberus is used by several more major online stores. It has GPL licence and once the integration is mature, the Cerberus team will communicate to all its users to allow for a broader adoption.

**Visual integration** considering the above ERRATUM web extension, the tester no longer needs to analyze the HTML source code and manually assemble a robust locator. It means that, theoretically, CERBERUS could integrate a fully visual locator selection approach where *(i)* the tester fills in the web application URL, *(ii)* the page opens in an iframe, *(iii)* the tester selects the target element, *(iv)* the ERRATUM locator is automatically generated. This process was originally considered, however, it would not be usable in more complex situations where some pages are not directly accessible from a given URL (*e.g.*, a page that requires login first).

**Mobile testing** the web testing teams in the *La Redoute* company also use CERBERUS to automate tests from mobile devices. Web pages are described using HTML and Android views are written in XML. While HTML and XML are very similar and our tree matching library theoretically allows ERRATUM to match any kind of tree (and not only HTML trees), some technical adjustments still need to be completed

to include XML trees in Cerberus. Furthermore, the robustness of Erratum's locators to typical Android XML view mutations still needs to be assessed, as they may differ from typical HTML mutations.

## 4.8 Conclusion

Web locators remain a fragile keystone of automated test suites executed by modern test platforms. Whenever a locator fails to locate a web element in a test suite, it directly impacts the whole test campaign and imposes some additional maintenance tasks on the web testing team.

In this section, we described how we extended and successfully integrated the locator repair solution Erratum into Cerberus, a widely used open-source test-automation solution. In particular, we reported on the development of a robust locator based on Erratum that eases the pain of web testers, by saving time and reducing the cost of test campaigns.

Through a thorough benchmark and the integration of Erratum to a largely used open-source testing framework, we showed that we are capable to build robust locators on any web application. In the following section, we extend our work to build locators that are not only robust but also *semantically rich*.

# Chapter 5

# APPSTRACT


## Summary

Web applications are at the cornerstone of modern society. Web applications are made for humans, yet in many situations we need to automate or monitor interactions with these applications. Such situations include web data extraction, web analytics or web testing.

As soon as the need to scale a solution on multiple web applications in the wild arise, state-of-the-art research in web-related fields often require clever heuristics to compensate for the lack of abstract model. We argue that a considerable amount of research in a variety of web-related fields can hugely benefit from a universal abstraction inference solution.

We introduce APPSTRACT, a solution allowing to infer an abstraction model on any web application with almost no human intervention.


## 5.1 Introduction

Web applications are often perceived by end-users as online systems exposing a limited set of views, concepts, and related actions. From a user perspective, an online shop—like Amazon—mainly offers to search and list products, while clicking on a given product brings us to its details view. From a machine perspective, like a bot, navigating the same web application will be perceived as crawling thousands of unique pages, exposing unrelated content and seemingly unique actions.

This inability of a machine to understand the navigation model of a web application, like a user intuitively does, makes it very hard to automate interactions with

web applications. Typically, research topics benefiting from more intuitive navigation include: *(a) web data mining* to automatically extract, de-noise, and structure data from web pages, *(b) web testing* to generate, maintain, repair, and augment tests on web applications, and *(c) web analytics* to report on how users interact with the web application.

While there is a considerable amount of existing literature, notably in the fields of data mining [92, 4, 18, 72, 14, 59], most of the existing works are highly specific and do not provide adequate insights to build an application-wide understanding of a navigation model.

In particular, state-of-the-art approaches focus on either extracting data within a page (so-called, *intra-page* extraction) or across two pages (*inter-page* extractions).

For example, Miao *et al.* [59] clusters full tag paths—*i.e.*, `/html/body/div[2]/em`—to detect repeating occurrences of a template within a page. This allows their approach to extract what is usually called *records* in a web page (*e.g.*, a single product card in a product list). This is an example of what we categorize as *intra-page abstraction*.

*Inter-page abstraction* solutions usually try to detect templates of a webpage or to learn a wrapper by studying two different instances of the same template. That is what is done by the state-of-the-art algorithms, like EXALG [4] and RoadRunner [18]. However, none of these approaches provide any insight to take into account the intra-page variability.

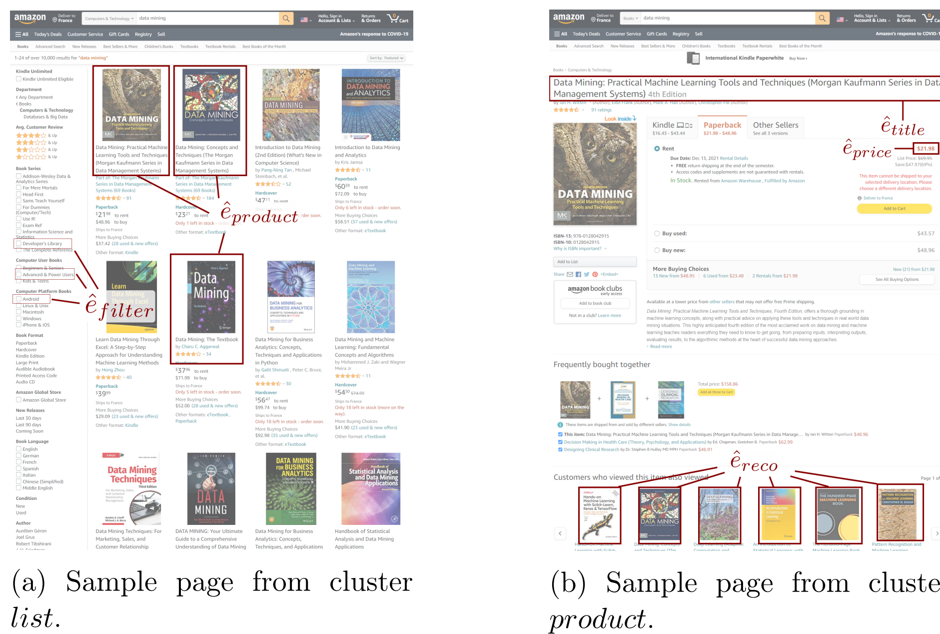
In this chapter, we thus propose an unsupervised approach to infer the navigation model of a whole web application that we call an *appstraction*. The *appstraction* of a web application aims at delivering actionable insights: it allows a machine to understand application states (*e.g.*, product pages, blog page), as well as elements within states (*e.g.*, product title, price), hence guiding the information extraction process, as well as identifying relevant navigation actions. The *appstraction* process aims to abstract away the natural variability of web pages into a canonical model built as a compact tree of *template pages* and *template elements*. Our unsupervised approach assumes that even web applications with billions of pages will build on a limited set of template pages, thus making it possible to infer these generative templates from a dataset of visited pages. To achieve this, our approach—named Appstract—builds on three stages: 1. *web page clustering* to group instances of related web pages, 2. *intra-page abstraction* to extract repeating patterns within each cluster of pages, and 3. *inter-page abstraction* to capture repeating patterns across clusters.

We empirically demonstrate that our *appstraction* succeeds to generate application-wide locators that can be used to support semantic guidance across multiple pages of any web application.

The remainder of this chapter is therefore organized as follows. Section 5.2 introduces the required background and related works in this area. Section 5.3 presents the design and implementation of APPSTRACT, while Section 5.5 reports on an evaluation of the perceived accuracy of APPSTRACT. Finally, Section 5.6 concludes.

## 5.2 Background & Related Works

In this section, we present several studies across different research fields that offer partial solutions to the general problem of web application abstraction inference. Throughout this chapter, we consider a subset of the Amazon web application as a running example. In particular, we focus on two related page templates: the product details and product list pages (cf. Figure 5.1).




(a) Sample page from cluster *list*.

(b) Sample page from cluster *product*.

Figure 5.1: Screenshots illustrating some elements $\hat{e}_\lambda$ included in the template pages $\hat{p}_{list}$ and $\hat{p}_{product}$ of 2 distinct clusters.

### 5.2.1 Data Extraction

The role of a web page is mainly to *present* a subset of the information it has access to in a certain way. All data extraction solutions attempt to separate the information

from its presentation—*i.e.*, the template. The process of data extraction is thus a form of abstraction of an application: instead of viewing the different pages as a multitude of unrelated blobs of HTML, the application is seen as a limited set of templates that consistently presents various information. The process of information extraction is thus closely related to our *appstraction* objectives and our approach is highly inspired by the ideas behind data extraction.

Following the classification made by a survey on data extraction [12], most existing literature on data extraction can be classified according to the levels of supervision needed to extract the data:

- *(semi-)supervised approaches* where the user inputs more or less detailed directions to describe to how to extract data [7, 32, 53, 62, 23, 24, 31],
- *unsupervised approaches* where data is automatically extracted by analysing recurring patterns [18, 4, 55, 84, 84].

Supervised approaches are usually qualified as "wrapper-based". The idea of wrapper-based data extraction is to consider the set of web pages containing the data to extract as an unstructured (or semi-structured) database and build a query language (and its query engine) to query the desired data.

In this chapter, we focus solely on *unsupervised approaches*. These approaches rely on the assumption that even though there are a lot of different pieces of information exposed by an application, the same type of information will always be structured in the same way. For example, on the product list page of an online shop, each product will be structured in a similar fashion (*e.g.*, product name, price, description). In this context, we distinguish two families of unsupervised approaches: *intra-page* and *inter-page* data extraction. One should note that *intra-page* data extraction is usually referred as *record extraction* [12, 19].

*Intra-page data extraction* refers to the extraction of data within a page. For example, in the case of the Amazon page presented in Figure 5.1a, an intra-page data extraction solution, such as MDR [55], relies on the topology of the DOM tree and string matching to detect data regions $\hat{e}_{product}$ within a single page. As is always the case when attempting to extract data in an unsupervised way, MDR can only detect data regions if at least two of these regions are present on the page. This is a necessary condition since all unsupervised algorithms must rely on some kind of pattern discovery to detect data regions.

*Inter-page data extraction* refers to the extraction of data across several pages. In particular, all existing algorithms apply to pages that are assumed to belong to the same template (*e.g.*, two Amazon product pages). The idea is then to use the simi-

larity between the two pages to understand what is common and what has changed between the two pages. The parts that changed are then assumed to be data, while the common part has to do with its presentation (*template*). To compare these pages, one solution [19] uses a modified version of the most popular general tree matching solution: *tree edit distance*. In the inter-page abstraction part of our approach, we also use a tree matching solution but not the same one.

The challenge of data extraction presented above is very similar to that of our *appstraction* objective. However, it differs in a few important points:

1. while data extraction focuses on extracting data, we try to abstract any kind of variability. For example, in Figure 5.1a, a data extraction solution should not attempt to extract the $\hat{e}_{filter}$ template elements, as they do not represent data.

2. several data extraction studies are focused on how to infer a schema of the extracted data (*e.g.*, relational schema [4, 63])), which we are not interested to do in the context of *appstraction*, and

3. most importantly, to the best of our knowledge, no previous work has attempted to extract data throughout a whole web application (combining both inter-page and inter-page abstraction).

### 5.2.2 Web Testing

Web testing covers a large body of research that encompasses various themes, such as test generation, test coverage, or test robustness. We claim that, essentially, the main challenge of web testing comes from the lack of abstract understanding of the application under study and most works in this field have to resort to ingenious ideas to compensate for this lack of an abstract model.

**Test Robustness** One of the main challenges of web testing encountered by the industry is test breakage: tests written for a given version of a web application break when the application evolves. For example, let us consider a testing script that applies to a web page $D$. One part of the script instructs to click on a given button $e$ on the page $D$. To locate $e$, scripts usually use xPath or CSS selector to build what is called a *locator* $l_e$. Breakage can then happen when a new version $D'$ of the page $D$ is published. Most of these breakage are locator based [29]: the locator $l_e$ that successfully located $e$ on $D$ does not locate the matching element $e'$ in $D'$. To solve this problem, some attempt to automatically generate more robust locators relying

on the structure of the tree [50, 45, 88, 6] while other works offered solutions to repair broken locators 3.

In particular, The ERRATUM approach 3 achieves high accuracy specifically thanks to a more holistic stance: instead of looking at individual locators independently, the approach considers the page as a whole using a similar technique to the *inter-page abstraction* part of our *appstraction*.

The "locator" problem is a more specific instance of the web application abstraction: the breakage happens because when a machine sees a multitude of different pages (*e.g.*, different versions), a human perceives a single page template with slight differences and thus comes to expect the locators on $D$ should naturally work on $D'$.

In the present work, we go beyond ERRATUM 3 on the generalization scale and create "locators" that are valid through all pages of a web application.

**Test Generation** Automatic test generation is a domain of research that studies different approaches to automatically generating tests. Most test generation techniques rely on an abstraction of the application called a navigational model. A navigational model of a web application describes the different states an application can be in and the transitions between them [57].

There exist many approaches to take advantage of this navigational model. The goal of these techniques is to generate the minimum amount of tests that covers a maximum of the application's behavior given a navigation model [52, 57, 90, 8]

We are particularly interested in the generation of the underlying navigational model. The navigational model can be written manually, for example, Leveau and al. [52] developed a whole Domain Specific Language to describe this navigational model. It can also be generated. The only approach we managed to find that generates a navigational model is APOGEN [76]. APOGEN is quite close in spirit to our APPSTRACT solution. The idea of APOGEN is not originally to create a navigational model but to create *page objects*. However, it has been used as a way to generate a navigational model [8]. APOGEN crawls, clusters, and then analyses the different clusters of a web application to generate page objects. The main difference with our approach is that APOGEN is heavily based on the analysis of links to understand how different pages interact with one another and most importantly, it does not include any *intra-page abstraction*.

### 5.2.3 Web Analytics

Web analytics deals with the analysis of the user behavior on a given web application. A lot of studies have been published on the subject since the popularisation of the web. However, most works are applied in controlled environments where the ability to automatically infer an abstract model of an application is not needed. For example, some research focuses solely on mouse movements [26, 86] or focuses on a given search engine and tries to analyze behavior to optimize search results [2, 1]. Research that does rely on a certain model of an application usually uses an *url-based* model. While the first research works on the subject, at the beginning of the web, could use simple logs of visited urls [25, 61, 73], modern web applications are generally much more complex with virtually unlimited amounts of URLs dynamically generated making it hard to reason about unprocessed logs of visited URLs. To tackle this challenge, recent work on web browsing behavior uses a variety of techniques to encode URLs like Recurrent neural network [66] so that several strictly different URLs that refer to the same template page will have similar encodings. While using URL-based encoding to create an abstract model of a web application can help abstract relationship *between* pages (given that they contain enough information which depends on the application), it will not provide the *intra-page* abstraction that we need to obtain meaningful encodings within pages.

## 5.3 Appstract

### 5.3.1 Abstracting a Web Application

This section starts by defining the notions of *application* and *application abstraction* we assume, before discussing the challenges of inferring such defined abstractions.

**Definition 5.3.1** (*Application*)**.** *A web application $A$ is a set of web pages $\{p \in A\}$, where every page $p$ is captured by a* Document Object Model *(DOM).*

The number of web pages $\|A\|$ can be very high and keep rising with time (*e.g.*, Amazon products or blog posts). In addition, we consider that any mutation of a web page (even if the URL does not change) is considered a new page. Obviously, from a human perspective, a web application is much more than a collection of pages (*e.g.*, the pages are linked, the application has different features usable by different users), but we choose to take the perspective of a machine, for which an application is perceived as a set of visited web pages. As a DOM of a page $p$ is a tree, we

represent it as a tuple $\langle N(p), par\rangle$ where $N(p)$ is the set of elements (or nodes) in $p$ and $par : N(p) \rightarrow N(p)$ is the parent function that associates a DOM node with its parent. To lighten the notations, we write $e \in p$ to describe an element $e$ in a page $p$, instead of writing $e \in N(p)$.

**Definition 5.3.2** (*Application abstraction*)**.** *The abstraction of a web application $A$ is a tuple $\langle \hat{A}, T_{\hat{A}}\rangle$ where* i) *$\hat{A}$ is the set of template pages and each page $\hat{p} \in \hat{A}$ contains template elements $\hat{e} \in \hat{p}$ and* ii)*:*

$$\begin{gathered} \forall p \in A, T_{\hat{A}}(p) = \langle \hat{p}, T_{\hat{p}}\rangle \\ \textit{and } \forall e \in p, T_{\hat{p}}(e) = \langle \hat{e}\rangle \end{gathered} \tag{5.1}$$

In other words, the abstraction of a web application $A$ is a set of template pages (*e.g.*, product page, list page) and a function that allows mapping any page from $A$ to its corresponding template, and every element within the page to the corresponding element in the template page. This mapping is completed by two functions:

1. $T_{\hat{A}} : A \rightarrow \hat{A}$ is the template function that takes any page $p \in A$ and returns the matched *template page* $\hat{p} \in \hat{A}$, and

2. $T_{\hat{p}} : N(p) \rightarrow N(\hat{p})$ is a function that takes any element $e \in p$ and returns the matched element in the template $\hat{e} \in \hat{p}$.

Additionally, we also use:

- $T_{\hat{A}}^{-1}(\hat{p}) \subset A$ as the set of pages $p \in A$, such that $T_{\hat{A}}(p) = \hat{p}$, and

- $T_{\hat{p}}^{-1}$ as the inverse function of $T_{\hat{p}}$.

These notations allow us to easily refer to the instance pages/elements of a given template page/element.

Figure 5.1 depicts a theoretical application of the appstraction to 2 web pages crawled from Amazon. For each related clusters of similar web pages, the $T_{\hat{A}}$ function will return either template page $\hat{p}_{list}$ or $\hat{p}_{product}$. Then, for each clustered web page, each web element can be mapped to its corresponding template element (*e.g.*, $\hat{e}_{title}$, $\hat{e}_{product}$) using the $T_{\hat{p}}$ function.

Please note that our *appstraction* process does not intend to label template pages or elements. Thus, our references to $\hat{p}_{list}$ or $\hat{e}_{product}$ should be understood as a *global unique identifier* (GUID) capturing a class of pages and elements within and across clusters.

### 5.3.2 Building an Abstraction

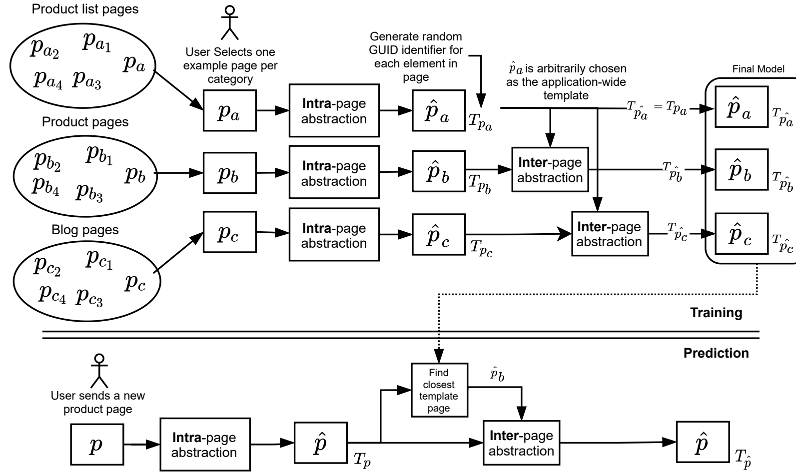



Figure 5.2: Overview of the two stages of the *appstraction* process: *learning* and *prediction*. The identifier map of $p$ noted $T_p$ associates an identifier to each element of $p$. The identifiers are first randomly generated, then merged to existing maps during inter-page abstraction. The prediction phase produces an identifier map $T_{\hat{p}}$ in which each original element of $p$ is associated with an application-wide identifier.

To abstract the web page $p$ into its *appstraction* $A_p$, we operate at two levels:

1. *intra-page abstraction* to extract repeating patterns of elements within a cluster of web pages, and
2. *inter-page abstraction* to group repeating patterns of elements across template pages.

This bi-level abstraction process combines two steps: *learning* and *prediction*, as depicted in Figure 5.2. In this section, we first overview the abstraction process by considering intra- and inter-page abstractions as black boxes. In the following sections, we provide an in-depth description of individual-level abstraction techniques.

**Learning** The objective of the learning phase is to build a model of the application that will deliver accurate predictions. The upper part of Figure 5.2 describes how intra- and inter-page abstractions are used to build this model.

Before any abstraction, the application is perceived as a set of related pages. The first step is thus to organize these pages into clusters, each cluster representing a template page (*e.g.*, product page). To do this, any clustering algorithm could be used and, in this chapter, we consider the clustering part as out of scope as we focus

on the actual abstraction process. The learning approach thus starts with the user picking one example page per template.

Each template page goes through intra-page abstraction. The intra-page abstraction takes a DOM tree $p$ as input and returns a *template* as output. A *template* is a tuple $t = \langle \hat{p}, T_p \rangle$ where $T_p$ is the identifier map that maps every element from the original tree $p$ to a global identifier. After the intra-page abstraction, $\hat{p}$ typically contains much fewer elements than $p$ since all repeated patterns have been abstracted away. The $T_p$ map is a way to keep track of the elements in the original page $p$ after abstraction. For example, in Figure 5.3 illustrating intra-page abstraction, the identifier map would contain 11 entries, one for each original element in $p$ and 5 distinct values corresponding to the five distinct nodes of $\hat{p}$.

In practice, the values of $T_p$ are *Global Unique Identifiers* (GUID). Before starting the abstraction process, we transform the page $p$ into a *template* tuple, where the identifier mapping maps every node to a randomly generated GUID. Each abstraction step will thus transform the input tree and update the associated identifier map.

Once each page has been abstracted at the intra-page granularity, we build a model allowing us to achieve two aspects of inter-page abstraction:

- *Template abstraction:* same elements within different instances of a template must have the same identifiers (*e.g.*, the title of a product on different product pages)

- *Cross-template abstraction:* same elements within different pages (regardless of templates) should have the same identifiers. (*e.g.*, menu link that appears on all template pages)

To achieve cross-template abstraction, we arbitrarily select one template to represent the whole application ($\langle \hat{p}_a, T_{p_a} \rangle$ in Figure 5.2) we call this template the *mother template*. All other template pages are then inter-page abstracted against the mother template. The inter-page abstraction function takes two templates: a reference template and a query template. It then returns one template in which the values of the identifier map have been updated to match the reference template.

The final model is obtained after applying intra-page abstraction between the mother template and all other pages. The final model is simply a set of $n$ templates where each template is a tuple $\langle \hat{p}, T_p \rangle$.

**Prediction** During the prediction phase described in the bottom part of Figure 5.2, the user sends a previously unseen page and APPSTRACT returns the mapping be-

tween each element $e$ of the page and the $id$ of the associated template element $\hat{e}$.

To do so, APPSTRACT first applies intra-page abstraction to create an abstracted DOM tree $\hat{p}$, then applies the tree matching algorithm at the core of the inter-page abstraction between $\hat{p}$ and all templates in the model. The template page from the model that matches best with $\hat{p}$ is retained (*e.g.*, $\hat{p_b}$ in Figure 5.2).

The matching between $\hat{p}$ and the corresponding template is then used to compute the final mapping $T_{\hat{p}}$.

Overall, the user sent a page and received the mapping between each element of the page and the corresponding template element in the corresponding template page.

In the following sections, we describe both intra-page abstraction and inter-page abstraction in more detail.

### 5.3.3 Intra-Page Abstraction

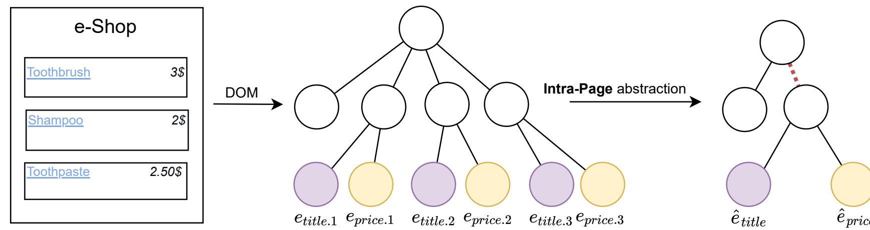



Figure 5.3: Illustration of Intra-Page abstraction. DOM leaves at the end of repeating branches are tagged and then recursively merged.

Intra-page abstraction relies on the detection of repeating patterns within a page.

Building the intra-page abstraction corresponds to creating the $T_e$ function defined above. The intra-page abstraction deals with a single page: given an input page $p$, we want to build a template page $\hat{p}$ and a function $T_e$ that maps all the elements $e \in p$ to their corresponding elements $\hat{e} \in \hat{p}$ Overall, the Intra-Page abstraction is done in three steps:

1. *Leaf clustering* clusters repeating DOM leaves together,

2. *Node tagging* propagates information about leaf clusters in all ancestor nodes to prepare for the final step,

3. *Recursive leaf-group merging* builds the intra-page abstraction by recursively merging branches.

### Leaf Grouping

In the first step of intra-page abstraction, we attempt to identify groups of leaf nodes that present data of the same type. To do so, we: 1. build all root-to-leaf paths from the DOM tree, 2. group all same paths together, and 3. filter out groups of leaves containing less than a fixed threshold $k$ of elements. At the end of this phase, we have a set of leaf groups ($LG$), each containing at least $k$ elements.

The *root-to-leaf* path of a node is the formatted sequence of tags from the root to the node. For example, on the web page from Figure 5.4, the root-to-leaf path of the element containing the text *price2* is `//html/body/div/span/`.

```
<html>
  <head> <!-- header --> </head>
  <body>
      <div> <!-- content --> </div>
      <div>
          <a href="...">Item1</a>
          <span>price1</span>
      </div>
      <div>
          <a href="...">Item2</a>
          <span>price2</span>
      </div>
  </body>
</html>
```

Figure 5.4: Web page example to illustrate intra-page abstraction with no nested records

To find leaf groups, we extract all root-to-leaf paths from the DOM tree and group the same ones together and only keep the groups that contain more than a fixed threshold $k$ of elements. In example 5.4, assuming $k = 2$, it means we have two groups:

1. Leaf group 1 (`//html/body/div/span/`) containing `price1` and `price2`, and
2. Leaf group 2 (`//html/body/div/a/`) containing `Item1` and `Item2`.

In our evaluation, however, this threshold is set to $k = 4$.

**Limits** Our grouping method may fail to cluster items correctly in two situations:

- *Over-abstraction*: Items can be clustered together even if they are not of the same type, or

- *Sub-abstraction*: Items can be put in different clusters even though a human would put them in the same cluster.

To a certain extent, *sub-abstraction* can be compensated afterwards. For example, when extracting information, it is common for websites to structure data differently according to the type of exposed products (*e.g.*, regular products or "sponsored" products). In this case, using the approach we presented, the two types of products will be classified in different clusters, meaning that if the data is extracted as a table, the prices will be spread in two different columns that the user will be able to easily merge.

The cases of *over-abstraction* are more problematic since it will mean that the associated data has been lost when abstracting the page.

### Node Tagging

At this stage, we have a list of all leaf groups ($LG_1, LG_2...$). To be able to recursively merge all leaf groups, we propagate the information that a leaf belongs to a certain group to all ancestors of the leaf using the node-tagging algorithm 3.

**Algorithm 3** Intra-Page abstraction: Node Tagging

```
1: function CREATETAGS(LGs = [LG_1, LG_2...])
2:     function TAGBRANCH(depth, LG, e)
3:         tag ← ⟨LG, depth⟩
4:         e.tags[tag]+ = 1 // Inc or init tag count of node e
5:         tagBranch(depth + 1, LG, e.parent)
6:     end function
7:     for all LG ∈ LGs do
8:         for all e_leaf ∈ LG do
9:             tagBranch(0, LG, e_leaf)
10:        end for
11:    end for
12: end function
```

Algorithm 3 iterates through all the leaves that belong to a leaf group $LG$ and for each leaf, it recursively tags all the ancestors of the leaf. The *tag* of an element $e$ is the tuple $\langle LG, depth \rangle$ where $LG$ is a leaf group and *depth* is the number of nodes between $e$ and the leaf from its offspring that belongs to $LG$. The tag is used to identify nodes that should be merged.

When the algorithm ends, each node $e$ of the DOM tree contains a map *tags* whose keys are tags and values are integers count. Figure 5.5 shows an example result of the node-tagging algorithm on a simple DOM tree.

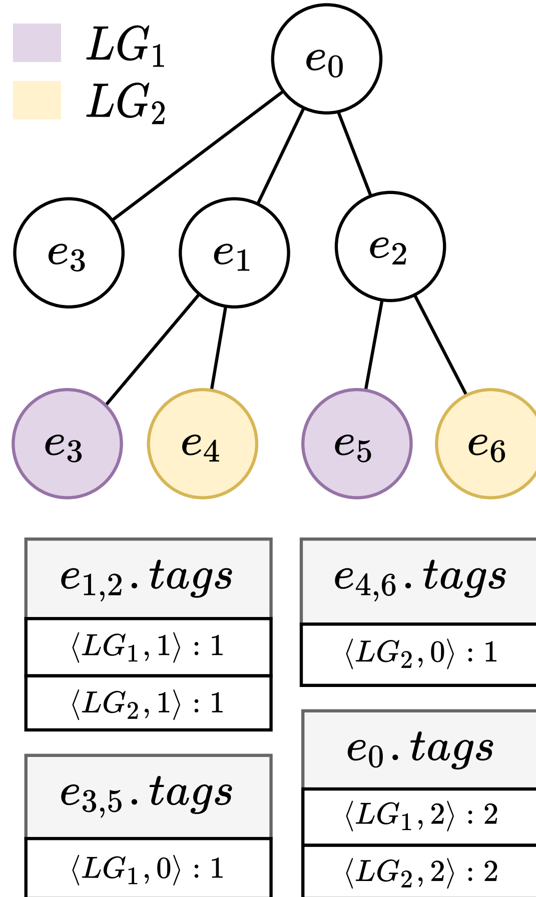



Figure 5.5: Output of the node-tagging algorithm. Each node is assigned a *tags* map keeping track of the number of leaf groups in its offspring.

An important implementation detail: the above algorithm assumes that the *tag* tuple's hash code will be generated from the value of its components (and not the object's reference)—*i.e.*, two *tag* tuples containing the same leaf group $LG$ and the same *depth* should have the same hash code when inserted into the *e.tags* map (even though the tags will have different addresses in memory since they were created at a different stage of the algorithm).

#### Recursive Branch Merging

**Overview** The last step of the intra-page abstraction consists in merging all required nodes such that the resulting DOM contains only one node instance of each leaf group. Figure 5.3 shows an example of the result obtained after this last step of the intra-page abstraction. In the final abstract tree of Figure 5.3, the red dotted edge connecting the template element $\hat{e}$ to its parent indicates that $\hat{e}$ has a special

relationship with its parent, in this case: a 1-to-many relationship.

The algorithm must be recursive because it is possible (and likely) that some leaf groups will have to be merged at different levels of the tree. Figure 5.6 illustrates this use case: $e_{a1}$ and $e_{a2}$ must be merged first before their ancestor $e_{b1}$ can merge with $e_{b2}$.

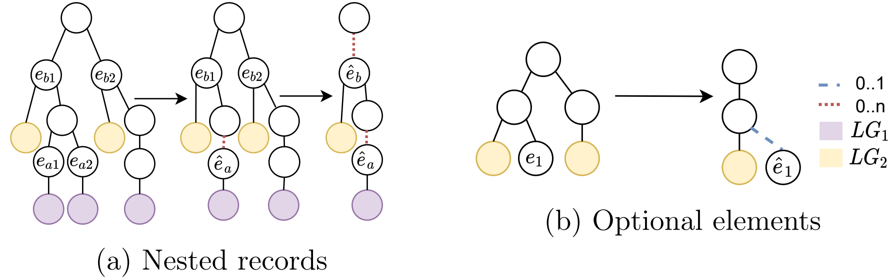



Figure 5.6: Illustrating two important cases our merging algorithm must cover: *nested records* and *optional elements*.

The second case illustrated in Figure 5.6b occurs when merging two nodes even though not all their respective children are merged. In this case, the inferred abstract node $\hat{e}_1$ is said to have an *optional* relationship with its parent template element—*i.e.*, it only exists as a child node in certain instances of $\hat{e}_1$. In general, a template element $\hat{e}$ can have many types of relationships with its parent. In this chapter, we distinguish two relationships:

- A *zero-to-many relationship* when at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ has no child element that is an instance of $\hat{e}$ and at least one other element has more than 1,
- An *optional (or zero-to-one) relationship* when at least one element from $T_e^{-1}(par(\hat{e})) = par(e_1), par(e_2)...$ has no child element that is an instance of $\hat{e}$ and at least one other element has more than 1.

**Algorithm** Algorithm 4 gives a high level view of the solution. The function *abstractTree* recursively traverses the tree. For each node $e$ considered, it checks if $e$ is already abstract using the *isAbstract* function (line 2). A node is abstract if none of its children need to be merged. If $e$ is already abstract then it is returned, otherwise, we:

1. recursively call the *abstractTree* function on all children of $e$,
2. group all abstract children by group of tags using *assocGroup* (see paragraph 5.3.3), and
3. merge each group of children using *mergeGroup* (see paragraph 5.3.3)

**Algorithm 4** Intra-page abstraction: recursive merge

```
1: function ABSTRACTTREE(e : Element)
2:     if isAbstract(e) then
3:         return e
4:     else
5:         children ← e.children.map(abstractTree)
6:         children ← assocGroup(children).map(mergeGroup)
7:         ê ← { e with e.children = children }
8:         return ê
9:     end if
10: end function
```

We describe in details the three functions used in algorithm 4, namely: 1. *isAbstract*, 2. *assocGroup*, and 3. *mergeGroup*.

**isAbstract** The function *isAbstract* checks if an element $e$ is already abstract. An element $e$ is abstract if none of its offsprings need to be merged. This information is obtained using the tags computed in previous steps (see Section 5.3.3).

Each tag *tag* associated to a node $e$ is associated to its tag count $|tag|$. Using the tag counts, we can then detect if $e$ is abstract: a node $e$ is abstract iff all tag counts are equal to 1. In this case, it means that no node in the offspring will have to be merged. For example, in Figure 5.5, all nodes are abstract except $e_0$ since two tags in $e_0.tags$ have a tag count of 2. In practice, it means that some children of $e_0$ (in this case $e_1$ and $e_2$) will have to be merged.

**assocGroup** The function *assocGroup* (for associative grouping) groups all the nodes that need to be merged. To know if two nodes need to be merged, we look at their associated *tags* map. The merging condition is simple: two nodes should be merged iff they share the same tag. Below is an example of input/output pair of the *assocGroup* function:

```
Input:
    A has t1, t2
    B has t2, t3
    C has t3
    D has t4
    E has t4

Output:
```

```
[A, B, C] -> [t1, t2, t3]
[D, E] -> [t4]
```

In our case, the letters $A$ to $E$ are nodes and $t1$ to $t4$ are tags.

**mergeGroup** The function *mergeGroup* is the core of the recursive merging algorithm. As an input, it takes a list of abstract nodes—*i.e.*, abstract subtrees—that need to be merged and the set of common tags between the groups of nodes. The output is the abstract node $\hat{e}$. The inputs come from the output of the function *assocGroup*, described above, and the abstract node output replaces all children that were merged in the tree, as shown in the *abstractTree* algorithm 4

The function *mergeGroup* reduces the group list taken as input by repeatedly applying the function *mergeAbstractTrees*. Algorithm 5 describes how the function merges two abstract trees into one.

**Algorithm 5** Intra-page abstraction: merge two abstract trees

1: **function** MERGEABSTRACTTREES($\hat{e}_1, \hat{e}_2$)
2: // Group pairs of children containing the same tags
3: pairs, orphans ← groupPairs($\hat{e}_1$, $\hat{e}_2$)
4: mergedChildren ← pairs.map(mergeAbstractTrees)
5: **for** $e$ in orphans **do**
6: $e$.rel ← relType.Optional
7: **end for**
8: $\hat{e}$ ← new Node()
9: $\hat{e}$.tag = $\hat{e}$.tag
10: $\hat{e}$.attrs = mergeAttrs($\hat{e}_1$.attrs, $\hat{e}_2$.attrs)
11: $\hat{e}$.rel = mergeRel($\hat{e}_1$.rel, $\hat{e}_1$.rel)
12: $\hat{e}$.children = mergedChildren + orphans
13: $\hat{e}$.tags = $\{\hat{e}_1 \cup \hat{e}_2$ with tag counts set to $1\}$
14: **return** $\hat{e}$
15: **end function**

Before diving into the details of the algorithm, it is important to highlight that the inputs of the function *mergeAbstractTrees* are already abstract. At this stage of the algorithm, we are assured that along the whole branch starting at the root of both nodes sent as parameters, there are never two children belonging to the same cluster—*i.e.*, that needs to be merged. It means that the algorithm's sole purpose is to recursively merge the two trees between them (and not within).

The function starts by grouping the children of the two nodes into pairs and orphans. The function *groupPairs* returns two lists: the pairs of nodes that must be

merged and the orphan nodes. The orphan nodes are the children in $\hat{e_{1/2}}$ that have no corresponding element to be merged within $\hat{e_{2/1}}$, these elements are set as optional using the *rel* (as in relationship) property.

Before merging, we recursively call the function *mergeAbstractTrees* on each pair of nodes returned by *groupPairs*. Intuitively, the algorithm will stack the calls to *mergeAbstractTrees* until it reaches the leaves of the trees, then it will merge the groups of leaves and merge their ancestors as the function calls unstack.

Most of the steps described above help compute the *children* property of the abstract node returned by the *mergeAbstractTrees* function. Other properties of the nodes are also merged:

***rel*:** In case the relationships of the nodes to merge are different, they are merged using the following pattern matching:

```
mergeRelTypes :: (RelType,RelType)->RelType
mergeRelTypes (t1, t2)
    | (t1, t2) when t1 = t2 -> t1
    | (t1, Normal) -> t1
    | (_, ZeroToMany) -> ZeroToMany
    | (Optional, OneToMany) -> ZeroToMany
    | (t1, t2) -> mergeRelTypes (t2, t1)
```

***attrs*:** In case the attributes of the nodes to merge are different, they are merged. To merge the attributes, we select the attributes that are present in both nodes and merge their values using the *Longest Common Subsequence* (LCS) algorithm. For example, given three nodes having the following class values: *class* attribute: `"link nav-link"`, `"link active nav-link"` and `"link nav-link"`, the function *mergeAttrs* will return the following value: `"link nav-link"`.

**Stack Diagram** Since the algorithm has several levels of recursion, it may be hard to understand how a given tree will be abstracted.

In Figure 5.8 we describe the different steps of the algorithm. At each step, we show the output of the current function that is called. Below each tree, we show the current stack of functions called.

The functions we mention are all described earlier:

Figure 5.8 describes an example application of the *abstractTree* function. At each step, the current stack of functions is shown at the bottom. All functions shown have been described in the previous sections. Figure 5.7 summarizes all these functions

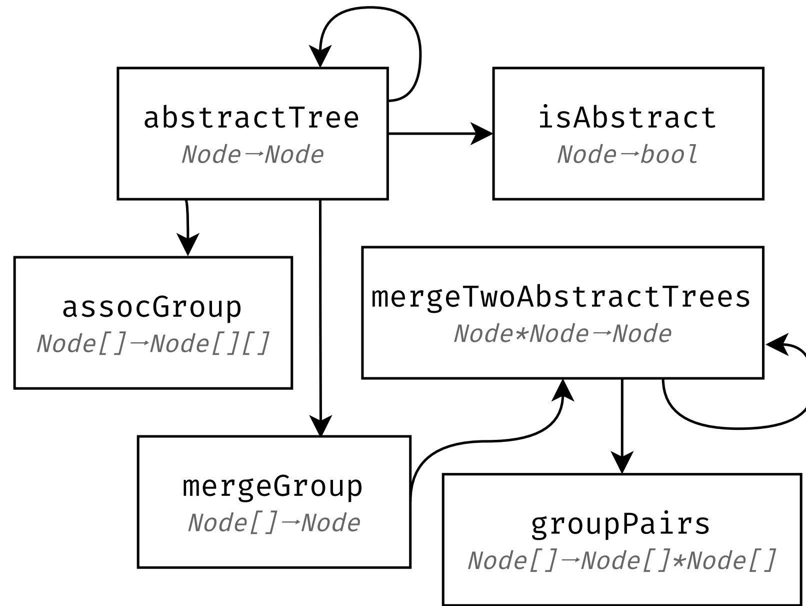



Figure 5.7: Key functions involved in the *abstractTree* algorithm. A directed arrow from function $f$ to $g$ indicates that $f$ calls $g$.

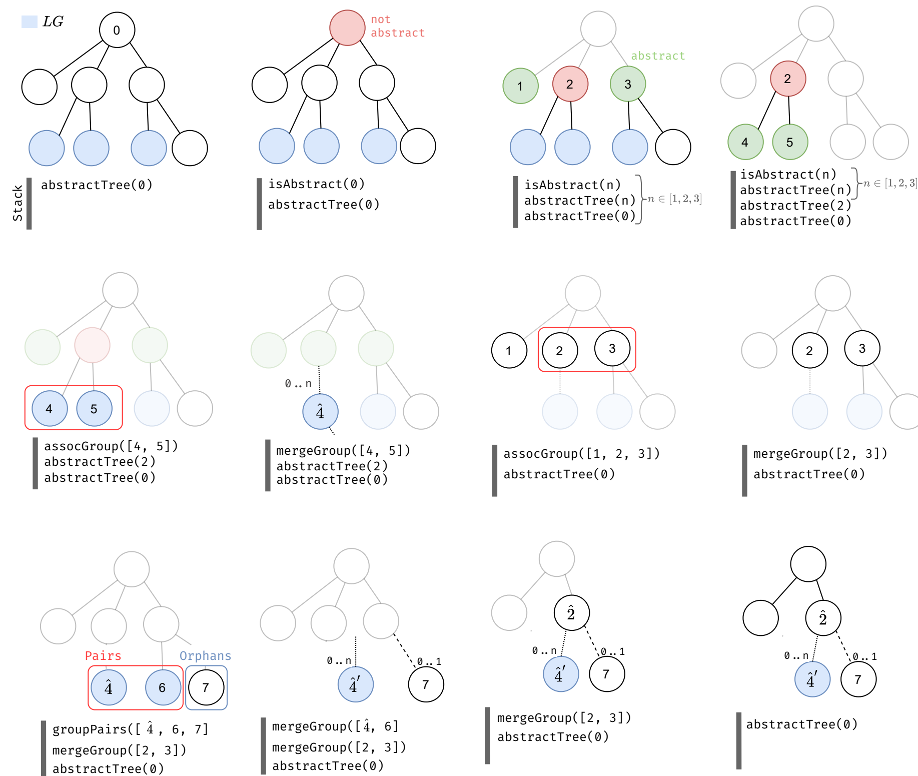



Figure 5.8: An example application of the *abstractTree* function. For each figure, the stack of the current step is shown below.

and how they interact. For simplicity, however, the *mergeTwoAbstractTrees* is not explicitly mentioned, it is considered as part of the function *mergeGroup*.

### 5.3.4 Inter-Page Abstraction

We described how to detect and abstract the repeating patterns contained within one page. In the second part of the APPSTRACT approach, we detect and abstract repeating patterns across pages of the web application.

The inter-page abstraction relies on tree-matching. A tree matching solution allows matching two web page DOM trees $p$ and $p'$. The matching $M_{p,p'}$ obtained is a subset of $p \times p'$ such that each tuple $(e, e') \in M_{p,p'}$ represents the fact that the element $e \in p$ matches with the element $e' \in p'$ (e.g. $e$ and $e'$ contain the names of two different products on two different product pages $p$ and $p'$).

As described in Section 5.3.3, inter-page abstraction is used at both the learning and prediction phases. In both cases, the inter-page abstraction can be described as a function that:

1. takes two templates as input: the reference template and a new template,
2. returns a template in which every template element $\hat{e}$ of the new template references its corresponding template element $\hat{e'}$ in the reference template (if a match was found).

**Algorithm 6** Inter-page abstraction

1: **function** INTER($\langle p_{ref}, T_{ref} \rangle, \langle p, T \rangle$)
2: $\quad M \leftarrow$ tree-matching($p$, $p_{ref}$)
3: $\quad T_{new} \leftarrow$ T.map(($e$, $id$) $\rightarrow T_{ref}[M[e]]$ if $e$ in $M$ else $id$)
4: $\quad$ **return** $\langle p, T_{new} \rangle$
5: **end function**

Algorithm 6 describes the inter-page abstraction process. The function role is to create a new identifier map $T_{new}$ in which each element from $p$ maps to the id of the matching element in $p_{ref}$ (if there is any) or remains the same.

Figure 5.2 describes how the function *inter* is used for both learning and prediction:

1. during learning, we apply inter-page abstraction between the mother template and each of the other templates. This step allows for building a model of the application where elements that appear in all page templates will have the same id,

2. during prediction, inter-page abstraction is used to match all elements from an unseen page to the template elements of its matching template page.

## 5.4 Limits

Modeling a whole application is a highly ambitious task. We hope our work can help progress toward this goal, but we cannot claim that it already does. Indeed, our current work has several limits, mainly:

*Template Topology* Real-life templates may have a much more complex structure than the one we assume:

1. there can be a tree-like structure with deeply nested templates,

2. there could be graph-like template structure (*e.g.*, components).

Our current *appstraction* method does not allow us to infer such template topologies.

*Mother Template Selection* During the learning stage, we choose the mother template arbitrarily among the existing template pages. This selection process assumes that all template pages have an equivalent amount of common parts.

## 5.5 Evaluation

In this section, we describe the experiment we devised allowing us to evaluate Appstract. The idea of the experiment is to consider several samples of DOM elements couples and manually judge if they are correctly labeled (should they have the same id?)

### 5.5.1 Experiment

We intend to measure over- and sub-abstraction rates of Appstract. Fundamentally, Appstract allows the creation of semantically rich, application-wide ids for elements on a webpage. Where:

- *Semantically Rich* means that two instances of the same template should have the same id
- *Application-wide* means that on a given application instances of a template will have the same id regardless of the page

There are two ways in which Appstract can fail:

- *Over-abstraction*: Items have the same id even though they are not of the same type, or
- *Sub-abstraction*: Items have different ids even though a human would assign them the same.

To analyze the performance of Appstract, we thus propose to measure the rates of over- and sub-abstraction. To do so, we developed a visual way to explore the web page abstractions created by Appstract. Given an application $A$:

- We apply Appstract to $A$ thus obtaining an abstraction $\langle \hat{A}, T_{\hat{A}} \rangle$
- Open two pages $p_{a_1}, p_{a_2}$ that we assume to be instances from the same template $p_a$ (e.g. two product pages or two blog posts)
- Use $\langle \hat{A}, T_{\hat{A}} \rangle$ to get the id $T_{\hat{p}}(e) = \hat{e}$ of every element $e$ on both pages $p_{a_1}, p_{a_2}$
- Visually display the id $\hat{e}$ when the mouse hovers over an element $e$

This method allows us to simply analyze the results of Appstract.

We separate our experiment into two parts: 1. Over-abstraction and 2. Sub-abstraction

### Defining Measures

After having described what we refer to as over-abstraction and sub-abstraction, in this section, we give formal definitions of these measures. To do so, we formulate the experimental application of Appstract as a binary classification problem in which each couple of elements between two pages can either have the same locator or a different one. We then define over-abstraction and sub-abstraction as complementary to the precision and recall of our binary classification problem.

Let us consider two pages $p_1$ and $p_2$ from the same application $A$. We apply Appstract to $A$ and obtain the abstraction $\langle \hat{A}, T_{\hat{A}} \rangle$. Then, we apply the abstract model to $p_1$ and $p_2$:

$$T_{\hat{A}}(\hat{p}_1) = \langle \hat{p_1}, T_{\hat{p_1}} \rangle \tag{5.2}$$

$$T_{\hat{A}}(\hat{p}_2) = \langle \hat{p_2}, T_{\hat{p_2}} \rangle \tag{5.3}$$

We formulate the experimental application of Appstract as a binary classification problem.

**Definition 5.5.1** (*Ground Truth Binary Classification function $f$*).

$$f : p_1 \times p_2 \rightarrow [0, 1] \tag{5.4}$$

$$f(e_1, e_2) = 1 \textit{ if } e_1 \sim e_2 \tag{5.5}$$

$$f(e_1, e_2) = 0 \textit{ otherwise} \tag{5.6}$$

*where $e_1 \sim e_2$ means that $e_1$ and $e_2$ are instances of the same template element (e.g. a buy button on a product page) as judged by a human.*

$f$ is the ground truth of our experiment. In practice, in the experiment, we only know a small manually labeled sample of $f$.

**Definition 5.5.2** (*Prediction function $\hat{f}$*).

$$f : p_1 \times p_2 \rightarrow [0, 1] \tag{5.7}$$

$$f(e_1, e_2) = 1 \textit{ if } T_{\hat{p_1}}(e_1) = T_{\hat{p_2}}(e_2) \tag{5.8}$$

$$f(e_1, e_2) = 0 \textit{ otherwise} \tag{5.9}$$

$\hat{f}$ is a guess made using APPSTRACT predicting whether two elements should have the same locator.

**Definition 5.5.3** (*Prediction function $\hat{f}$*).

$$f : p_1 \times p_2 \rightarrow [0, 1] \tag{5.10}$$

$$f(e_1, e_2) = 1 \textit{ if } T_{\hat{p_1}}(e_1) = T_{\hat{p_2}}(e_2) \tag{5.11}$$

$$f(e_1, e_2) = 0 \textit{ otherwise} \tag{5.12}$$

Given the above definitions, we can define the traditional precision and recall of our binary classification problem:

**Definition 5.5.4** (*Binary Classification Measures*).

$$tp = |\{e_1, e_2 \in p_1, p_2 / f(e_1, e_2) = \hat{f}(e_1, e_2) = 1\}| \tag{5.13}$$

$$fp = |\{e_1, e_2 \in p_1, p_2 / f(e_1, e_2) = 0, \hat{f}(e_1, e_2) = 1\}| \tag{5.14}$$

$$fn = |\{e_1, e_2 \in p_1, p_2 / f(e_1, e_2) = 1, \hat{f}(e_1, e_2) = 0\}| \tag{5.15}$$

*where $tp, fp,$ and $fn$ stand for true positive, false positive, and false negative, and $|S|$*

*is the cardinality of a set $S$.*

$$precision_{p_1,p_2}(\hat{f}) = \frac{tp}{tp + fp} \tag{5.16}$$

$$recall_{p_1,p_2}(\hat{f}) = \frac{tp}{tp + fn} \tag{5.17}$$

Finally, we define over-abstraction and sub-abstraction as complementary measures of precision and recall:

**Definition 5.5.5** (*Over-abstraction rate)*.

$$o_{p_1,p_2}(\hat{f}) = 1 - precision \tag{5.18}$$

**Definition 5.5.6** (*Sub-abstraction rate)*.

$$s_{p_1,p_2}(\hat{f}) = 1 - recall \tag{5.19}$$

In the next section, we describe an empirical experiment allowing us to estimate the over-abstraction and sub-abstraction rates of predictions made by APPSTRACT.

### Experimental Protocol

We devise an experimental protocol allowing us to evaluate the performance of APPSTRACT on an application. We separate the experiment into two parts: sub-abstraction and over-abstraction evaluations. In both experiments, we manually label couples of DOM elements from two pages of the same application. The idea is to estimate the precision and recall of the function $\hat{f}$ defined in section 5.5.1.

**Selecting couples of pages** Given an application containing several pages, we need to select a subset of page couples on which we experiment. There are two possibilities:

1. Picking random couples
2. Picking couples of pages that seem to belong to the same template

We choose the second possibility because it provides more relevant data to evaluate.

In the rest of the section, we thus describe the evaluation process on a single couple of DOMs $(p, p')$ belonging to the same template of the same application to which we already applied APPSTRACT. Formally, it means we use the functions $T_{\hat{p}}$ and $T_{\hat{p'}}$.

**Over-abstraction** Over-abstraction measures the precision of our algorithm: to how many couples of elements do we mistakenly assign the same id? More formally:

$$o_{p,p'}(\hat{f}) = 1 - \frac{tp}{tp + fp} \tag{5.20}$$

where $tp$ is the number of true positives and $fp$ is the number of false positives.

The objective of this part of the experiment is thus to estimate $tp$ and $fp$ given a couple of DOMs $(p, p')$. Both $tp$ and $fp$ concern the subset of positive couples: the subset of DOM elements couple $(e, e') \in p \times p'$ such that $T_{\hat{p}} = T_{\hat{p'}}$.

To label true and false positives, we develop a script that, given two abstracted web pages $p, p'$:

1. Display $p$ and $p'$ side by side,
2. Highlights one element of each page that has the same id,
3. Offer a popup allowing the tester to manually choose between "Correct", "Wrong" or "Skip"

This experiment allows to estimate the amount of true positives $tp$ ("Correct") and false positives $fp$ ("Wrong"). The "Skip" option is most useful in cases where the elements highlighted are not visible. Results allow us to calculate an estimate of the over-abstraction $\hat{o_{p,p'}}(\hat{f})$

**Sub-abstraction** Sub-abstraction measures the recall of our algorithm: how many elements should have the same id but not? More formally:

$$s_{p,p'}(\hat{f}) = 1 - \frac{tp}{tp + fn} \tag{5.21}$$

To estimate sub-abstraction, we develop a script that, given two abstracted web pages $p, p'$:

1. Display $p$ and $p'$ side by side
2. Ask the user to select one element from each web page that should have the same id
3. Check if they indeed have the same id. If yes, increment the number of true positive $tp$ else increment the number of false negatives $fn$

We do not disclose the results of the comparison to avoid influencing the following experimenter's choices.

#### Preliminary Results and Limitations

While we, unfortunately, did not have enough time to lead the described experiments quantitatively, our first tests of Appstract are promising but show some limitations, mainly related to Free Text detection. We call *Free Text*, portions of web pages containing text written and, more importantly, formatted by the user. The most significant example of free text is content written on forums or comment sections. Comment editors on these websites often offer users the option to format (bold, italic, paragraphs...) their text. The formatting tags in free text introduce noise that is difficult to ignore for Appstract. Indeed, the main assumption used by Appstract to separate content from presentation (templates) is that web applications present a high variability of content (text) encoded in a limited amount of templates. In the case of free text, this assumption is false: the structure of the webpage defined through tags is partially written by the user which leads to the high variability of structure since, in this case, the structure is also part of the content. Appstract then mistakenly believes the formatting tags should be used to infer the template which can lead to unexpected results. To solve the free-text limitation, we need to find a way to detect free text.

## 5.6 Conclusion

We believe Appstract's approach is an important and pioneering solution in two ways.

Firstly, we introduce the webpage abstraction problem independently to its traditional areas of application (e.g. web testing, data extraction, web analytics). We propose a formalization of the abstraction problem using the idea of robust and semantically-rich application-wide identifiers.

Secondly, we propose an abstraction approach combining intra- and inter-page abstraction to generate such identifiers. While both intra- and inter-page abstraction underlying techniques are akin to existing approaches in different research areas, to our knowledge, the idea to combine both to infer a web application abstraction is original.

The web application abstraction problem we described is very complex, mainly because of the high variety of existing modern web applications. Unfortunately, Appstract is not the off-the-shelf web application abstraction solution working on any application that we intended to develop. Appstract mainly suffers from a lack

of quantitative evaluation benchmark and can be misled by Free Text sections (e.g. comment sections or forum posts). However, we hope it will serve as a starting point to developing new solutions for the web application abstraction problem.

# Chapter 6

# Conclusion

The original purpose that inspired my work was to build a comprehensive set of tools allowing for analyzing interactions between the user and any real-life web applications to detect every time the user struggles to achieve her purpose. I initially expected most challenges to arise from the analysis of these interactions. However, very quickly, I encountered one major obstacle: how to even describe an *interaction*? In the end, all my studies and contributions have been guided by this question, leaving the analysis part unfortunately untouched.

Describing an interaction between a user and a web application requires a surprisingly deep and subtle understanding of the application. An interaction between a user and an application is the expression of the intent of the user. She translates this intent into a series of actions. These actions can be seen as a language, a language entirely defined by the makers of the application. This language has to be easily discoverable by the user by just looking at the application, it is also as similar as possible to the one used by other applications (UX/UI conventions). In this context, describing action is akin to figuring out what is the limited set of *tokens* the language is made of. Without such preliminary abstraction work, each node of every page is seen as unique. Analyzing streams of such actions would be like trying to understand a language where almost no two words are the same.

The main challenge in describing an interaction is the description of a location on the web page. Such description is called a *locator*. The first kind of variability we attempted to handle is the variability between pages: *how to build a locator of an element that remains the same for each page on which the element appears?* To answer this question we attempted to compare every couple of pages from a given web page and mark every matching element so they would have the same locator. However, every off-the-shelf state-of-the-art tree-matching solution we tried to use had either

unpractical computation times (seconds to minutes) or unacceptable accuracy. That is why we developed SFTM which stands for Similarity-base Flexible Tree Matching (see chapter 2).

SFTM is a practical solution to tree-matching specialized web pages. Tree-matching algorithms have been studied for more than 50 years. To improve on such a legacy, we took a radically different approach from traditional generic tree-matching solutions. While state-of-the-art solutions are always topology first, we instead used statistical tools to match nodes using their labels first before taking into account topology. This approach relies on the specificity of web pages: their labels contain a large amount of information (tags and attributes).

The second major obstacle we had to face was the lack of available open-source benchmarks for web page matching. This is mainly because most existing generic tree-matching solutions are correct by construction and their efficiency is thus proved analytically rather than empirically. Since our approach included more statistical tools such an approach could not be applied to SFTM. We thus developed a synthetic large-scale benchmark based on mutations. Results showed significant gains in terms of both computation times and accuracy thus allowing us to move towards our initial goal: applying tree matching to build meaningful and robust locators.

Following our work on tree matching, we studied the state of the art on robust locators and found out no existing solution was based on tree matching. We thus worked on an approach to improve the robustness of locators using SFTM that we named Erratum (see chapter 3). While our approach could be used to generate robust locators (and we did so later on when we integrated Erratum) we described Erratum as a locator repair solution. The idea of locator repair is to use the last web page on which a given locator is working and try to relocate the located element on the newer version of the page where the locator breaks. Existing locator repair solutions try to find individual elements of one version of the page in the other. Our approach uses all nodes from both pages to serve as anchors and thus tremendously help relocate the element.

Once again, we could not find any off-the-shelf benchmarking solutions to compare Erratum to other locator repair solutions. We thus built an open benchmark using both synthetic and manually labeled test data. Results showed that Erratum was significantly more accurate for almost no overhead when compared to existing solutions.

Following our work on Erratum, a large online French retail company reached out to us. They shared how painful the locator breakage problem is to their test

department, entailing large waste of resources and sometimes even the cancellation of whole test campaigns. Since they relied on a famous open-source library for their tests, we thus integrated Erratum into the open source framework (see chapter 4). Our integration allows the user to use automatically generated Erratum-based locators as a replacement for traditional manually written XPath or CSS-based locators. This allowed testers to both save time on manually writing XPaths and benefit from much more robust locators.

Finally, we set out to apply our previous work to achieve our original objective: to build a tool that can infer an abstract model of any web application without requiring any human intervention. We called our tool Appstract (see chapter 5). Intuitively, Appstract works in the following way: given a set of web pages from the same application, we group the pages belonging to the same templates together (e.g. blog page, product page, product list page). We then use an intra-page abstraction algorithm to extract patterns within one page. The output of the intra-page abstraction step is a set of abstract templates. We then select one mother template and use tree matching (i.e. inter-page abstraction) to match every template to the mother template. The matchings are then used to update all templates so that elements appearing on all pages (e.g. menu, logo) have the same locators. The result of the above steps is an abstract model of the application. Each new page can then be matched against this model to retrieve semantically rich and robust locators for each element of the page.

## Perspectives

**Short Term** In the short term, some work remains to be done so that Appstract can be applied confidently on any web page. In particular, three challenges remain to be addressed:

At the moment Appstract is not entirely unsupervised since the clustering phase has to be done manually. To solve this issue, we need to experiment on unsupervised clustering of web pages into templates (e.g. blog page, product page, product list page).

Secondly, Appstract currently does not support what we called *free text*. Free text is a part of a page where users can write formatted content. This is a problem because Appstract relies on the assumption that while the data of an application varies greatly, the structure does not. This assumption is wrong in the case of free text because it is user-generated content (large variability) that contains a structure (formatting)

Finally, we have not had the time to evaluate Appstract quantitatively which is necessary if we want to solve its limitations described above and allow the community to trust and build upon Appstract.

**Middle & Long Term** In the middle to long term, the most exciting perspectives are the application of Appstract to web testing, web analytics, and web data extraction. We strongly believe our approach can bring a whole range of novel approaches to each of these fields. As an example, Appstract can be used to build a navigational model allowing us to generate web test scenarios, it can be used to direct crawling and extract data from an application, and finally, our original intent, Appstract can be used to mine user behavior where user actions are encoded using the inferred abstract identifiers.

# Bibliography

[1] Eugene Agichtein, Eric Brill, and Susan Dumais. Improving web search ranking by incorporating user behavior information. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 19–26, 2006.

[2] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning user interaction models for predicting web search result preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–10, 2006.

[3] Emil Alegroth, Michel Nass, and Helena H Olsson. Jautomate: A tool for system- and acceptance-test automation. In *ICST*, pages 439–446. IEEE, 2013.

[4] Arvind Arasu and Hector Garcia-molina. Extracting structured data from web pages. pages 337–348.

[5] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Synthesizing web element locators (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 331–341. IEEE, 2015.

[6] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. Synthesizing web element locators. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 331–341, 2016.

[7] Greg Barish, Yi Shin Chen, Dan DiPasquo, Craig A. Knoblock, Steven Minton, Ion Muslea, and Cyrus Shahabi. TheaterLoc: Using information integration technology to rapidly build virtual applications. *Proceedings - International Conference on Data Engineering*, pages 681–682, 2000.

[8] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on*

*European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 142–153, 2019.

[9] Karl Bringmann, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. 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*, pages 1190–1206. Society for Industrial and Applied Mathematics, 2018.

[10] Horst Bunke and Kim Shearer. A graph distance metric based on the maximal common subgraph. Technical report, 1998.

[11] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *UIST*, pages 473–484, 2013.

[12] Chia-Hui Chang, Mohammed Kayed, M R Girgis, Khaled F Shaalan, Ramzy Girgis, and Khaled F Shaalan. A Survey of Web Information Extraction Systems. Technical Report 10, 4 2006.

[13] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. Gui testing using computer vision. In *CHI*, pages 1535–1544, 2010.

[14] K P Chaudhari, Miss Poonam, and Rangnath Dholi. Template Extraction From Heterogeneous Web Pages. Technical report.

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

[16] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting changes in XML documents. *Proceedings - International Conference on Data Engineering*, pages 41–52, 2002.

[17] Antoine Craske. Cerberus: an automated tool for continuous testing. In *ICST*, 2020. tool demonstration.

[18] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards automatic data extraction from large web sites. *VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases*, pages 109–118, 2001.

[19] Davi Castro De Reis, Paulo B. Golgher, Altigran S. Da Silva, and Alberto H.F. Laender. Automatic web news extraction using tree edit distance. In *Thirteenth International World Wide Web Conference Proceedings, WWW2004*, pages 502–511, New York, New York, USA, 2004. ACM Press.

[20] Yeem Dinitz, Alon Itai, and Michael Rodeh. On an Algorithm of Zemlyachenko for Subtree Isomorphism. Technical report, 1998.

[21] Rodrigo Cordeiro Dos Santos and Carmem Hara. A semantical change detection algorithm for xml. *SEKE 2007*, page 438, 2007.

[22] Marios Fokaefs, Rimon Mikhaiel, Nikolaos Tsantalis, Eleni Stroulia, and Alex Lau. An empirical study on web service evolution. In *2011 IEEE International Conference on Web Services*, pages 49–56. IEEE, 2011.

[23] Georg Gottlob and Christoph Koch. Logic-based Web information extraction. *SIGMOD Record*, 33(2):87–94, 2004.

[24] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for Web information extraction. *Journal of the ACM*, 51(1):74–113, 2004.

[25] Şule Gündüz and M Tamer Özsu. A web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–540, 2003.

[26] Qi Guo and Eugene Agichtein. Exploring mouse movements for inferring query intent. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 707–708, 2008.

[27] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. 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*, pages 751–762. ACM, 2016.

[28] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do record/replay tests of web applications break? In *ICST*, pages 180–190. IEEE, 2016.

[29] Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do Record/Replay Tests of Web Applications Break? In *2016 IEEE International Conference*

*on Software Testing, Verification and Validation (ICST)*, pages 180–190. IEEE, 6 2016.

[30] Yanan Hao and Yanchun Zhang. Web services discovery based on schema matching. In *Proceedings of the thirtieth Australasian conference on Computer science-Volume 62*, pages 107–113. Australian Computer society, Inc., 2007.

[31] Chun Nan Hsu and Ming Tzung Dung. Generating finite-state transducers for semi-structured data extraction from the Web. *Information Systems*, 23(8):521–538, 1998.

[32] Yuqing Huang, Guangzhi Qi, and Fuyan Zhang. Extracting semi-structured information from the WEB. *Ruan Jian Xue Bao/Journal of Software*, 11(1):73–78, 2000.

[33] Javaria Imtiaz, Muhammad Zohaib Iqbal, et al. An automated model-based approach to repair test suites of evolving web applications. *Journal of Systems and Software*, 171:110841, 2021.

[34] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology*, 113:1–19, 2019.

[35] Lucija Ivančić, Dalia Suša Vugec, and Vesna Bosilj Vukšić. Robotic process automation: systematic literature review. In *International Conference on Business Process Management*, pages 280–295. Springer, 2019.

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

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

[38] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[39] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[40] Hiroyuki Kirinuki, Haruto Tanno, and Katsuyuki Natsukawa. Color: correct locator recommender for broken test scripts using various clues in web application. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 310–320. IEEE, 2019.

[41] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[42] Ranjitha Kumar, Jerry O Talton, Salman Ahmad, and Scott R Klemmer. Bricolage: example-based retargeting for web design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2197–2206. ACM, 2011.

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

[44] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing, Verification and Reliability*, 31(3):e1767, 2021.

[45] Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Sidereal: Statistical adaptive generation of robust locators for web testing. *Software Testing Verification and Reliability*, 31(3):1–31, 2021.

[46] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Reducing web test cases aging by means of robust xpath locators. In *SBST*, pages 449–454. IEEE, 2014.

[47] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Meta-heuristic generation of robust xpath locators for web testing. In *SBST*, pages 36–39. IEEE, 2015.

[48] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Using multi-locators to increase the robustness of web test cases. In *ICST*, pages 1–10. IEEE, 2015.

[49] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.

[50] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. <span style="font-variant:small-caps;">Robula+</span> : an algorithm for generating robust XPath locators for web testing: ROBULA+: An Algorithm for Generating Robust XPath Locators. *Journal of Software: Evolution and Process*, 28(3):177–204, 6 2016.

[51] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Pesto: Automated migration of dom-based web tests towards the visual approach. *STVR*, 28(4):e1665, 2018.

[52] Julien Leveau, Xavier Blanc, Laurent Réveillère, Jean-Rémy Falleri, and Romain Rouvoy. Fostering the diversity of exploratory testing in web applications. *Software Testing, Verification and Reliability*, 32(5):e1827, 2022.

[53] Alon Y Levy, Anand Rajaraman, and Joann J Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. *Proceedings of 22th International Conference on Very Large Data Bases*, 1:1–26, 1996.

[54] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33. ACM, 2006.

[55] Bing Liu, Robert Grossman, and Yanhong Zhai. Mining data records in web pages. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 601–606, 2003.

[56] Zhenyue Long, Guoquan Wu, Xiaojiang Chen, Wei Chen, and Jun Wei. Webrr: self-replay enhanced robust record/replay for web application testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1498–1508, 2020.

[57] Ali Mesbah and Arie Van Deursen. Invariant-based automatic testing of ajax user interfaces. In *2009 IEEE 31st International Conference on Software Engineering*, pages 210–220. IEEE, 2009.

[58] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[59] Gengxin Miao, Junichi Tatemura, Wang-Pin Hsiung, Arsany Sawires, and Louise E Moser. *Extracting Data Records from the Web Using Tag Path Clustering.*

[60] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[61] Bamshad Mobasher, Honghua Dai, Tao Luo, and Miki Nakagawa. Effective personalization based on association rule discovery from web usage data. In *Proceedings of the 3rd international workshop on Web information and data management*, pages 9–15, 2001.

[62] Ion Muslea. Extraction Patterns for Information Extraction Tasks: A Survey. Technical report, 1999.

[63] Svetlozar Nestorov and SERGE ABITEBOUL. Extracting Schema from Semistructured Data. Technical report.

[64] Vu Nguyen, Thanh To, and Gia-Han Diep. Generating and selecting resilient and maintainable locators for web automated testing. *Software Testing, Verification and Reliability*, 31(3):e1760, 2021.

[65] Alessandreia Oliveira, Gabriel Tessarolli, Gleiph Ghiotto, Bruno Pinto, Fernando Campello, Matheus Marques, Carlos Oliveira, Igor Rodrigues, Marcos Kalinowski, Uéverton Souza, et al. An efficient similarity-based approach for comparing xml documents. *Information Systems*, 78:40–57, 2018.

[66] Changkun Ou, Daniel Buschek, Malin Eiband, and Andreas Butz. Modeling Web Browsing Behavior across Tabs and Websites with Tracking and Prediction on the Client Side. 2021.

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

[68] Mateusz Pawlik and Nikolaus Augsten. Efficient computation of the tree edit distance. *ACM Transactions on Database Systems (TODS)*, 40(1):1–40, 2015.

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

[70] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[71] Davi de Castro Reis, Paulo Braz Golgher, Altigran Soares Silva, and AlbertoF Laender. Automatic web news extraction using tree edit distance. In *Proceedings of the 13th international conference on World Wide Web*, pages 502–511. ACM, 2004.

[72] Sunita Sarawagi. *Information Extraction*, volume 39. 1996.

[73] Ramesh R Sarukkai. Link prediction and path analysis using markov chains. *Computer Networks*, 33(1-6):377–386, 2000.

[74] Stanley M Selkow. The tree-to-tree editing problem. *Information processing letters*, 6(6):184–186, 1977.

[75] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *FSE*, pages 488–498, 2013.

[76] Andrea Stocco, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. Apogen: automatic page object generator for web testing. *Software Quality Journal*, 25(3):1007–1039, 2017.

[77] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 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*, pages 503–514. ACM, 2018.

[78] Sathya Sundaram and Sanjay K Madria. A change detection system for unordered xml data using a relational model. *Data & Knowledge Engineering*, 72:257–284, 2012.

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

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

[81] Gabriel Valiente. An efficient bottom-up distance between trees. In *spire*, pages 212–219, 2001.

[82] Waraporn Viyanon and Sanjay K Madria. Xml-sim-change: structure and content semantic similarity detection among xml document versions. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 1061–1078. Springer, 2010.

[83] Jason T.L. Wang and Kaizhong Zhang. Finding similar consensus between trees: An algorithm and a distance hierarchy. *Pattern Recognition*, 34(1):127–137, 1 2001.

[84] Jiying Wang and F.H. Lochovsky. Wrapper induction based on nested pattern discovery. *World Wide Web Internet And Web Information Systems*, pages 1–29, 2002.

[85] Yuan Wang, David J DeWitt, and J-Y Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings 19th international conference on data engineering (Cat. No. 03CH37405)*, pages 519–530. IEEE, 2003.

[86] Takashi Yamauchi. Mouse trajectories and state anxiety: feature selection with random forest. In *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, pages 399–404. IEEE, 2013.

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

[88] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. Robust test automation using contextual clues. *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 304–314, 2014.

[89] Xuchen Yao, Benjamin Van Durme, Chris Callison-Burch, and Peter Clark. Answer extraction as sequence tagging with tree edit distance. In *Proceedings of the 2013 conference of the North American chapter of the association for computational linguistics: human language technologies*, pages 858–867, 2013.

[90] Xun Yuan and Atif M Memon. Using gui run-time state as feedback to generate test cases. In *29th International Conference on Software Engineering (ICSE'07)*, pages 396–405. IEEE, 2007.

[91] Yanhong Zhai and Bing Liu. Web data extraction based on partial tree alignment. In *Proceedings of the 14th international conference on World Wide Web*, pages 76–85. ACM, 2005.

[92] Yanhong Zhai, Bing Liu, Zha Yanhong, Liu Bing, and Yanhong Zhai. Web data extraction based on partial tree alignment. pages 76–85, 2005.

[93] Kaizhong Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern recognition*, 28(3):463–474, 1995.

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

[95] Yu Zheng, Song Huang, Zhan-wei Hui, and Ya-Ning Wu. A method of optimizing multi-locators based on machine learning. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 172–174. IEEE, 2018.