UI driven dynamic analysis and testing of web applications

by

Rahul Krishna Yandrapally

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

April 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

UI driven dynamic analysis and testing of web applications

submitted by **Rahul Krishna Yandrapally** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Electrical and Computer Engineering**.

Examining Committee:

Ali Mesbah, Professor, Electrical and Computer Engineering, THE UNIVERSITY OF BRITISH COLUMBIA
Supervisor

Julia Rubin, Associate Professor, Electrical and Computer Engineering, THE UNIVERSITY OF BRITISH COLUMBIA
Supervisory Committee Member

Sathish Gopalakrishnan, Associate Professor, Electrical and Computer Engineering, THE UNIVERSITY OF BRITISH COLUMBIA
Supervisory Committee Member

Konstantin Beznosov, Professor, Electrical and Computer Engineering, THE UNIVERSITY OF BRITISH COLUMBIA
University Examiner

Gene Moo Lee, Associate Professor, Information Systems and Analytics, THE UNIVERSITY OF BRITISH COLUMBIA
University Examiner

William G.J. Halfond, Professor, Computer Science, UNIVERSITY OF SOUTHERN CALIFORNIA
External Examiner

Abstract

Modern web applications have evolved to become highly complex, interact-able software capable of replicating the functionalities of traditional software. The web application development itself evolved to present a variety of options in terms of development frameworks and programming languages available to the developers. This however, also leads to one of core challenges in web application testing, the heterogenous nature of web applications, which makes software testing techniques that rely on analyzing source-code ineffective. Instead, end-to-end UI testing is the preferred mode of ensuring the web applications do not contain faults or bugs. However, the web UI testing ecosystem has failed to evolve at a similar pace to web development but instead relies on human effort in practice, making web application testing a costly endeavour.

The goal of this thesis is to enable automatic testing of web applications reducing reliance on human effort. First, we identified web page equivalence to be a core challenge faced by existing automatic UI test generation techniques, performed an empirical evaluation of ten existing state comparison techniques and identified the characteristics of modern web apps that render these techniques ineffective in generating optimal UI test suites. Thereafter, we designed a novel state comparison and test generation technique which treats a web page as a set of individual functionalities that can be represented in a hierarchy of fragments, helping us test modern web applications in an effective manner. Next, we designed the first universally applicable mutation analysis framework for web applications regardless of the back-end and front-end technologies they were built upon. It is capable of assessing the quality of a given UI test suite without even requiring the source-code of the web application. Finally, we tackle the challenge of enabling

API testing universally for any web application. Our API testing framework exercises the UI to carve API test suites and generates API specification that would enable API testing for any given web application.

We implement our techniques in open-source tools and evaluate them through a set of controlled experiments. The results show that our techniques succeeded in accomplishing the set research goals.

Lay Summary

Testing modern web applications to ensure they are bug free in practice is a costly manual process which consumes a bulk of the development time. As the web applications grow more complex to enable greater interact-ability and functionality, testing them becomes equally challenging. The work in this dissertation aims to facilitate automatic testing of modern web applications by proposing techniques that tackle a series of core challenges that have so far hindered effective web application testing. Through controlled experiments, we show how our techniques improve upon the existing state-of-the-art and also provide solutions that are first of their kind for web applications.

Preface

Each research chapter of this dissertation corresponds to a research paper, which has been published or is currently under review. I have collaborated with my PhD advisor, Prof. Ali Mesbah, for conducting the research projects in all chapters. I was the main contributor for all chapters, including the idea, design, development, and evaluation of the work. I had the collaboration of Dr. Andrea Stocco, a former Post Doctoral researcher in our lab, for Chapter 2, and worked in collaboration with Dr. Saurabh Sinha and Dr. Rachel Tzoref–Brill from IBM Research for Chapter 5. In Chapter 2, Dr. Andrea Stocco helped me design initial experiments and writing the research paper. Similarly, in Chapter 5, Dr. Saurabh Sinha and Dr. Rachel Tzoref-Brill have helped me design research goals and write the research paper. The publications for each chapter are as follows.

- Chapter 2
  - ”Near-Duplicate Detection in Web App Model Inference” [173]: **Rahul Krishna Yandrapally**, Andrea Stocco, and Ali Mesbah, *ACM/IEEE International Conference on Software Engineering* (ICSE 2020). 186–197. (*acceptance rate: 20.9%*)
- Chapter 3
  - ”Fragment-Based Test Generation For Web Apps” [172]: **Rahul Krishna Yandrapally**, Ali Mesbah, *IEEE Transactions on Software Engineering* (TSE 2022). 16 pages [Early access].
- Chapter 4

  - ”Mutation Analysis for Assessing End-to-End Web Tests” [166]: **Rahul Krishna Yandrapally**, Ali Mesbah, *IEEE International Conference on Software Maintenance and Evolution* (ICSME 2021). 183–194. (*acceptance rate: 24%*)

- Chapter 5

  - ”Carving UI Tests to Generate API Tests and API Specifications” [174]: **Rahul Krishna Yandrapally**, Saurabh Sinha, Rachel Tzoref-Brill, Ali Mesbah, *ACM/IEEE International Conference on Software Engineering* (ICSE 2023). 12 pages. (*acceptance rate: 26%*)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Ali Mesbah, for his exceptional guidance and supervision in the past five and a half years. He gave me the freedom to make mistakes, learn important lessons and find my own way, but always steering me towards becoming a competent independent researcher. This thesis would not have been possible without his impeccable insights and unwavering support. I am indebted to Andrea Stocco, a great mentor and a friend, whose support and encouragement helped me endure and progress beyond the difficult first year of my PhD. I would like to thank Saurabh Sinha and Rachell Tzoreff-Brill from IBM research, whose collaboration helped me formulate the final research problem of my PhD. Saurabh has been a mentor to me since my days in IBM, and it was his support and encouragement that made me take up PhD in the first place. I am thankful to my supervisory committee Dr. Julia Rubin and Dr. Sathish Gopalakrishnan for providing valuable comments and suggestions on my PhD proposal as well as my dissertation. My sincere gratitude goes to my final examination committee, Dr. Konstantin Beznosov, Dr. Gene Moo Lee, and Dr. William G.J.Halfond. I would also like to acknowledge all my dear friends and colleagues in Software Analysis and Testing (SALT) Lab at UBC. I also gratefully acknowledge the financial support by the Natural Sciences and Engineering Research Council of Canada (NSERC), as well as the Four Year Doctoral Fellowship of UBC, that helped me to focus full time on my research. Finally, I would like to thank my parents, sister, family and friends for their love and support over the years. I am specially indebted to my wife, Keerthi, for her love, understanding, encouragement and most importantly patience through my journey as a PhD student.

To my daughter Sitara.

Chapter 1

Introduction

Commercial software, made possible by the advent of personal computing, was distributed physically using drives such as CDs to be used on individual computers for much of the previous century. A rapid rise of cloud infrastructure and internet availability has rendered the concept of owning software itself obsolete where, today, users need only a *web browser* to satisfy all their computing needs.

This transformation of personal computing is made possible by the development of complex web applications, which respond to user interactions just like desktop software while relying on a server to handle computationally intensive tasks. Modern web apps are a heterogeneous collection of dynamically integrated web components that are spread across multiple tiers. While the server-side components written in languages such as Java and PHP contain the bulk of the business logic, client-side components are built using JavaScript, HTML, and CSS and, are rendered by browsers as interact-able web pages.

Owing to their heterogeneous nature, testing web apps programmatically is challenging and is often performed in an end-to-end (E2E) fashion by exercising the GUI functionality of web apps. Given the short release cycles of web apps, automated regression testing assumes a significant role in the validation of software changes. In practice, regression testing for web apps is carried out using automated UI test suites such as the one shown in Listing 1.1. A UI test case is a sequence of UI actions and assertions which verify GUI states resulting from actions.

Listing 1.1: Selenium JUnit Test Case

```
private WebDriver driver = new ChromeDriver();

@Before
public void setUp() {
        driver.get(app_url);
}

@Test
public void testCollabtiveLoginUser() throws Exception {
        driver.findElement(By.id("username")).sendKeys("username001");
        driver.findElement(By.id("pass")).sendKeys("password001");
        driver.findElement(By.cssSelector("button.loginbtn")).click();
        driver.findElement( By.xpath("//*[@id=\"mainmenue\"]/li[2]/a")).click();
        assertTrue( driver.findElement(By.cssSelector("body")).getText()
                        .matches("^[\\s\\S]*username001[\\s\\S]*$"));
```

However, regression testing using such automated UI test suites is a costly activity [152], requiring testers to manually create regression test suites, using a combination of programming and record/replay tools such as Selenium. In traditional test suites, test failures during regression testing typically indicate an application change or an application bug. However, automated UI test suites are prone to *test breakages* which cause test failures which reveal no application bugs. These breakages typically occur due to broken locators such as “id” attributes that are used to locate web elements to perform UI actions. As a result, maintenance of such regression test suites is known to be expensive as it requires human intervention [58, 93]. Even minor changes of the app can cause many test cases to fail; for example, according to a study at Accenture [58] even simple modifications to the user interface of apps result in 30–70% changes to tests, which costs $120 million per year to repair. When the test maintenance cost becomes overwhelming, whole test suites are abandoned [33]. To create regression suites that can handle web app evolution, researchers have attempted to either develop test suites that are robust to application changes [151, 167] or repair broken tests [31, 146]. Both these approaches are limited in their scope and cannot handle changes in app behavior beyond broken web element locators.

Therefore, automatic generation of regression test suites seems a necessity. However, the effectiveness of web test generation techniques [23, 97] is limited by the availability of an accurate and complete model of the app under test. Manual construction of such models for complex apps is not practical and hence, needs to be automated. Web app crawling [94, 156] is a popular technique for automated model inference, where, the state space of a given web app is explored iteratively to capture state transitions in response to user actions. While crawlers are efficient even for large complex web apps, they output sub-optimal models for even small apps that can contain near-duplicate states which are states that have similar functionally but can contain any number of structural, textual or visual differences.

In addition to the challenges to model inference, near-duplicate states also pose a challenge to creation of reliable test oracles both for automated as well as generated test oracles. In traditional software, the program output is deterministic and therefore regression test suites can easily verify output. However, in web applications, the same does not hold true and therefore, test oracles need to be resilient to minor cosmetic changes in web pages while still being able to detect functional changes.

1.1 Dynamic State Equivalence and Near-Duplicates in Web Apps

The state in the inferred models of web apps represents the client-side dynamic webpage of the app, as represented by the Document Object Model (DOM) [1] in the browser. However, an adequate model should contain only the minimal set of unique states that represent the web app functionalities, while discarding insignificant states that dot not contribute to exposing new functionality to the end user. Instances of such insignificant states are, for instance, pages only differing by small cosmetic changes, which are also referred to as *near-duplicates* in the literature.

From an end-to-end (E2E) testing perspective, clone and near-duplicate states in web app models negatively impact their accuracy and completeness, undermining the quality of the test suites generated from such models in terms of size, runtime, and coverage.

While web testing deals with the similarity between web pages of the same app, clone and near-duplicate detection *across* different web apps has been an active research topic in many fields [43, 44, 64, 86]. However, there is, neither a consensus on the definition of near-duplicate web pages, nor a unified standard against which a technique can be assessed [63, 64]. Depending on the purpose of comparison, researchers have defined various abstractions for a web page [48, 96, 98, 132] to programmatically detect near-duplicates.

Definition 1 (State Abstraction Function).

$$\mathscr{A}(dist, p_1, p_2, t) \begin{cases} true & : dist(p_1, p_2) < t \\ false & : otherwise \end{cases}$$

Existing crawlers employ a state abstraction function(SAF), defined in 14, which determines if $p_1 \simeq p_2$ for any two given web states p_1 and p_2, by comparing the distance between an abstraction over p_1 and p_2 to a constant t. t is called the distance threshold, and is often determined by a trial and error process as no systematic means have been proposed so far to estimate it automatically. However, the changes in the DOM or visual characteristics used as abstractions, do not always result in semantic changes such as the functionality, defeating the goal of state equivalence in functional web testing.

Research Question 1. *What are functional near-duplicates and how to detect them?*

In order to design effective state comparison techniques, we needed to understand the nature of changes that characterize semantic relationships between web states from a functional testing perspective. In consideration, near-duplicates become interesting as they differ from each other while being classified as similar. Our study on near-duplicate pages (Chapter 2) proposed a methodology to classify a given state-pair and assess the effectiveness of existing techniques. We purposed 10 different existing near-duplicate detection techniques from three domains for model inference and proposed a systematic approach to select optimal thresholds. Our study concluded that regardless of the abstraction used, neither structural nor visual characteristics of the whole page can reliably be used to capture their underlying functionality.

Research Question 2. *How to produce accurate web app models and effective regression test suites?*

As the existing model inference techniques are ineffective in correctly classifying state changes for dynamic web apps, they produce inaccurate models rendering dependent techniques such as test generation ineffective. We developed a novel model inference technique (Chapter 3), that is effective in handling web app dynamism to improve model precision and recall while generating *smart* test oracles in regression test suites.

1.2 Generating and Maintaining Effective Regression Test Suites

Regression test suites are often generated/created to cover all possible actions/states in web app. Another important but often neglected aspect of regression testing is the creation of effective test assertions or test oracles, which are key in detecting app regressions. Given an accurate model of the web app, model-based test generation can efficiently create test suites based on predefined model coverage criteria. However, generated test assertions are known to be brittle [157] and result in false test failures which require costly manual analysis.

Regression test assertions should be able to detect unexpected app behavior, but at the same time, be tolerant to minor changes that do not affect the functionality. However, existing techniques[97, 127] employ abstractions over DOM similar to definition 14 and hence become ineffective. In practice, therefore, test oracles are often manually designed even for generated test cases [157]. In this proposed thesis, I also propose means to generate effective test oracles which require a nuanced comparison of web pages beyond simple whole-page comparison currently being employed.

Regression test suites can be declared *effective* only if they can detect application bugs. However, current test generation techniques [23, 89, 96] focus exclusively on coverage metrics which are shown to be irrelevant to determine test effectiveness [65]. Since manual seeding of application bugs is impractical for large software, mutation testing has become an accepted norm in establishing the fault-revealing capabilities of test suites.

Unfortunately, there exist no mutation analysis tools for web apps owing to their heterogeneous nature. Existing mutation analysis tools for web applications focus on source code mutation limiting them to a subset of web app components written in the target programming language.

Research Question 3. *How to measure the effectiveness of regression test suites?*

To address this research question, we developed a configurable and extensible mutation analysis framework (Chapter 4) for web apps that can assess the effectiveness of UI test suites irrespective of the nature of underlying individual web components. Our technique mutates the DOM in the browser at runtime instead of source-code in order to make it applicable to all web apps universally.

1.3 Enabling API Testing for Web Applications

Modern web applications make extensive use of API calls to update the UI state in response to user events or server-side changes. In particular, they increasingly rely on web APIs that follow the REST (REpresentational State Transfer) architectural style [53] and are referred to as RESTful or REST APIs. For such applications, API-level testing can play an important role, in between unit-level testing and UI-level (or end-to-end) testing.

API testing places the focus of testing on the operations of a service as well as sequences of operations; it exercises the server-side flows more comprehensively than unit testing but without the need to go through the UI layer. Existing API testing tools require API specifications (e.g., OpenAPI), which often may not be available or, when available, be inconsistent with the API implementation, thus limiting the applicability of automated API testing to web applications.

Research Question 4. *How to enable API testing universally for all web apps?*

We developed a first-of-its-kind approach for carving API test cases and API specifications from UI paths (Chapter 5) that enables API-level testing for web applications, irrespective of the frameworks they use. Our dynamic technique executes the web application via its UI to automatically create (1) API-level test cases that invoke the application's APIs directly, and (2) a specification describing the application’s APIs that can be leveraged for development and testing purposes.

Chapter 2

Near-Duplicate Detection in Web App Model Inference

2.1 Introduction

Automated techniques such as web app crawlers are widely used to reverse-engineer state-based models as a viable vehicle for various analysis and testing tasks such as automated test generation. The *state* in such models represents the dynamic webpage of the app, as represented by the Document Object Model (DOM) in the browser. Crawlers are capable of efficiently exploring a large state space of any given web app. However, an adequate model should contain only the minimal set of *distinct* states that represent the web app functionalities, while discarding insignificant states that do not contribute to exposing new functionality to the end user. Instances of such states are pages only differing by small cosmetic changes, which are also referred to as *near-duplicates* in the literature [43, 44, 64, 86]. To discard such near-duplicate webpages, crawlers have adopted state abstraction functions over the DOM [48, 96, 98, 132] as a proxy for the similarity of webpages. The downside of these abstractions is that minimal changes to the DOM can result in duplicate states in the model, even if such DOM changes are not reflected on the final UI visually, and therefore might not be representative of a new webpage functionality. From an end-to-end (E2E) testing perspective, clone and near-duplicate states in web app models negatively impact their accuracy and completeness, undermining the quality of the test suites generated from such models in terms of size, runtime, and coverage.

Clone and near-duplicate detection *across* different web apps has been an active research topic in many fields [43, 44, 64, 86]. In information retrieval, the content of a webpage has been the primary focus, because the purpose of web search engines is to index and retrieve information from webpages through search queries. Computer vision techniques have been employed to detect visually similar webpages, for instance in phishing detection [6, 30]. Other approaches leverage state abstractions based on the similarity of URLs, textual content and the DOM [24, 130, 155]. Detecting near-duplicate pages is a challenging problem as there is no generally accepted definition of near-duplicate states and there is no unified standard against which a technique can be assessed [63, 64]. A second challenge pertains to the selection of similarity *thresholds* that such techniques need as input to determine when two pages are similar. These thresholds are usually educated guesses, as no systematic means have been proposed so far to estimate them automatically.

In this work, we are interested in detecting distinct states in web app models in the context of functional E2E web testing. Our aim is to study the nature of duplicate states occurring *within* a web app, and provide a systematic approach to selecting thresholds for inferring an optimal model, i.e., having the lowest number of (near-)duplicate states. To this end, we evaluate the capability of 10 near-duplicate detection algorithms in identifying clone, near-duplicate, and distinct web app states. We adopt techniques from three different domains—information retrieval, web testing, and computer vision—where the textual content, the DOM tree, and the visual screenshot of the page are used to measure the similarity between states. Our goal is to assess whether textual, structural, or visual features are related with semantic properties of webpages and provide meaningful means to understanding their degree of functional relatedness from an E2E testing perspective.

To select the similarity thresholds for fine-tuning such techniques, we first crawled 6*k* websites randomly selected from Alexa's top million URLs. We retrieved 493*k* pairs of states belonging to the same application, and computed the similarity distance between these pairs using each near-duplicate algorithm. We then manually classified 1*k* random state-pairs into three categories of clone, near-duplicate, or distinct. We used our empirical data of distances to choose thresholds for each algorithm through statistical and optimization search methods. We evaluated their accuracy in automatically classifying clones and near-duplicates in the remaining unlabelled portion of the dataset. Further, we evaluated these configured algorithms on a subject set of nine unseen web apps, for which manual ground truth models were previously created.

Our work makes the following novel contributions:

- The first study of 10 different near-duplicate detection techniques applied in the context of web app model inference.
- A classification of different categories of near-duplicates occurring within a web app.
- Systematic ways of threshold selection for near-duplicate detection as well as an empirical evaluation of their effectiveness in test models.
- The toolset comprising the 10 near-duplicate detection algorithms, which is available for download [168].
- A dataset of 99*k* manually classified pairs of webpages, of which (1) 1,5*k* pairs are randomly sampled from 6k websites, and (2) 97.5*k* from nine real-size web apps. Our dataset can be used by others to conduct similar near-duplicate detection studies and is also available for download publicly [168].

Our results show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates within apps. In practice, existing near-duplicate detection techniques are *not* designed to find functional similarity in a way that human testers regularly assess while testing web apps. For certain types of near-duplicates, we observed that the model deteriorates over time as the crawl progresses. For instance, although RTED was able to achieve a high accuracy F_1 score of 0.95 initially, the final produced model had only an F_1 of 0.45. This deterioration is due to the accumulation of numerous near-duplicates to the model, which decreases precision. Our results underline the need for further research in devising techniques geared specifically toward web test models, i.e., that can distinguish between different types of near-duplicates such as those found in our study.

2.2 Redundancies in Web App Models

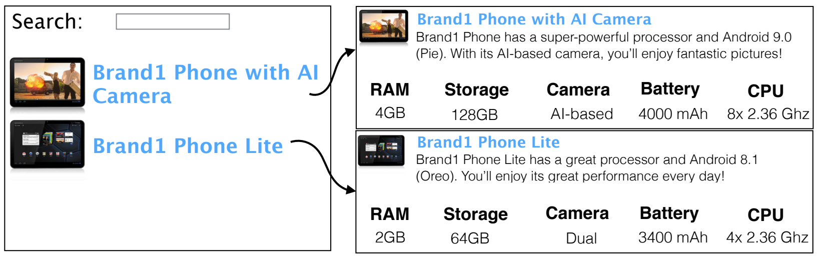
In practice, web testing is often performed in an end-to-end (E2E) fashion, by verifying the correctness of the web app state in response to user events and interactions with the GUI (e.g., clicks, and forms submissions). This task is performed either manually by testers, or by writing test scripts with test automation tools such as Selenium [134].

Automated techniques, on the other hand, generate web test cases from models that are inferred through reverse-engineering techniques. A popular method to model construction for modern web apps is automated state exploration, also known as web app crawling [94, 156]. Such techniques dynamically analyze the web app under test by automatically firing events and checking the webpage for changes. When new state changes are detected, the model is updated to reflect the event causing the new state. Generated models can be represented in various formats such as UML state diagrams, Finite State Machines (FSM), or State-Flow Graphs (SFG) [94, 126, 156].

To avoid redundancies in the model, states that are identical or highly similar to previously encountered states should be discarded. For instance, let us consider Figure 2.1, a web app in which the homepage shows a list of phones. When the user clicks on any of the phones in that list, she is redirected to another web page displaying the detailed characteristics of the selected phone. From a functional testing viewpoint, however, a page containing a list of 20 phones is *conceptually* the same as one listing the same 20 phones plus one extra phone.

The problem of detecting already visited states can be cast as an *equivalence problem*: given two web page states p_1 and p_2 explored by the crawler, a state abstraction function determines whether $p_1 \simeq p_2$. More formally:

Definition 2 (**State Abstraction Function**). *A state abstraction function (SAF) $\mathscr{A}$ is a pair (dist, t), where dist is a similarity function, and t is a threshold defined over the values of dist. Given two web pages, p_1, p_2, $\mathscr{A}$ determines whether the distance between p_1 and p_2 falls below t.*


Search:

Brand1 Phone with AI Camera

Brand1 Phone Lite

Brand1 Phone with AI Camera

Brand1 Phone has a super-powerful processor and Android 9.0 (Pie). With its AI-based camera, you'll enjoy fantastic pictures!

RAM	Storage	Camera	Battery	CPU
4GB	128GB	AI-based	4000 mAh	8x 2.36 Ghz

Brand1 Phone Lite

Brand1 Phone Lite has a great processor and Android 8.1 (Oreo). You'll enjoy its great performance every day!

RAM	Storage	Camera	Battery	CPU
2GB	64GB	Dual	3400 mAh	4x 2.36 Ghz

Figure 2.1: Example of near-duplicate web pages.

$$\mathscr{A}(dist,p_1,p_2,t)\begin{cases} true & : dist(p_1,p_2) < t \\ \\ false & : otherwise \end{cases}$$

In practice, $\mathscr{A}$ is defined based on the similarity of some abstracted notion of the web pages such as their URLs, textual content, DOM structure, or screenshot image. However, the amount and nature of changes occurring in a web page with respect to the functionality of the app is not always directly proportional to the amount of changes in the DOM tree, textual content, or visual aspects of the page.

Let us consider using a crawler equipped with a SAF based on DOM content similarity on our sample web app of Figure 2.1. This SAF is less tolerant to content (textual) changes occurring in web pages. Therefore, each page displaying a new phone's characteristics might be considered a different state and many functionally similar occurrences of already modelled pages (i.e., near-duplicates) would be included in the model. If we use this *inflated model* to generate test cases, the overall functional coverage does not change when the generated tests exercise the phone details page multiple times, thus potentially wasting precious testing time and resources.

On the other hand, another "better" SAF, for instance based on the DOM tree similarity with a proper threshold value, would consider all such phone detail pages as the same, providing a more concise model for the web application of our example. However, a high threshold value might cause other relevant functionality to be abstracted away as well, resulting in an incomplete model.

Near-duplicate detection techniques have been studied for reducing the occurrence of redundant similar pages *across* web apps, e.g., in web search engines [29] or phishing detection [108]. An understanding of whether such techniques apply also in detecting functional near-duplicates *within* the same web app is missing in the literature. Despite its prevalence and importance, this problem is understudied, because it is hard to define a notion of equivalence for two arbitrary webpages. Moreover, in the general case, deciding *a priori* which abstraction function and which threshold would work best for a given web app is a challenging task, as it requires substantial domain-specific knowledge of the web app under test.

Table 2.1: Near-Duplicate detection algorithms included in our study.

Domain	Algorithm	Input	Description	Distance Output
Information Retrieval				
Web Search	simhash [29]	DOM (content)	64 bit fingerprinting technique which uses features extracted from the web page content	Hamming distance of two 64 bit digests
Malware detection	TLSH [108]	DOM (content)	Locality-sensitive 256 bit hashing scheme that is robust to minor variations of the input	Hamming distance of two 256 bit digests
Web Testing				
	RTED [98, 112]	DOM (Tree)	Minimum-cost sequence of node edit operations that transform one DOM tree into another	Tree edit distance value normalized by the sum of nodes in the two trees
	Levenshtein [75, 96]	DOM (String)	Minimum number of single-character edits required to transform one string into another	Edit distance value normalized by the sum of the string lengths
	String Equality (baseline)	DOM (String)	String equality comparison	Boolean value
Computer Vision				
Image Hashing	PHash [179]	Screenshot	128 bit perceptual hash that represent the lowest frequencies of pixel brightness, to which discrete cosine transform (DCT) is applied to retrieve a brightness matrix	Hamming distance of two 128 bit digests
	Block-mean [175]	Screenshot	256 bit perceptual hash obtained by dividing the image into non-overlapping blocks, which are encrypted with a secret key and normalized median value is calculated	Hamming distance of two 256 bit digests
Whole Image Comparison	Histogram [150]	Screenshot	Color distribution of a digital image	χ^2 distance between two color histograms
	PDiff [178]	Screenshot	Adopts a human-like concept of similarity that uses spatial, luminance, and color sensitivity	Number of different pixels normalized by the maximum number of pixels in the two images
Structural Similarity	SSIM [10]	Screenshot	Simulates the high sensitivity of human visual system to structural distortions while compensating for non-structural distortions	Normalized structural distortion value
Feature Detection	SIFT [80]	Screenshot	Computes local feature vectors and image descriptors which are invariant to geometric affine transformations like scaling/ rotation	Number of different key-points normalized by the maximum number of key-points in both images

2.3 Near-Duplicate Algorithms

In this work, we study 10 near-duplicate detection algorithms from three different domains, namely, information retrieval, web testing, and computer vision. Table 2.1 presents the techniques, along with the domain they belong to, the input types, a short description, and their distance output.

2.3.1 Information Retrieval

Near-duplicate detection has been applied to index the massive volume of web pages continuously retrieved by web crawlers for search engines. The overall goal is to select only a relevant set of pages based on the provided user search string. In this setting, performance is the most important factor, therefore hashing mechanisms have been adopted due to their design simplicity and speed of comparison. As an input, the web page content is typically the primary focus when designing algorithms used in this domain.

We chose two content hashing algorithms from this domain: (1) `simhash` [29], a popular and effective web page fingerprinting technique adopted by Google to index web pages [64], and (2) Trend Locality Sensitive Hash (`TLSH`) [108], a hashing technique for fingerprinting source code, employed for malware detection [159].

2.3.2 Web Testing

In the web testing domain, researchers have studied DOM-based abstractions to compare webpages during the crawling of the application under test. The assumption is that two web pages sharing similarities among their DOMs are likely to represent pages having analogous functionalities, hence it is worthwhile to consider them the same. The DOM can be treated either as a tree-like structure, or as a simple string of characters.

We chose three different similarity algorithms over the DOM that have been employed as state abstraction functions in prior web testing research [96, 98, 144]: (1) tree edit distance with the RTED algorithm [112], (2) Levenshtein distance [75] over the string represented by the DOM, and (3) string equality between two DOM strings, which we use as baseline.

2.3.3 Computer Vision

Image similarity is one of the main topics in computer vision. Many techniques have been proposed and studied, at different levels of granularity, ranging from low-level pixel matching up to high-level feature-based matching. These techniques are applied in indexing and searching, summarization, object detection and tracking, facial recognition, and also copyright image detection. We consider different classes of image-based algorithms.

Image hashing techniques map visually identical or nearly-identical images to the same (or similar) digest called image hash. We chose two image hashing algorithms: (1) block-mean hash [175] and (2) perceptual hash (PHash) [179], which have been used in multimedia security for image retrieval, authentication, indexing and copy detection. *Whole image matching* techniques focus instead on individual pixels composing the image. Color-Histogram [129] and Perceptual Diff (PDiff) [178] have been successfully applied in previous web testing work for detecting cross-browser incompatibilities [84]. A downside of those techniques is that they are affected by changes in coordinates of web elements common in responsive web layouts. *Structural similarity* techniques quantify image quality degradation. For instance, Structural Similarity Index (SSIM) [10] has been shown to be effective due to the highly structured nature of web apps' GUIs [30]. Lastly, *feature detection* techniques have been widely employed for near-duplicate image detection. For instance, Scale Invariant Feature Transform (SIFT) [80] has been applied to aid web test repair [146] and phishing detection [6].

To the best of our knowledge, this work is the first to consider and evaluate visual image similarity as a near-duplicate detection technique for web application crawling.

2.4 Empirical Study Design

The goal of our study is to determine how existing near-duplicate detection techniques can be employed to obtain an optimal model of a web application that can be used for E2E testing.

RQ$_1$: *What type of functional near-duplicates exist within apps?*

RQ$_2$: *How well can functional near-duplicates be detected?*

RQ$_3$: *What is the impact of near-duplicates and detection techniques in inferring a web-app model?*

First, in Section 2.5, we randomly sample 1,000 within-app state-pairs from a dataset created by crawling 6*k* randomly selected URLs. We characterize the changes occurring between states within an app and identify how they lead to different classes of functional near-duplicates (RQ$_1$). We label these 1,000 pairs as either clones, near-duplicates or distinct states, and compute the distance between them for all the ten near duplicate techniques described in Section 2.3.

In Section 2.6, using these labelled pairs, we compute statistical and optimal thresholds to fine-tune each near duplicate technique. Through this, we aim to determine whether such randomly sampled distances from a large dataset can be used to automatically classify state-pairs in unseen web apps and detect near-duplicates (RQ$_2$).

In Section 2.7, we determine the best near-duplicate detection techniques and application-specific thresholds to infer web app models for nine open-source web apps covering the different near-duplicate categories. Finally, we analyze these models to determine how different kinds of near-duplicates impact model inference (RQ$_3$).

2.5 RQ$_1$: Near-Duplicates in Web Apps

In order to determine what kinds of functional near-duplicates occur within apps, we first create a dataset of *within app state-pairs* and their calculated distances for each near-duplicate detection algorithm. Then, we manually characterize the nature of differences between pairs of pages and classify them in a random sample.

2.5.1 Dataset Creation

First, we crawl randomly selected website URLs from the top one million as provided by Alexa,[1] a popular website that ranks sites based on their global popularity for a *week* using CRAWLJAX [96], an event-driven crawler for exploring highly

[1] http://www.alexa.com

dynamic web apps. We configured CRAWLJAX to run using the Chrome browser, with its default simple state abstraction function, namely string equality (see Table 2.1), and a runtime limit of five minutes for each crawl.

To account for network communication errors and the tool's exploration limitations, e.g., on sites that require login credentials, we filtered out sites for which the crawl models obtained contained less than 10 states. After this filtering stage, we retained 1,064 different sites accounting for 30,202 states from the original 6,359 web crawls. We then created all possible 677,415 pair-wise combinations of states *within each crawl*, which we call *state-pairs*.

Computing Distances

We computed the distance for each state-pair using each of the 10 algorithms presented in Table 2.1. We discarded the state-pairs for which the distance could not be computed correctly, such as the case of DOM-based tree edit distance of malformed HTML trees. The final dataset, called $\mathscr{DS}$, contained 1,031 sites and 29,704 states, from which 493,088 state-pairs with properly computed distances were obtained.

Distance Normalization

The raw distances which quantify the difference between two given pages have different output spaces based on the page characteristic used by the technique. As an example, given a state-pair of web pages, PDiff outputs the number of perceptually different pixels between their screenshots, whereas BlockHash returns the hamming distance between image hashes. For the sake of comprehensibility, we normalized all distances computed by each algorithm, as described in the *Distance Output* column of Table 2.1, but we never compare outputs of different techniques.

2.5.2 Classification of Changes

To gain a better understanding of what changes within web pages characterize near-duplicates, we classify the differences of the state-pairs in our dataset from the point of view of a human tester who is interested in functionality coverage.

Procedure

Manually examining state-pairs is a time consuming task requiring familiarity with the functionality of the application. Therefore, we randomly sampled a set, called $\mathscr{RS}$, of 1,000 state-pairs from our final dataset of 493,088 state-pairs, which allows us to have a confidence level of 99% with a 4% margin of error in deriving a representative statistic. For each state-pair $(p_i, p_j) \in \mathscr{S}$, the authors of the paper visually analyzed, in isolation, the screenshot images (and the original web pages where necessary) of the two web app states from a functional testing perspective, to obtain a set $\mathscr{D}$ of differences. Each difference in $\mathscr{D}$ is defined as $\Delta(p_i, p_j) = \{\delta(e_i, e_j)\}$ where $\delta(e_i, e_j)$ is a pair of non-identical web elements in which $e_i \in p_i$ and $e_j \in p_j$. Finally, each author assigned a descriptive label to each detected difference.

Difference Categorization

After enumerating all differences across the 1,000 state-pairs in $\mathscr{RS}$, the authors reviewed them together to resolve conflicts and reached consensus on equivalence classes of differences. Our study revealed the following categories.

Definition 3 (**Unrelated** (U))**.** Given a difference $\delta(e_i, e_j)$, neither of e_i or e_j are related to any functionality offered by the web app.

Examples of these differences include changes in background images, or GUI widgets related to advertisement (see red ovals in Figure 2.2a).

Definition 4 (**Duplicated** (D))**.** Given a difference $\delta(e_i, e_j)$, e_i and e_j replace each other in the original pages p_i and p_j without adding any new functionality to either page.

Two distinct subcategories of duplicated differences emerged:

- *Replacement* (D_1): $D_1 : e_i \equiv e_j$ meaning the difference represents a functionality or content that is equivalent. For instance, in Figure Figure 2.2b, the red ovals highlight equivalent content.

- *Addition* (D_2): $D_2 : e_i = \emptyset \wedge \exists e'_i \in p_i : e'_i = e_j \vee \delta(e'_i, e_j) \models D_1$ meaning the non-empty e_x in δ has a duplicate e_y in the same page, and therefore its addition does not affect the overall functionality of the page. For example, in

Figure Figure 2.2c, the oval identifies a duplication of an existing functionality.

Definition 5 (New (*N*)). Given a difference $\delta(e_i, e_j)$, δ represents a new functionality or a semantically different content, i.e.:
$\delta(e_i, e_j) \models N : (e_i = \emptyset) \wedge (\nexists e'_i \in p_1 s.t. e'_i = e_j \vee \delta(e'_i, e_j) \models D)$.

For example, the search box in Figure 2.1 is absent in phone description pages and is an example of new functionality.

State-Pair Classification

Following the classification of differences described above, we classified state-pairs from a functional point of view, in three distinct categories defined as follows.

Definition 6 (Functional Clone (*Cl*)). Given two web pages p_1 and p_2, the state-pair (p_1,p_2) is a functional clone (*Cl*) if there are no semantic, functional or perceptible differences between them, defined as $Cl : \Delta(p_1, p_2) = \emptyset$.

Definition 7 (Functional Distinct (*Di*)). Given two web pages p_1 and p_2, p_1 is functionally distinct from p_2 if there is any semantic or functional difference between the two pages, $Di : \exists \delta(e_i, e_2) \models N$.

Definition 8 (Functional Near-Duplicate (*Nd*)). Given two web pages p_1 and p_2, p_1 is a functional near-duplicate of p_2 if the changes between the states do not change the overall functionality being exposed: $Nd : \Delta \not\models Cl \wedge \nexists(\delta(e_1, e_2) \models N) \in \Delta$.

We further observed three fine-grained subclasses of near-duplicates in our dataset.

Cosmetic (Nd_1) when changes related to the aesthetics of the webpage such as advertisements or background images occur, which leave the functionalities unaltered (see Figure Figure 2.2a): $Nd_1 : \Delta(p_1, p_2) \ni \delta(e_1, e_2) \models U$

Dynamic data (Nd_2) when both states of the pair are generated from the same template and populated with dynamic data, according to a user query or app business logic (see Figure Figure 2.2b): $Nd_2 : \Delta(p_1, p_2) \ni \delta(e_1, e_2) \models D_1 \vee U$

Duplication (Nd_3) when there are additional web elements in a page the functionality and semantics of content of which is entirely represented within the other page (see Figure 2.2c): $Nd_3 : \exists\delta(e_1, e_2) \models D_2 \in \Delta(p_1, p_2)$

Following these definitions, we manually labelled the 1,000 state-pairs in $\mathscr{RS}$, and found 441 clones, 275 near-duplicates (45 Nd_1, 219 Nd_2, 11 Nd_3), and 284 distinct pairs.

2.6 RQ$_2$: Classification of state-pairs

2.6.1 Subject Systems

To address RQ$_2$ (and later RQ$_3$), we need to infer models with different algorithms and thresholds numerous times, which requires web apps with deterministic behaviours.

To this aim, we selected nine open-source web apps (Table 2.2) used in previous research of web testing [22, 23, 144, 145], as *subjects*: Claroline (*v. 1.11.5*) [34], Addressbook (*v. 8.2.5*) [116], PPMA (*v. 0.6.0*) [14], MRBS (*v. 1.4.9*) [103] and MantisBT (*v. 1.1.8*) [15] are open-source PHP-based applications while Dimeshift (*commit 261166d*) [46], Pagekit (*v. 1.0.16*) [110], Phoenix (*v. 1.1.0*) [115] and PetClinic (*commit 6010d5*) [13] are web apps that cover popular JavaScript frameworks *Backbone.js*, *Vue.js*, *Phoenix/React* and *AngularJS*, respectively.

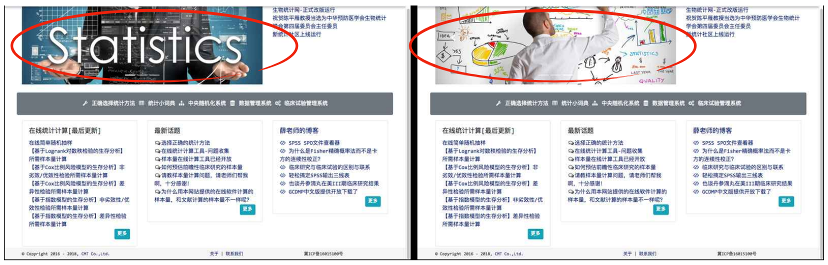
Note that these nine subject apps are not part of the dataset $\mathcal{DS}$.

2.6.2 Manual Classification (Ground Truth)

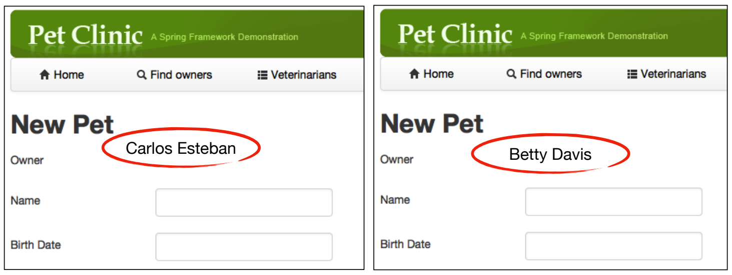
We set out to create manually labelled models for each subject, which we can use as ground truths for comparison of techniques.

First, we use CRAWLJAX to create a master crawl model with default depth-first exploration strategy, default state abstraction function based on DOM string equality, and a maximum time budget of one hour, which allow us to capture a large portion of each app's state space.

Next, we created state-pairs from the states in each model, as follows. The



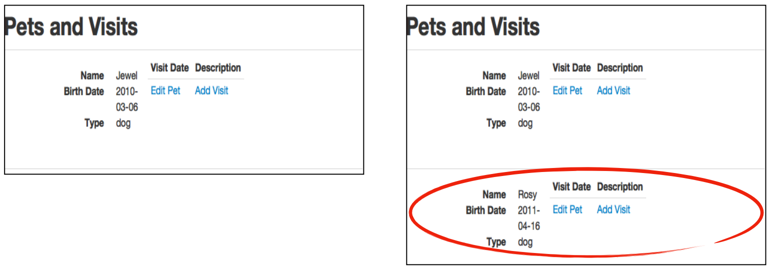
(a) *Near-Duplicate (Nd_1): Background Image Changes*



New Pet	
Owner	Carlos Esteban
Name	
Birth Date	

New Pet	
Owner	Betty Davis
Name	
Birth Date	

(b) *Near-Duplicate (Nd_2): Dynamic Data*


Pets and Visits

		Visit Date	Description
Name	Jewel		
Birth Date	2010-03-06	Edit Pet	Add Visit
Type	dog		

Pets and Visits

		Visit Date	Description
Name	Jewel		
Birth Date	2010-03-06	Edit Pet	Add Visit
Type	dog		

		Visit Date	Description
Name	Rosy		
Birth Date	2011-04-16	Edit Pet	Add Visit
Type	dog		

(c) *Near-Duplicate (Nd_3): Duplicated Functionality*

Figure 2.2: Different subclasses of near-duplicate state-pairs.

authors of this work *manually classified* each state-pair into a clone, near-duplicate (with subcategories) or distinct category, following the same procedure described in Section 2.5.2. In addition, we also assigned each state to a *bin* that represents a part of the application's state space devoted to a certain functionality. As such, each bin is a logical container for all dynamically generated concrete webpages

Table 2.2: Subject Set with Manual Classification

	Bins	**States**	**Pairs**	**Clones**	**Near-Duplicates**			**Distinct**
					Nd_2	*Nd_3*	*Total*	
Addressbook	25	131	8,515	26	52	2,295	2,347	6,142
PetClinic	14	149	11,175	2	1,433	180	1,613	9,411
Claroline	36	189	17,766	2,707	71	0	71	14,988
Dimeshift	21	153	11,628	375	570	0	570	10,683
PageKit	20	140	9,730	0	904	3,044	3,948	5,782
Phoenix	10	150	11,175	1	25	4,580	4,605	6,569
PPMA	23	99	4851	64	467	0	467	4,320
MRBS	14	151	11,325	27	4,044	0	4,044	7,254
MantisBT	53	151	11,325	2	1,117	0	1,17	10,206
Total	216	1,313	97,490	3,204	8,683	10,099	18,782	75,355

upon crawling (e.g., all webpages related to *login*). We consider the first concrete instance of a bin B to be a *coverage of* B by that crawl model. Additional concrete instances of a bin are considered clones or near-duplicates of the bin B.

Table 2.2 shows the master crawl characteristics for each web app as well as our classification outcome. In the rest of this chapter, we refer to the nine master crawls with manually classified 97.5k state-pairs of the nine apps as *subject set* ($\mathcal{SS}$), and to our manual classification and identified bins as *ground truth*. Our classification of the subject-set did not find any near-duplicates of category Nd_1 in $\mathcal{SS}$ as the subjects did not feature unrelated changes (U) such as advertisements, commonly found in other kind of websites. MantisBT has the most bins (53), representing a state-space five times bigger than that of Phoenix, which has the smallest number of bins (10). Addressbook, PageKit and Phoenix have a high number of near-duplicates of category Nd_3, differently from the other six. To study how different near-duplicate categories impact web-app model inference, we group these three subjects referring to them as *Nd3-Apps* and the other six as *Nd2-Apps*.

Table 2.3 compares the subjects webpage characteristics in terms of DOM size, complexity, and image size to *DS*. For example, the content of a web page in $\mathcal{DS}$ on an average is almost eight times that of the web pages in $\mathcal{SS}$.

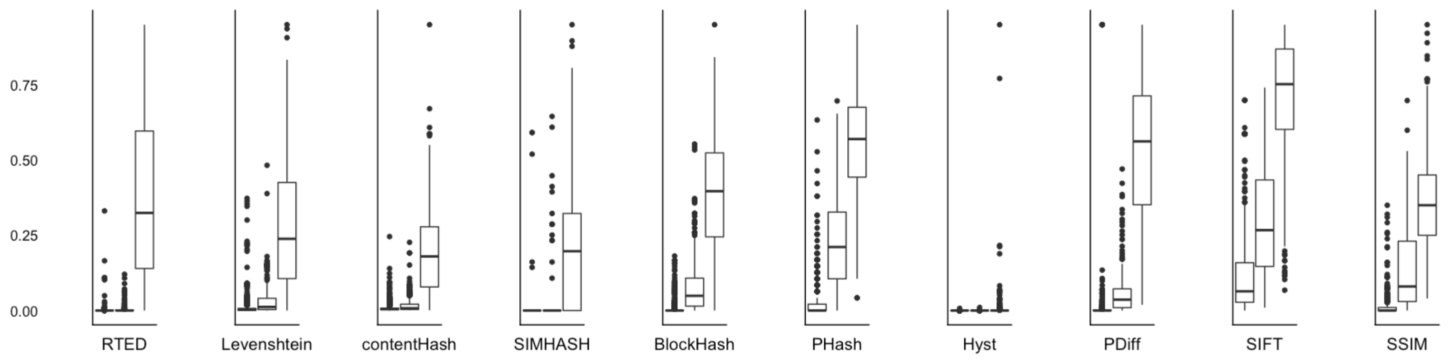


Figure 2.3: Normalized Distance distribution of labelled pairs in the dataset $\mathcal{DS}$. Within each box-plot, from left to right: clone, near-duplicate and distinct pairs.

Table 2.3: Average webpage characteristics
state (DOM and Screenshot) across the two datasets

	DOM			IMAGE
	Tree (# nodes)	**Source** (length)	**Content** (length)	**Pixels** (#)
Dataset ($\mathcal{DS}$)	810	105,445	45,575	3,575,837
Subjects ($\mathcal{SS}$)	290	17,655	6,216	1,190,230

2.6.3 Threshold-Based Classification

We aim to evaluate the effectiveness of the near-duplicate detection algorithms in classifying a given pair as either clone, near-duplicate, or distinct. Essentially, this is a multi-class classification problem, which we propose to solve using a classification function Γ. Function Γ takes as inputs a near-duplicate detection algorithm f and computes the distance between two given states in a state-pair (p_1, p_2), classifying the pair to a category according to a threshold-pair (t_c, t_n), as follows:

$$\Gamma(p_1, p_2, f, t_c, t_n) \begin{cases} Cl \ : f(p_1, p_2) < t_c \\ \\ D \ \ : f(p_1, p_2) > t_n \\ \\ Nd : otherwise \end{cases}$$

To evaluate Γ, we need to find appropriate threshold values for each algorithm that maximize the classification scores.

Threshold Determination

We employ two different approaches, namely, *statistical* and *optimization*, to find a suitable threshold-pair (t_c, t_n) for each algorithm. In the statistical approach, we follow a data-based approach in which we use the distance distributions of different classes (Figure 2.3). In the optimization approach, instead, we determine the thresholds that maximize the classification score on a given labelled set, a commonly adopted strategy in machine learning for hyper-parameters selection of predictive models [136].

Definition 9 (**Statistical Threshold Pair** (St_c, St_n))**.** Threshold St_c is the 3rd quartile (Q_3) of the distances calculated by a technique on a given set of clone state-pairs, whereas, threshold St_n is the *median* distance on a given set of near-duplicate state-pairs.

Definition 10 (**Optimal Threshold Pair** (O_c, O_n))**.** Given a labelled set of clones, near-duplicates and distinct state-pairs, the optimal thresholds O_c and O_n are retrieved by a Bayesian optimization search that maximizes the average F_1 classification score for Γ over all three classes.

Figure 2.3 shows the distribution of distance values among the three classes, for each considered algorithm. As the box-plots show, a clear separation between distance values among classes emerged upon statistical analysis (despite some overlaps caused by outliers), which motivates using this data to determine statistical thresholds on $\mathcal{RS}$. For instance, clones (left-most plot for all techniques) have low distances, whereas distinct pairs have high distance scores. Near-duplicates, as expected, lie in between those two categories for all 10 techniques considered in our study. We use quartile data for choosing thresholds since prior work [76] has shown that the median value is a better estimator of the central tendency than mean in such cases.

We refer to the four thresholds $\{St_{c\_DS}, St_{n\_DS}, O_{c\_DS}, O_{n\_DS}\}$ as *universal thresholds*, as the state-pairs in $\mathcal{DS}$ represent a large set of randomly selected real-world webpages (see Section 2.5.1).

Classification Accuracy

To address RQ$_2$, we evaluate the algorithms by comparing the effectiveness of Γ (Section 3.5.1) with corresponding state-pair inputs. We evaluate the effectiveness of Γ using the F_1 measure, which is the harmonic mean of precision Pr (ratio of correctly classified pairs to total number of classified pairs in each class), and recall Re (ratio of correctly classified pairs to the actual number of pairs that belong to the class).

Since we have more than two classes, we treat it as a multi-class classification problem, and obtain the average F_1 over the scores of all three classes (Cl, Nd, D). However, the datasets are unbalanced, i.e., the ratio of state-pairs of the classes

Table 2.4: Estimated statistical (St) and optimal (O) thresholds for clone (c) and near-duplicate (n) bounds, in dataset $\mathcal{DS}$

	St_{c_DS}	St_{n_DS}	O_{c_DS}	O_{n_DS}
TLSH	0.00794	0.00794	0.01742	0.07052
Levenshtein	0.00638	0.01089	0.00704	0.07029
RTED	0.00000	0.00000	0.00007	0.04099
SimHash	0.00000	0.00000	0.00044	0.00108
BlockHash	0.00000	0.04082	0.00301	0.13371
HYST	6.52E-11	1.29E-09	1.15E-09	1.49E-08
PDIFF	0.00160	0.03800	0.00120	0.20080
PHASH	0.01754	0.17544	0.04018	0.32232
SIFT	0.16691	0.27993	0.10192	0.61876
SSIM	0.01000	0.08000	0.02020	0.15560

are not equal; hence, we employ macro-averaging, to avoid favouring classes with higher representation [138]. We calculate the F_1 score of each algorithm using Γ with the universal thresholds (see Table 2.4) on two disjoint inputs: 1) a manually labelled random sample of 500 state-pairs, $\mathcal{TS}$, from the dataset $\mathcal{DS}$, and 2) the *97.5k* labelled pairs from $\mathcal{SS}$.

While the scores on $\mathcal{TS}$ can validate these thresholds, scores on $\mathcal{SS}$ assess the viability of discovering universal thresholds for a near-duplicate detection algorithm for unseen web apps.

Findings (RQ$_2$)

Table 2.5 shows the F_1 classification scores for all techniques on the two labelled sets, $\mathcal{TS}$ and $\mathcal{SS}$. As a baseline to compare the techniques, we use a *stratified-random-classifier* [3] that classifies each state-pair randomly based on proportions of classes in the labelled set.

All evaluated techniques perform better on $\mathcal{TS}$ than $\mathcal{SS}$ when *universal thresholds* are used (+15% on average). This result is not surprising as $\mathcal{TS}$ is sampled from $\mathcal{DS}$, as well as $\mathcal{RS}$ from which we derived these thresholds. $\mathcal{SS}$, on the other hand, is completely disjoint and different from $\mathcal{DS}$ (Table 2.3).

Although *statistical* and *optimal* thresholds have similar overall average F_1

Table 2.5: F_1 Measure for Statistical and Optimal threshold sets

Algorithm	**statistical** (St_{c_DS},St_{n_DS})			**optimal** (O_{c_DS},O_{n_DS})			**All**		
	$\mathcal{TS}$	$\mathcal{SS}$	Avg	$\mathcal{TS}$	$\mathcal{SS}$	Avg	$\mathcal{TS}$	$\mathcal{SS}$	**Avg**
TLSH	0.50	0.40	0.45	0.56	0.44	0.50	0.53	0.42	**0.48**
Levenshtein	0.54	0.46	0.50	0.59	0.48	0.54	0.57	0.47	**0.52**
RTED	0.50	0.45	0.47	0.57	0.50	0.54	0.53	0.48	**0.50**
SIMHash	0.48	0.17	0.33	0.48	0.17	0.33	0.48	0.17	**0.33**
BlockHash	0.62	0.54	0.58	0.66	0.50	0.58	0.64	0.52	**0.58**
HYST	0.52	0.37	0.44	0.57	0.31	0.44	0.55	0.34	**0.44**
PDIFF	0.63	0.57	0.60	0.67	0.53	0.60	0.65	0.55	**0.60**
PHASH	0.59	0.43	0.51	0.63	0.40	0.52	0.61	0.41	**0.51**
SIFT	0.59	0.44	0.52	0.61	0.47	0.54	0.60	0.45	**0.53**
SSIM	0.62	0.53	0.57	0.65	0.48	0.56	0.64	0.50	**0.57**
Average	**0.56**	**0.44**	**0.50**	**0.60**	**0.43**	**0.51**	**0.58**	**0.43**	**0.51**
Random	0.32	0.32	0.32	0.32	0.32	0.32	0.32	0.32	**0.32**

scores (*0.50*, *0.51*), it is important to notice that optimal thresholds perform worse than statistical thresholds on $\mathcal{SS}$, contrary to expectation.

These two findings essentially indicate that, the distance thresholds for optimal classification of state-pairs can vary based on the characteristics of the particular web app. The thresholds obtained from a labelled data such as $\mathcal{RS}$ are therefore, not necessarily applicable for a random unseen web app. Hence, *universal thresholds that can classify any given state-pair may not be feasible*.

Amongst the techniques, SimHash has the lowest average F_1 score (*0.17*) on $\mathcal{SS}$, almost 90% worse than the random baseline. The results concur with findings of a previous study [64], which points to the fact that the algorithm is poor at distinguishing states that belong to the same app.

On average, *five out of top six* techniques belong to the computer vision domain. *PDIFF* is the best with a classification F_1 score of *0.60*, more than **85%** better than the baseline and **13%**, **20%** better than Levenshtein and TLSH, the best techniques in DOM and IR categories, respectively. On average, most visual techniques outperform DOM and IR techniques (with the exception of *PHash* and *color-histogram*). On $\mathcal{SS}$, *PDIFF* again outperforms all techniques while *Block-*

Hash and *SSIM*, both visual, are the only other techniques that have an F_1 score of more than *0.50*.

2.7 RQ$_3$: Impact on Inferred Models

With RQ$_3$, we evaluate the impact of the near-duplicate detection algorithms in automated web app model inference.

RQ$_{3.1}$: *How can classification thresholds be applied to state abstraction functions (SAFs)?*

RQ$_{3.2}$: *Can domain knowledge be employed to improve the obtained models?*

RQ$_{3.3}$: *How does efficiency of SAFs impact the obtained models?*

Specifically, we evaluate the quality of crawl models inferred using each of the near-duplicate detection algorithms as *state abstract function (SAF)* (see Definition 14) along with the determined thresholds. CRAWLJAX already includes all DOM-based algorithms described in Section 2.3.2; we added the computer vision and information retrieval near-duplicate algorithms within CRAWLJAX as SAFs. More specifically, we integrated the implementations of PDiff, SIFT, and SSIM from the open-source computer-vision library OpenCV, and the publicly available versions of TLSH[2] and `simhash`.[3]

Since we need to run and analyze many crawl sessions (i.e., nine apps, 10 algorithms, different threshold sets), we limit the crawl session with a *maximum runtime* of five minutes.

Model Quality. We measure the quality of a generated model through its F_1 score, the harmonic mean of *Pr* and *Re*. Lower precision (*Pr*) denotes a greater redundancy in the model and is computed as the ratio of unique states (*bins*) covered by the model to the total number of states in the model. Recall (*Re*) quantifies the application state coverage achieved in the model and is computed as the number of *bins* covered by the model to the total number of *bins* identified by humans, for the corresponding app, in the *ground truth* (see Section 2.6.2).

[2] https://github.com/idealista/tlsh

[3] https://github.com/albertjuhe/charikars_algorithm

The recall *Re* of a crawl model is highly dependent on the ability of the SAF to reliably distinguish the distinct state-pairs and its precision *Pr* on its ability to exclude near-duplicates and clones of states already present from the model. Crawlers, however, typically expect one single similarity threshold for deciding if a state is new to be added to the model; i.e., they do not distinguish between clone/near-duplicate. Therefore, we frame the problem of finding optimal thresholds for a SAF as maximizing the F_1 score of its *distinct-pair detection*.

2.7.1 Thresholds for SAFs ($\mathbf{RQ_{3.1}}$)

Before we employ the near-duplicate techniques as SAFs in crawling and evaluate the generated models, which is a manual and time consuming process, we assess the techniques and the *universal* thresholds based on the F_1 score of the distinct-pair detection, which indicates the applicability of the techniques as SAFs.

Findings ($\mathbf{RQ_{3.1}}$). In the distinct state-pair detection scores from RQ_2 shown in Table 2.6, scores on $\mathcal{TS}$ allow us to assess the ability of a technique to distinguish distinct state-pairs in the wild, while $\mathcal{SS}$ lets us simulate each technique as a SAF on generated models captured in our *subject-set*. In contrast to the RQ_2 results, where both the threshold sets had better average classification F_1 on $\mathcal{TS}$ compared to $\mathcal{SS}$, Table 2.6 shows that *statistical threshold* had better distinct state-pair detection F_1 of 0.78 on $\mathcal{SS}$ than 0.73 in $\mathcal{TS}$. Optimal threshold O_{n_DS}, which is higher/stricter than St_{n_DS}, in terms of actual threshold value, as shown in Table 2.4, has a poor *recall* on $\mathcal{SS}$ (53%) compared to $\mathcal{TS}$ (81%). Also in $\mathcal{TS}$ statistical threshold has the highest recall, but by sacrificing precision; the optimal threshold emerges with a better overall F_1 score through a 25% better precision on $\mathcal{TS}$. The same threshold, however, could not improve precision in $\mathcal{SS}$ but has 50% lower recall.

As we optimized our threshold to be stricter to fit the distribution in $\mathcal{DS}$, we ended up misclassifying distinct pairs to be near-duplicates in $\mathcal{SS}$ because of the differences in the distributions between the two data-sets.

Table 2.6: Distinct pair (Pr, Re, F_1) on existing datasets

	$\mathcal{TS}$			$\mathcal{SS}$			Average		
	Pr	Re	F_1	Pr	Re	F_1	Pr	Re	F_1
O_{n_DS}	0.81	0.81	0.80	0.89	0.53	0.64	0.85	0.67	0.72
St_{n_DS}	0.63	0.90	0.73	0.87	0.76	0.78	0.75	0.83	0.76

As we pointed out in RQ_2, these results show the infeasibility of finding universal thresholds as the distances for state-pairs are highly influenced by the intrinsic characteristics of the web app they belong to.

2.7.2 Using Application Knowledge ($RQ_{3.2}$)

These results for universal thresholds prompted us to investigate whether having knowledge of the web app characteristics helps in selecting better thresholds to improve the detection rates of the techniques.

We use the manually labelled models (see Section 2.6.2) in the subject-set ($\mathcal{SS}$) for each app to represent application knowledge. In order to use this application knowledge, we apply the near-duplicate threshold definitions in Definition 9 and Definition 10 to each subject in $\mathcal{SS}$ to derive $St_{n\_\mathcal{SS}}$ and $O_{n\_\mathcal{SS}}$ respectively. In addition to these two thresholds, through initial experiments, we have observed that category Nd_3 near-duplicates overlap with distinct (Di) pairs and it is not possible to design a threshold that can distinguish them. We therefore created a new threshold definition that sacrifices the precision of distinct pair detection by allowing misclassification of Nd_3 near-duplicates as Di for better recall (Re).

Definition 11. St_{n3} is defined as the *median* of the data distribution of manually labelled near-duplicates $\{Nd_1 \vee Nd_2\}$. In other words, St_{n3} is St_n computed after excluding Nd_3 near-duplicates.

We refer to these thresholds obtained by applying application knowledge in $\mathcal{SS}$ for each algorithm as *app-specific thresholds*. We crawled each of our subjects with two universal and three app-specific thresholds with each technique as a SAF, separately, and assess the quality of the generated models.

Findings ($\mathbf{RQ_{3.2}}$). Table 2.7 shows the average F_1 of crawls for all algorithms for each threshold. Overall, as expected, the universal optimal near-duplicate threshold O_{n_DS} has the worst score of *0.24*; only half of the 0.42 scored by the best threshold O_{n_SS}, the optimal threshold derived with application knowledge. On average, app-specific thresholds improve the model quality by *34%* compared to universal thresholds underlining the need to *consider app characteristics to choose thresholds*. For *Nd3-Apps*, it can be seen that St_{n3_SS} derived using the statistical Definition 11 significantly (90%) improves the F_1 score over the St_{n_SS}, showing that *threshold design needs to consider fine-grained near-duplicate categories prevalent in the app under test.*

Application knowledge improves generated models.

Table 2.7: Inferred model F_1 score

	Universal			App-Specific			
	St_{n_DS}	O_{n_DS}	Avg	St_{n_SS}	St_{n3_SS}	O_{n_SS}	Avg
AddressBook	0.33	0.27	0.30	0.17	0.46	0.41	0.34
PetClinic	0.36	0.25	0.31	0.50	0.50	0.52	0.51
Claroline	0.30	0.18	0.24	0.42	0.42	0.44	0.43
DimeShift	0.31	0.22	0.26	0.33	0.33	0.38	0.34
PageKit	0.30	0.27	0.29	0.27	0.39	0.37	0.34
Phoenix	0.44	0.29	0.37	0.24	0.47	0.42	0.38
PPMA	0.31	0.19	0.25	0.49	0.49	0.51	0.49
MRBS	0.37	0.35	0.36	0.43	0.43	0.46	0.44
MantisBT	0.24	0.18	0.21	0.26	0.26	0.27	0.26
Average	0.33	0.24	0.29	0.34	0.41	0.42	0.39
Nd2-Apps	0.32	0.23	0.27	0.40	0.40	**0.43**	0.41
Nd3-Apps	0.36	0.28	0.32	0.23	**0.44**	0.40	0.35

Table 2.8: Inferred model F_1 for each algorithm for selected thresholds

Thresholds	Apps	TLSH	SIMHash	Levenshtein	RTED	BlockHash	PHASH	HYST	PDIFF	SIFT	SSIM	Average
All Five	All	0.10	0.05	0.47	**0.62**	0.46	0.39	0.41	0.34	0.34	0.35	0.35
	Nd2	0.10	0.04	0.48	0.62	0.47	0.39	0.41	0.36	0.31	0.39	0.36
	Nd3	0.10	0.06	0.43	0.62	0.43	0.40	0.41	0.29	0.39	0.28	0.34
O_{n_SS}	All	0.15	0.08	0.48	0.55	0.54	0.49	0.54	0.45	0.42	0.51	0.42
	Nd2	0.17	0.08	0.53	0.61	0.52	0.49	0.58	0.46	0.37	0.52	**0.43**
	Nd3	0.10	0.10	0.37	**0.43**	0.58	0.49	0.45	0.42	0.52	0.51	0.40
St_{n3_SS}	All	0.09	0.03	0.46	0.67	0.57	0.50	0.55	0.43	0.36	0.48	0.41
	Nd2	0.08	0.02	0.47	0.62	0.55	0.50	0.53	0.44	0.35	0.46	0.40
	Nd3	0.10	0.07	0.44	**0.76**	0.60	0.51	0.60	0.42	0.37	0.51	**0.44**

Table 2.8 shows the average F_1 scores for each algorithm for five minute crawls on our subjects. *RTED* consistently outperforms other techniques with an F_1 score of 0.62 averaged over all five thresholds. it is 29% better than Levenshtein, the next best algorithm.

The results for visual techniques in Table 2.8 are contrary to our expectation, given that, in RQ_2, they convincingly outperformed the DOM and IR techniques in state-pair classification using Γ. Apart from being slow compared to DOM based algorithms as shown in Table 2.9, visual techniques, rely on characteristics that cannot directly capture differences *corresponding to web elements* (e.g., SIFT keypoints). Techniques such as RTED, which use a DOM characteristic on the other hand, can reliably capture differences in individual web elements between given two web pages, essential to be able to classify states similar to a human tester.

In IR techniques, SimHash is not able to distinguish even two completely different states in our subject-set as already seen in RQ_2. TLSH on the other hand, fails to calculate digests for app states of our subjects due to lack of enough complexity as shown in Table 2.3 — the content in our subjects is 1/9th of the content size in the wild. Therefore, we exclude SimHash and TLSH from further analysis.

2.7.3 Impact of Efficiency ($RQ_{3.3}$)

An analysis of visited states per minute or *speed* of the algorithms, shown in Table 2.9, seems to suggest that faster algorithms such as *RTED* (25 states per minute) could explore more states in a given crawl time and improve its *Re* wheras, slower algorithms such as *PDiff*, which could only explore *four* states per minute on an average are at a clear disadvantage.

Table 2.8 shows that for all remaining eight techniques with the exception of SIFT, O_{n_SS} for Nd2-Apps and St_{n3_SS} for Nd3-Apps is the best threshold configuration.

Table 2.9 shows the statistics of the 5-min crawls for each technique with their best threshold configuration. Coverage (*Re*) data suggests that 5 minutes was not enough to cover all of the app state-space. Therefore, we experiment with a longer crawl time, i.e., 30 minutes. Given the exponential nature of increase in manual effort to analyze larger crawl models, we limit this experiment to the best perform-

Table 2.9: Techniques *Speed* and Inferred model (*Re*, *Pr*, F_1)
for best 5-minute crawls

	Levenshtein	**RTED**	**BlockHash**	**PHASH**	**HYST**	**PDIFF**	**SIFT**	**SSIM**
Speed	11	**25**	17	16	16	**4**	5	8
Recall	0.42	**0.61**	0.49	0.49	0.55	0.30	0.28	0.39
Precision	**0.84**	0.79	0.75	0.79	0.72	**0.91**	0.71	**0.85**
F1	**0.54**	**0.66**	**0.54**	0.52	**0.58**	0.44	0.39	0.51

Table 2.10: Inferred model F_1 for 30-Minute crawls

Apps	**BlockHash**	**Hyst**	**Levenshtein**	**PDiff**	**RTED**	**SSIM**
All	0.51	0.57	0.53	0.52	0.62	0.56
Nd2	0.57	0.62	0.59	0.51	0.66	0.52
Nd3	0.39	0.47	0.42	**0.56**	0.52	**0.64**

ing techniques tuned with thresholds from the best 5-minute crawls presented in Table 2.9. We select the top four techniques based on F_1 scores, however, as discussed before, since the slower algorithms were placed at a disadvantage in the 5-minute crawls, we also include PDiff and SSIM that produced models with the best precision (*Pr*) scores of 0.91 and 0.85 (respectively 12% and 6% better than RTED which has the best F_1 score of 0.66).

Findings (RQ$_{3.3}$). Average F_1 scores shown in Table 2.10 for 30 minute crawls indicate that, when tuned correctly and given enough time, Histogram, BlockHash, RTED and Levenshtein can all perform well on Nd2-Apps meaning that they managed to discard near-duplicates of type Nd_2 reasonably well. However, it is surprising to see that PDiff and SSIM score higher than all of them on Nd3-Apps. Thus, we decided to analyze how F_1 has changed over the 30 minutes for Nd3-Apps as opposed to the Nd2-Apps.

A plot of F_1 of the model over its states percentage for RTED crawls is shown in Figure 2.4. The figure highlights that for *Nd3-Apps, the model deteriorates as states being added are near-duplicates, mostly of type Nd_3*, while, the models of Nd2-Apps seem to stabilize as Nd_2 near-duplicates are being detected and discarded. During the manual analyses of models, we observed that the Nd_3 near-duplicates are dynamically created, typically through user-interactions that result in addition/removal of web elements whose functionality already exists in the state (e.g., addition/deletion of new rows in a table). Not only is this newly created state a near-duplicate that will eat into precious testing time, but each time the crawler revisits this state, it may invoke the same creation path adding even more duplicates resulting in a never-ending loop.

Efficiency may negatively impact the generated model in time-limited crawls for Nd_3 apps.

Given that RTED is the best algorithm and was fine-tuned to produce best model for each application, this surprising revelation points to the limitation of existing crawlers and threshold based SAFs and shows that *threshold based crawling may never produce an accurate and complete model of modern web apps with dynamic Nd3 near-duplicates*. We therefore think that future SAFs should incorporate characteristics that represent functionality and crawlers should be designed to utilize near-duplicate detection to establish the nature of duplication instead of quantifying the computed differences to actively guide the exploration to discover newer functionality.

2.8 Threats to Validity

External validity threats concern the generalization of our findings. We considered only nine web apps and experiments with other subject systems are necessary to fully confirm the generalizability of our results, and corroborate our findings. We tried to mitigate this threat selecting real-world web apps with different sizes, pertaining to different domains, and adopted in previous web testing

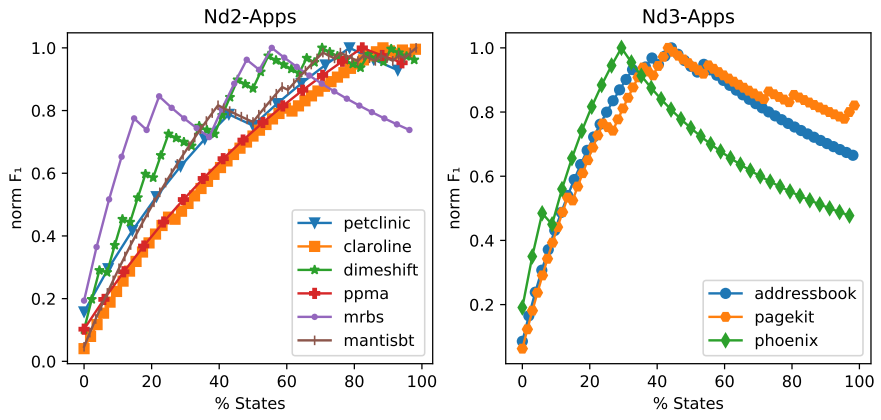


Figure 2.4: Normalized F_1 over %(states in model) during 30-minute crawls of RTED

work [22, 23, 144]. Another threat concerns the selection of thresholds for near-duplicate detection techniques, whose results may not generalize to other algorithms. We mitigated this threat by selecting 10 techniques from three different domains: web testing, computer vision and information retrieval. *Internal validity* threats concern uncontrolled factors that may have affected our results. A possible threat is represented by the manually created ground truth, which was unavoidable because no automated method could provide us with the ideal classification of web pages. To minimize this threat, the authors of this work created, in isolation, a ground truth. Then, the two established a discussion to produce a single ground truth for each web app.

For reproducibility of the results, we made our tool, datasets and used subject systems available [168], along with required instructions.

2.9 Related Work

A large body of research has addressed the analysis of web sites structure via clustering for clone detection and duplicate removal of web pages [25, 27, 37, 43–45, 64, 86, 122, 162].

Henzinger [64] performed an evaluation of two near-duplicate detection algorithms based on shingling on a large dataset of 1.6B web pages. Manku et al. [86]

followed up on the work using `simhash` to detect near-duplicates for web information retrieval, data extraction, plagiarism and spam detection with promising results. Fetterly et al. [43] study the evolution of near-duplicate web pages over time and conclude that near-duplicates have little variability over time, and two pages that have been found to be near-duplicates of one another will continue to be so for the foreseeable future.

Our study is different from the above work as we aim to detect near-duplicates *within* web apps and not across different web apps. Regarding detection of within app near-duplicates, Calefato et al. [27] propose a method to identify near-duplicates as well as functional clone web pages based on a manual visual inspection of the GUI. Crescenzi et al. [37] propose a structural abstraction for web pages as well as a clustering algorithm that groups web pages based on this abstraction. Di Lucca et al. [44, 45] evaluate the Levenshtein distance and the tag frequency methods for detecting near-duplicate web pages. Eyk et al. apply simhash and broders near-duplicate detection within Crawljax [52].

In mobile testing research, researchers [8, 18] used mobile GUI widget hierarchies in order to design optimal state abstractions. Our study did not consider such techniques as they are not directly applicable for web applications.

To the best of our knowledge, our work is the first one to study different near-duplication detection algorithms (from different fields) as SAFs in a web crawler. This work is the first to propose a systematic categorization of near-duplicates in web apps, from a functional E2E testing perspective and to study the impact of near-duplicate detection on generated web application models and web testing. Moreover, our work is the first to discuss selection of thresholds for near-duplicate detection, an important first step.

2.10 Conclusions and Future Work

Automatically asserting the equality of two complex web pages is a difficult problem, which the state abstraction function of a crawler needs to solve at runtime during the exploration. The problem is further complicated by the presence of near-duplicates that need to be detected and mapped to the logical pages in order to produce meaningful crawl models.

We study ten existing near-duplicate detection techniques from three different domains and compare their effectiveness as state abstraction functions in a crawler. Our results show that near-duplicates characterized by dynamic data, as categorized in the study, are detectable when application knowledge is employed. However, near-duplicates characterized by duplication of web elements, that are often a by-product of state exploration, cannot be handled by threshold-based model inference.

Future work includes devising novel types of abstraction functions, incorporating both page structural and visual characteristics in a single hybrid solution to detect different kinds of near-duplicates.

Chapter 3

Fragment-Based Test Generation For Web Apps

3.1 introduction

Regression testing of modern web apps is a costly activity [152] in practice, which requires developers to manually create test suites, using a combination of programming and record/replay tools such as Selenium [127]. In addition, maintaining such test suites is known to be costly [58, 93] as even minor changes of the app can cause many tests to break; for example, according to a study at Accenture [58] even simple modifications to the user interface of apps result in 30–70% changes to tests, which costs $120 million per year to repair. When the test maintenance cost becomes overwhelming, whole test suites are abandoned [33].

Given the short release cycles of modern web apps and maintenance costs of manually written tests, automatic generation of regression test suites seems a viable alternative. However, the effectiveness of web test generation techniques [23, 97, 99] is limited by the ability to obtain an accurate and complete model of the app under test. Manual construction of such models for complex apps is not practical. Automated model inference techniques [96] trigger user actions such as clicking on buttons and record corresponding transitions between states in the web app to build a graph-based model.

One particular challenge here is the presence of *near-duplicate* states in web apps, which can adversely impact the inferred model in terms of redundancy and adequacy [173]. Near-duplicates are states that are *similar* to each other in terms of functionality [64].

Another important challenge is the generation of test assertions. Two factors directly contribute to the challenge here, namely effectiveness and tolerance/robustness, i.e., regression test assertions should be able to detect unexpected app behavior, but at the same time, be tolerant to minor changes that do not affect the functionality.

Both model inference and test oracle generation thus require suitable abstractions to produce effective and robust test suites. Existing techniques generate regression tests that compare the *whole* page as seen during testing with an instance of the page recorded on a previous version [97, 127]. Our insight is that, such whole-page comparison techniques, although effective at detecting changes, are not tolerant enough to handle near-duplicates and make the test suites fragile. Test fragility is known to be a huge problem in web testing [73].

In this work, we propose a novel state abstraction that employs fine-grained fragments to establish functional equivalence. We conjecture that a web page is not a singular functional entity and thus partitioning it into separate fragments can help in determining functional equivalency when comparing different states. Using this novel state abstraction, we have developed a technique, called FRAGGEN. Our fragment-based analysis enables us to (1) prioritize available actions to diversify exploration, (2) accomplish state comparison without the need for manually selecting thresholds, a manual tedious fine-tuning process, required for all existing techniques [173]; our state comparison algorithm leverages both *structural* and *visual* properties of the page fragments to identify near-duplicate characteristics specific to the web app under test during model inference, and (3) generate test assertions that operate at the fragment level instead of the whole page level, and apply fragment *memoization* to make them much more robust to state changes that should not break regression tests.

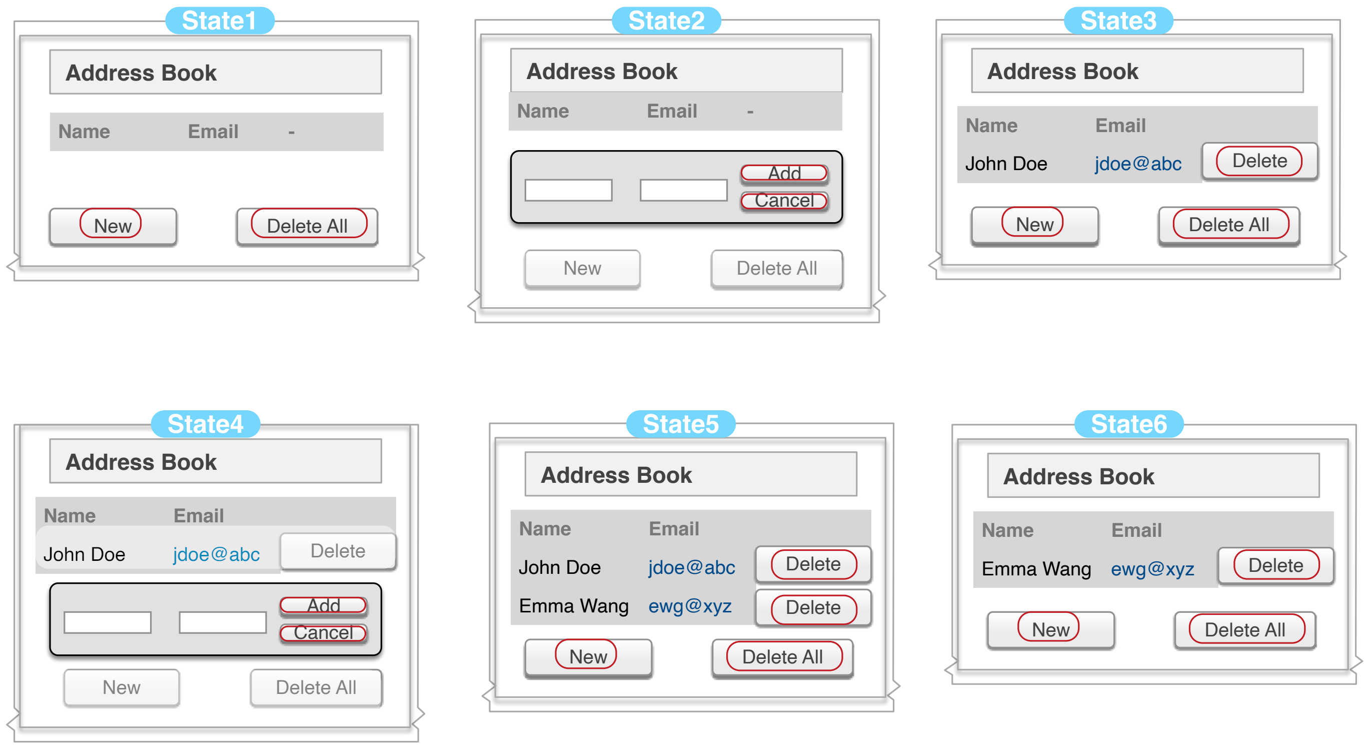


Figure 3.1: Motivating example: app states with *actionables* highlighted.

Our empirical evaluation shows that FragGen is able to outperform whole-page techniques in classifying state-pairs and identifying near-duplicates. On a dataset of 86,165 manually labelled state-pairs, FragGen detected 123% more near-duplicates on average and 82% more than the best performing existing technique. When employed to infer web app models, FragGen is able to diversify exploration using fine-grained fragment analysis to produce models with 70% higher precision and 62% higher recall on an average compared to the state-of-the-art technique. In addition, our evaluation shows that FragGen generated test suites that are better suited for regression testing. Where existing techniques generated brittle test suites with nearly 17% test actions that fail even on the same version of the web app, FragGen generated reliable test suites with nearly 100% successful test actions on the same version of the web app, and detected more app changes with fewer false positives when run on different application versions. When evaluated through mutation analysis, FragGen's test oracles could detect 98.7% of visible state changes while being tolerant enough to ignore 90.6% of equivalent mutants.

This work makes the following contributions:

- A technique for determining state equivalence (i.e., distinct, clone, near-duplicate) at the fragments-level, without a need for setting thresholds.
- A state abstraction technique that uses the structural as well as visual properties of fragments for equivalence checking.
- A novel model inference approach that employs page fragments to explore the web application state-space.
- A fine-grained fragment-based test assertion generation for effective and reliable regression testing.
- The evaluation and implementation of FragGen, which is publicly available [169].

3.2 Background and Motivation

In this section, we provide the background information on web app testing and analysis, and introduce key terms and concepts that are used in the rest of the chapter.

We use a running example, shown in Figure 3.1, based on one of our subject systems [116]. The example web app is a single page application which provides functionality to add, view, modify and delete addresses in a database through "actionable" web elements such as *buttons* and *links*, highlighted in their corresponding pages.

3.2.1 Automatic Test Generation for Web Apps

In this subsection, we describe automated model inference through state exploration and model-based test generation employed by existing techniques and their limitations.

Model inference. Automated model inference is an iterative process of exercising the functionality of a given web app by triggering events on actionable elements (α), such as *button clicks*, and capturing the resulting state transitions ($\mathcal{A}$) as a graph-based model ($\mathcal{M}$). Formally:

Definition 12 (**State Transition** ($\mathcal{A}_x$)). is a tuple ($\mathscr{S}_{src}$, α_x, $\mathcal{S}_{tgt}$) where exercising an actionable α_x in a state $\mathcal{S}_{src}$ produces a transition to state $\mathcal{S}_{tgt}$.

Definition 13 (**Application Model** ($\mathcal{M}$)). is a directed graph ($\{\mathcal{S}_1..\mathcal{S}_n\}$, $\{\mathcal{A}_1..\mathcal{A}_m\}$) with *app states* ($\mathcal{S}_a$) as nodes and *state transitions* ($\mathcal{A}_x$) as labelled directed edges between nodes.

Current model inference techniques rely on a state abstraction function (*SAF*), that determines similarity between two given states p_1 and p_2 in order to avoid redundancies in the captured model and duplication of exploration effort. Formally:

Definition 14 (**State Abstraction Function** (*SAF*)). is a pair ($dfunc$, t), where $dfunc$ is a similarity function that computes the distance between any two given web pages p_1, p_2, and t is a threshold defined over the output values of $dfunc$. *SAF* determines whether the distance between p_1 and p_2 falls below t.

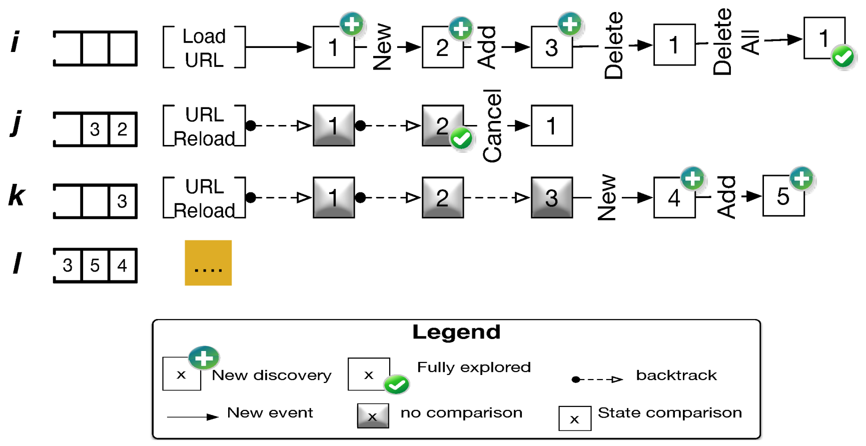


Figure 3.2: Model inference for the motivating example.

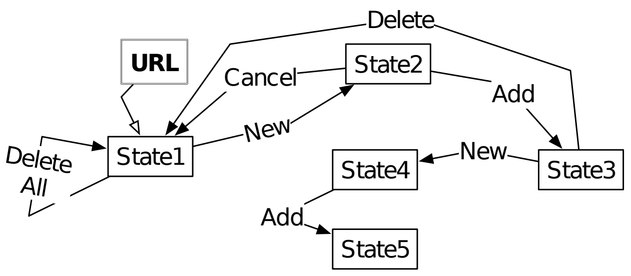


Figure 3.3: Inferred model of the motivating example.

$$SAF(dfunc, p_1, p_2, t) \begin{cases} true & : dfunc(p_1, p_2) < t \\ false & : otherwise \end{cases}$$

Figure 3.2 illustrates the steps of model inference for our motivating app, assuming a depth-first exploration strategy is followed. In the first iteration, labelled *i*, the technique loads the app in the browser using its URL, and stores the corresponding state as the root node in the model. Thereafter, each action performed on the web app can be either *exploration* step or a *back-tracking* step. In each exploration step, depicted in solid arrows, an unexplored actionable (α_{new}) from the current state is invoked and the resulting state is added to the model if it is deemed to be *different* from every existing state in the model by a SAF as defined in defintion 14. The observed state transition is then recorded as a directed edge between the source and target states in the model.

An iteration ends when the current state is fully explored and the next iteration starts by choosing an existing state in the model with unexplored actionables. In order to reach the selected state, *back-tracking* actions, indicated with broken arrows, are performed by using the transitions that are already recorded in the model. For example, iteration ”i” ends upon reaching state $\mathcal{S}_1$, which is fully explored at that point. The next iteration ”j” then starts by choosing and navigating to state $\mathcal{S}_2$, by using recorded transitions. Iteration ”l” in Figure 3.2 continues exploration by choosing one of the unexplored states $\mathcal{S}_3$, $\mathcal{S}_5$ and $\mathcal{S}_4$.

Termination and stopping criteria. Model inference techniques provide options to configure *stopping criteria* such as exploration time limit in order to end the inference process. *Termination* on the other hand happens when the technique decides that no unexplored actionables are left to exercise.

Figure 3.3 shows the model inferred for our motivating example at the end of iteration ”k”.

The model inference has not *terminated* at this point because there are unexplored actionables available.

Definition 15 (**Path ($\mathcal{P}$)**). A sequence of transitions ($\mathcal{A}_0 ... \mathcal{A}_n$) is a $\mathcal{P}$ if for 0¡=i¡n, $\mathcal{A}_i$, $\mathcal{A}_{i+1} \in \mathcal{P} \implies \mathcal{A}_i(\mathcal{S}_{tgt}) = \mathcal{A}_{i+1}(\mathcal{S}_{src})$.

Test generation. Once a model of web app is available, model-based test generation provides a set of paths $\{\mathcal{P}_1, \mathcal{P}_m\}$, with adequate coverage of states and/or transitions in the model, where, each path $\mathcal{P}$ (formally defined in 15) is a sequence of recorded transitions. Listing 4.1 shows a test case $\mathcal{T}$ generated from the inferred model of Figure 3.3. Each test case starts by loading the URL of the app and verifying the browser state to be the $\mathcal{S}^0_{src}$, i.e. the source state of the first transition of the path. Thereafter, for each transition ($\mathcal{A}_x$), a *test action* is derived from the actionable α_x and a *test oracle* is added for the target state $\mathcal{S}^x_{tgt}$. The test case in Listing 4.1 is examining the path $\mathcal{P}$: [$\mathcal{S}_1$, α_{List}, $\mathcal{S}_2$, α_{new}, $\mathcal{S}_3$, α_{add}, $\mathcal{S}_4$].

Listing 3.1: Generated test case

```
def Test1():
            driver.loadURL("Base_URL")
            assert(isEqual(driver.currentState, state1)
            driver.findElement("List").click()
            assert(isEqual(driver.currentState, state2)
            driver.findElement("New").click()
            assert(isEqual(driver.currentState, state3)
            driver.findElement("add").click()
            assert(isEqual(driver.currentState, state4)
```

3.2.2 Automatic Test Generation Challenges

The main challenge in model inference of web apps is the presence of a large number of nearly identical or near-duplicate web pages, which the existing techniques cannot identify effectively and as a result generate sub-optimal models.

While the presence of near-duplicates in the inferred web app model leads to generation of redundant test cases, it is also indicative of wasted exploration effort that could be spent discovering unseen states. Typically, exploration of similar actions in near-duplicate states does not improve functional coverage of the model but instead can lead to creation of even more near-duplicates, as can be seen even in our simple example app. In our example, the states $\mathcal{S}_3$, $\mathcal{S}_5$ and $\mathcal{S}_6$ shown in Figure 3.1 are all considered *functional near-duplicates* as they all offer similar web app functionality, and model inference may never *terminate* if similar actions α_{New} and α_{Add} are explored in each near-duplicate state.

In our previous work, we proposed [173] to categorize a given pair of web pages as either clone (*Cl*), near-duplicate (*Nd*) or distinct (*Di*) by labelling the observed changes between them. We formally defined *Near-duplicates* in web testing as:

Definition 16 (**Functional Near-Duplicate (Nd)**). A given state-pair (p_1, p_2), is considered to be a functional near-duplicate if the *changes* between the states do not alter the overall functionality of either state.

Near-duplicates are further divided in three categories based on the nature of changes between the two pages:

- Cosmetic (Nd_1-*data*): Changes such as different advertisements, that are

Table 3.1: Raw distances of state-pairs

state-pair	RTED [112]	Levenshtein [75]	TLSH [108]	SimHash [29]	HYST [150]	BlockHash [175]	PHASH [179]	PDIFF [178]	SIFT [80]	SSIM [10]	Human
	DOM				VISUAL						
$(\mathcal{S}_3, \mathcal{S}_6)$	0.0	0.002	1	0	91	0	0	0.0006	5	0.003	*Nd*$_2$
$(\mathcal{S}_1, \mathcal{S}_3)$	0.13	0.08	36	0	9751	4	2	0.008	14	0.04	*Di*
$(\mathcal{S}_3, \mathcal{S}_5)$	0.16	0.12	106	0	27683	5	2	0.045	14	0.05	*Nd*$_3$

irrelevant to functionality of web app.

- Dynamic Data (Nd$_2$-*data*): Changes are limited to data while the page structure remains the same. Example *($\mathcal{S}_3$, $\mathcal{S}_6$)*.
- Duplication (Nd$_3$-*struct*): Addition or removal of web elements equivalent to existing web elements. Example *($\mathcal{S}_3$, $\mathcal{S}_5$)*.

Researchers have employed various similarity functions and abstractions for web pages based on their DOM tree-structures [44, 95], and visual screenshots [32, 84, 85, 129] to perform state comparison. However, a study [173] on near-duplicates in web testing shows that the whole-page based SAFs currently being used by existing model inference techniques cannot reliably identify near-duplicates and as a result infer imprecise and incomplete models.

To illustrate the limitations of the SAFs currently being used, we analyzed three state-pairs of our example app in Table 3.1. For each of the three state-pairs, the table shows the distance between states computed by each of the ten state abstraction techniques and the *human* classification process followed in the study [173].

Consider the states $\mathcal{S}_1$, $\mathcal{S}_3$, $\mathcal{S}_5$, and $\mathcal{S}_6$ from Figure 3.1, all of which display stored addresses. When examined manually, $\mathcal{S}_1$ is considered distinct (*Di*) from $\mathcal{S}_3$, $\mathcal{S}_5$ and $\mathcal{S}_6$ as it does not contain table row functionality to select an address entry.

However, $\mathcal{S}_3$ and $\mathcal{S}_6$ differ only by the *data* in a table row, which does not alter the functionality and hence they are considered as functional near-duplicates. Further, even-though $\mathcal{S}_5$ contains extra table rows, they only duplicate the functionality of a single row in $\mathcal{S}_3$. Therefore, a human tester would label state-pairs such as ($\mathcal{S}_3$, $\mathcal{S}_5$) to be functional near-duplicates.

The 10 comparison techniques, as shown in the Table 3.1, consider the state-pair ($\mathcal{S}_3$, $\mathcal{S}_5$) to be the farthest apart of the four state-pairs even though they are functionally equivalent. If the distance thresholds were set to be higher, states such as $\mathcal{S}_3$ might be discarded from the model as they will be considered equivalent to $\mathcal{S}_1$, making the model *incomplete*. On the other hand, lower thresholds would make the model imprecise with the presence of states such as the state $\mathcal{S}_5$. This shows that threshold-based whole-page comparison cannot produce optimal web app models.

Test breakages. In addition to model inference, similarity between two given web page states is crucial in generating effective test oracles as well. If the state comparison techniques are sensitive to minor changes unrelated to functionality in modern apps, test oracles can break and result in a high number of false positive test failures. False positive test failures necessitate costly manual analysis and impact the effectiveness of automated regression testing techniques.

We identified duplicated functionality within a web page to be the root cause of Nd3-*struct* near-duplicates that make existing model inference and test generation techniques impractical for modern web apps. This observation that "whole-page techniques" cannot detect Nd3-*struct* near-duplicates motivated us to investigate the idea of decomposing a given web page into smaller fragments for a more fine-grained analysis.

3.2.3 Page Fragmentation

Page fragmentation, also known as page segmentation, is the decomposition of a given web page into smaller fragments or segments. In existing research, the most popular downstream tasks such as content extraction that apply page fragmentation relate to human consumption of web pages and focus on extracting textual semantics.

The "VIsion-based Page Segmentation algorithm" (VIPS) [26], proposed in

2003, is the de-facto standard for web page segmentation. A recent large scale empirical study [69] compares five page segmentation techniques using a dataset of 8,490 web pages and concludes that VIPS is still the overall best option for page fragmentation. VIPS employs a top-down approach, where, for each HTML node in the DOM [1], a DoC (Degree of Coherence) is assigned to indicate coherence of the content in the block based on visual perception. Then it tries to find the separators between these extracted blocks. Here, separators denote the horizontal or vertical lines in a webpage that visually cross with no blocks. Finally, based on these separators, it extracts a hierarchy of fragments. Each extracted fragment visually conforms to a rectangle in the page between two horizontal and vertical separators. All the DOM nodes that are part of the rectangle are then considered to be part of the fragment. We refer the interested reader to the original VIPS paper for more details of the algorithm [26].

In the next section, we explain how we leverage page fragmentation to overcome existing challenges in model inference and regression test generation for modern web apps.

3.3 Approach

At a high level, our approach, called FRAGGEN, relies on the insight that a web page is not a singular functional entity, but a set of functionalities, where each functionality may be available in more than one page. Based on this insight, we propose a novel state abstraction that defines a page as a *hierarchy of fragments*, where each fragment represents a semantic sub functionality. Our model inference technique then employs this state abstraction to detect near-duplicates and optimize the state space exploration by prioritizing actions that belong to unique page fragments in each page. From the inferred model, we subsequently generate test cases with robust assertions that rely on our fragment-based abstraction and are capable of reporting warnings in addition to errors by utilizing knowledge gained during model inference. Next, we describe our state abstraction, followed by our model inference, and test generation techniques.

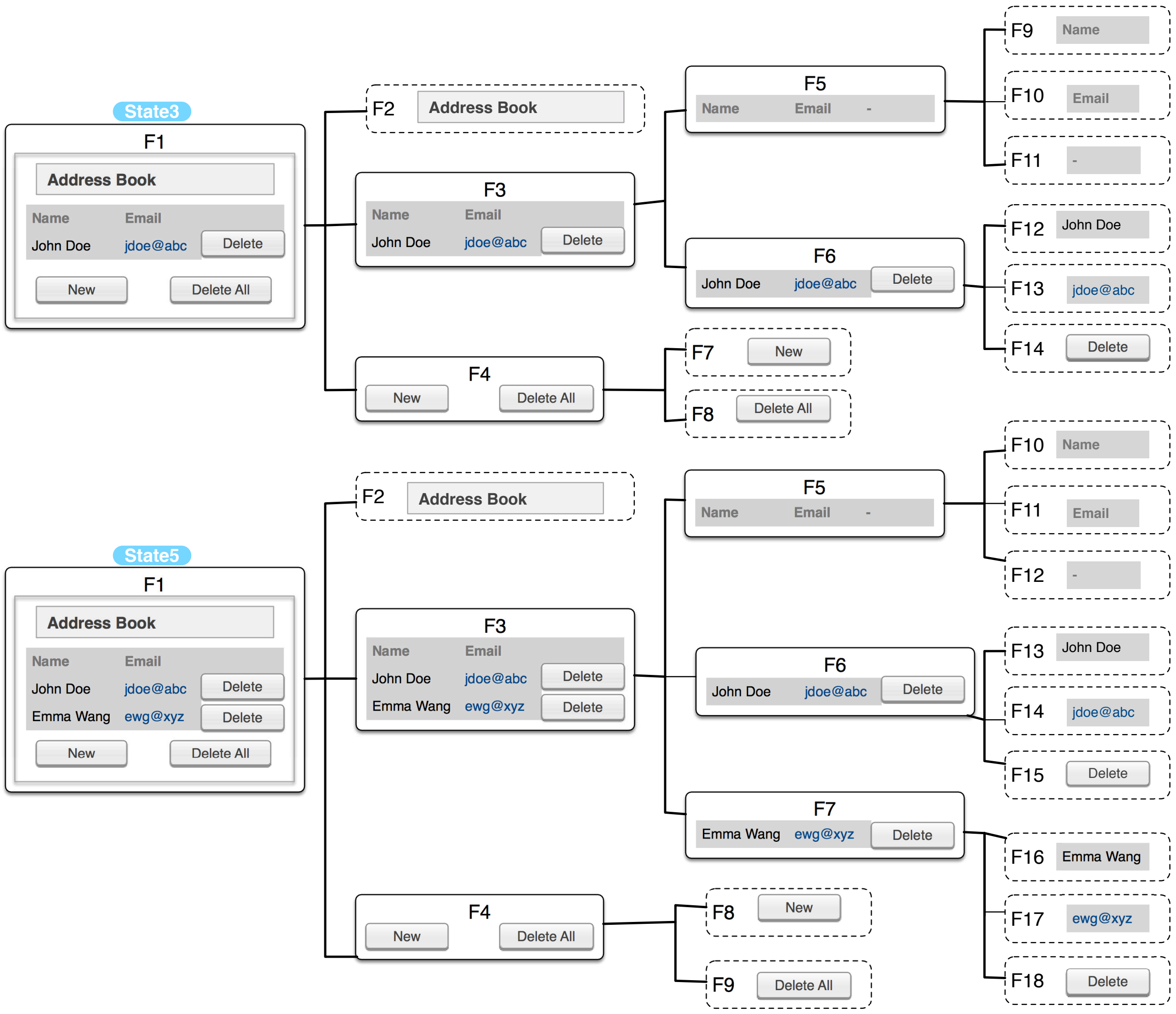


Figure 3.4: Fragment-based state comparison in FRAGGEN

3.3.1 Fragment-based State Abstraction

We consider a web page or a state as a hierarchy of fragments, where, each fragment is a portion of the state and represents functionalities offered by its child fragments.

Definition 17 (**Application State** ($\mathscr{S}$)). is a tuple ($\mathscr{D}$, $\mathscr{V}$, $\mathcal{F}_{root}$) where $\mathscr{D}$ is the dynamic DOM [1] of the page, $\mathscr{V}$ is the screenshot of the page and $\mathcal{F}_{root}$ is the root fragment.

The root fragment, $\mathcal{F}_{root}$, of a state $\mathscr{S}$ is the full page, and has no parent. It has all the nodes in the DOM tree $\mathscr{D}$ of $\mathscr{S}$. Therefore, comparing two states is the same as comparing their root fragments.

Definition 18 (**Fragment** ($\mathscr{F}$)). is a tuple ($\mathscr{N}$, V, $\{\mathcal{F}_{c0}, \mathcal{F}_{c_1} \dots\}$) where $\mathscr{N}$ is the set of DOM nodes of $\mathscr{F}$, V is the screenshot of $\mathscr{F}$ and each $\mathcal{F}_{c_i}$ is a child fragment.

Comparing fragments. Using this fragment-based representation, we classify a given state-pair by comparing the fragments they are composed of. Figure 3.4 shows the fragment hierarchies for states $\mathcal{S}_3$ and $\mathcal{S}_5$ from our motivating example in Figure 3.1. Our fragment-based classification, which takes both structural and visual aspects into account is shown in Algorithm 1. The structural aspect (line 2) uses the nodes on the DOM subtree of the fragment after pruning textual content and attributes. The visual aspect (line 4) uses a localized screenshot of the fragment.

As algorithm 1 shows, we combine structural and visual analysis to identify near-duplicates. Using visual similarity instead of element attributes and textual data from DOM allows us to disregard changes in the DOM that have no visual impact. In addition, we found that visual comparison is effective in identifying changes in dynamic web elements such as carousels that use only JavaScript and CSS.

FRAGGEN classifies two fragments to be *clones* if their structural and visual properties are exactly the same. We employ APTED [113] and Color Histogram [150] to compare the structural and visual aspects of the fragments, respectively. These techniques have been employed individually as state abstractions for

Algorithm 1: Fragment-based classification

```
Function Classify($\mathcal{F}_1$, $\mathcal{F}_2$):
    // clone(Cl), distinct(Di), Nd2-data, Nd3-struct
    Output: class
    $\mathcal{N}_{diff} \longleftarrow$ treediff ( $\mathcal{F}_1.\mathcal{N}$, $\mathcal{F}_2.\mathcal{N}$ )
    if $\mathcal{N}_{diff} = \phi$ then                       /* Matching DOM */
        $\mathcal{V}_{diff} \longleftarrow$ imagediff ( $\mathcal{F}_1.V$, $\mathcal{F}_2.V$ )
        if $\mathcal{V}_{diff} = \phi$ then             /* Matching Screenshots */
            return Cl                               /* class: clone */
        else
            return Nd2-data                         /* class: Nd2-data */
        end
    else                                    /* Check Child Fragments? */
        return MapChildFragments($\mathcal{N}_{diff}$, $\mathcal{F}_1$, $\mathcal{F}_2$)
    end
End Function

Function MapChildFragments($\mathcal{N}_{diff}$, $\mathcal{F}_1$, $\mathcal{F}_2$):
    // distinct(Di), Nd3-struct
    Output: class
    /* Mapping child fragments for every changed node */
    foreach $n \in \mathcal{N}_{diff}$ do
        found $\longleftarrow$ false
        $\mathcal{F}_{oth} \longleftarrow$ ( $n \in \mathcal{F}_1.\mathcal{N}$ ) ? $\mathcal{F}_2$ : $\mathcal{F}_1$
        $\mathcal{F}_{clo} \longleftarrow$ closest ( n )
        foreach $\mathcal{F}_{chi} \in \mathcal{F}_{oth}$.childen do
            if Classify ( $\mathcal{F}_{clo}$, $\mathcal{F}_{chi}$ ) != Di then
                found $\leftarrow$ true                     /* Found mapping */
            end
        end
        if ! found then                             /* No mapping found */
            return Di                               /* class: Distinct */
        end
    end
    return Nd3-struct                               /* class: Nd3-struct */
End Function
```

whole web pages in the literature [129, 173], but have not been combined to determine state equivalence. When the fragments differ visually, but are found to be structurally equivalent, FRAGGEN considers the fragments to be near-duplicates of the type Nd$_2$-*data* (see Definition 16).

In case they differ structurally, as root fragments of $\mathcal{S}_3$ and $\mathcal{S}_5$ do for example shown in Figure 3.4, the classification is performed by a mapping function (lines

14-28) that extracts all changed DOM nodes between the two fragments and maps the corresponding fragments in the hierarchy.

Given a pair of fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ and a list of changed nodes between the two fragments $\mathcal{N}_{diff}$, the function *MapChildFragments* gets the closest fragment $\mathcal{F}_{clo}$ for each changed DOM node (n). The closest fragment for a DOM node is the smallest child fragment in the fragment hierarchy containing n. Thereafter, in lines 19-23, it attempts to find an equivalent fragment for the closest fragment in the fragment hierarchy of the other fragment. For example, if n_1 is a changed node that belongs to $\mathcal{F}_1$, with $\mathcal{F}_{clo}$ as its closest fragment, then $\mathcal{F}_{oth}$ is $\mathcal{F}_2$ and we attempt to find if any of the child fragments of $\mathcal{F}_2$ are equivalent to $\mathcal{F}_{clo}$. If the closest fragment for any of the changed DOM nodes cannot be mapped to a child fragment of the other fragment (lines 24-26), we declare the two given fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ to be distinct (*Di*). Otherwise, the two fragments are considered to be near-duplicates of type Nd3-*struct*.

In the example shown in Figure 3.4, when classifying the root fragments for $\mathcal{S}_3$ and $\mathcal{S}_5$, *MapChildFragments* would be called to classify $\mathcal{F}_3$ of states $\mathcal{S}_5$ and $\mathcal{S}_3$. All the changed DOM nodes ($\mathcal{N}_{diff}$) between the two fragments belong to $\mathcal{F}_7$ in $\mathcal{S}_5$, As the function iterates (line 19) through child fragments to find a mapping, eventually $\mathcal{F}_7$ of $\mathcal{S}_5$ will be found to be equivalent to $\mathcal{F}_6$ of $\mathcal{S}_3$. As a result, the fragment pair would be classified as Nd3-*struct*. As the rest of the fragments are equivalent, the overall state-pair ($\mathcal{S}_3$, $\mathcal{S}_5$) would be classified as Nd3-*struct* near-duplicates as well.

When the fragments containing changed DOM nodes cannot be mapped, FRAGGEN considers the two states to be *distinct*. One such example is the state-pair ($\mathcal{S}_1$, $\mathcal{S}_3$), which is classified to be *distinct* because the table row in $\mathcal{S}_3$ contains no equivalent fragment in $\mathcal{S}_1$. On the other hand, ($\mathcal{S}_3$, $\mathcal{S}_6$) are classified as Nd2-*data* because the abstracted DOM hierarchy is equal, making them structurally similar, while the data changes inside the table make them visually dissimilar.

3.3.2 Fragment-based Model Inference

FRAGGEN infers the model of a given web app by iteratively triggering user interactions on actionables and recording the corresponding state transitions much like the existing techniques (described in Section 3.2.1).

After an action (α) is performed on a source state, the resulting browser state is compared to all the existing states in the model using the $Classify$ function algorithm 1. If the classification is clone or Nd$_2$-*data* for any of the existing states, the new state is discarded, otherwise, it is added to the model.

However, existing techniques assign equal importance to every actionable of a newly discovered state, often wasting exploration effort exercising similar actionables. In contrast, FRAGGEN identifies similar actionables through fragment analysis to diversify the exploration. In this section, we describe how our fragment analysis is used to 1) diversify state exploration to discover unique states faster, and 2) identify *data-fluid* fragments to generate effective test oracles.

Exploration strategy

FRAGGEN ranks the actionables and states using Equation 3.2 and Equation 3.3, respectively, using the fragment comparisons in Equation 3.1 to determine the equivalence of actionables. Actionables are special DOM nodes such as buttons which are used for user interaction.

Where a function $Xpath(\alpha, \mathcal{F})$ provides a relative XPath expression [164] for α within $\mathcal{F}$, we decide the equivalence of two actionables $\alpha_x \in \mathcal{F}_x$, $\alpha_y \in \mathcal{F}_y$ using:

$$\alpha_x \equiv \alpha_y \iff (\text{Classify}(\mathcal{F}_x, \mathcal{F}_y) = \text{Clone} \vee \text{Nd}_2\text{-}data) \wedge (Xpath(\alpha_x, \mathcal{F}_x) == Xpath(\alpha_y, \mathcal{F}_y)) \quad (3.1)$$

Given a constant c_0 where $0 < c_0 <= 1$, and α_{eq} is an equivalent actionable determined using Equation 3.1, the score for an actionable α is computed as:

$$score(\alpha) = \begin{cases} -1, & \text{if } \alpha\text{.explored.} \\ 0, & \text{if } \exists\ \alpha_{eq} \mid \alpha_{eq}\text{.explored.} \\ c_0 * size(\{\alpha_{eq} \ldots\}), & \text{otherwise.} \end{cases} \quad (3.2)$$

The score for a state $\mathcal{S}$ is computed using the scores of actionables in the state as:

$$score(\mathcal{S}) = \sum_{n=1}^{\text{size}(\{\alpha\})} \text{score}(\alpha) \quad (3.3)$$

Using the Equation 3.3, FRAGGEN chooses the next state to explore based on the total score of the actionables in each state. The score for each actionable (α) is assigned by Equation 3.2 based on its equivalent actionables (α_{eq}) across states. Once a state is chosen, again the actionable with a higher score is chosen for exploration, which helps prioritize unexplored actionables that have high repetition such as navigation links. Equivalence of actionables is established using the fragments that contain them using the Equation 3.1. Through these equations, every time an actionable is explored, FRAGGEN de-prioritizes all of its equivalent actionables to diversify the exploration. As a result of Equation 3.1, this de-prioritization often reduces redundant exploration effort spent in already seen functionality.

Figure 3.5 shows the model inference performed by FRAGGEN for our running example. As it can be seen, FRAGGEN can generate a more precise model, exercising actionables that are considered unique by our fine-grained fragment analysis. For example, FRAGGEN can identify α_{New} in $\mathcal{S}_3$ to be equivalent to that of $\mathcal{S}_1$ and doesn't consider it to be unexplored, thus *terminating* the exploration. On the other hand, as shown in Figure 3.3, existing techniques exercise α_{New} on $\mathcal{S}_3$ as well, leading to the creation of $\mathcal{S}_4$ and $\mathcal{S}_5$ and necessitating further exploration of the same actionables in the newly added states as well. In fact, without identifying the equivalence of such repeated actionables, state exploration will never *terminate* even for this simple example app we present in the chapter. Existing techniques would require a stopping criterion such as a time limit to end the inference process and will likely generate an imprecise app model with a high number of near-duplicates.

Configuration options. *constant c_0 in Equation 3.2*: c_0 determines the weight given to the presence of duplicates for an action that is yet to be explored. In our experiments, we used the default value which is set to 1. We provide the ability to tune it if necessary for specific web applications. The value of constant c0 in Equation 3.2 can impact prioritization of states, where, a high value could delay exploration of a potentially more interesting newly discovered state. This scenario is possible if the number of duplicates for a single unexplored actionable outnumber the total actionables in a newly discovered state. In this scenario, the older states containing this particular unexplored action would get a higher priority even if a majority of the actions in the state itself are already explored. However, once any one of the instances of the particular actionable is exercised, the value of

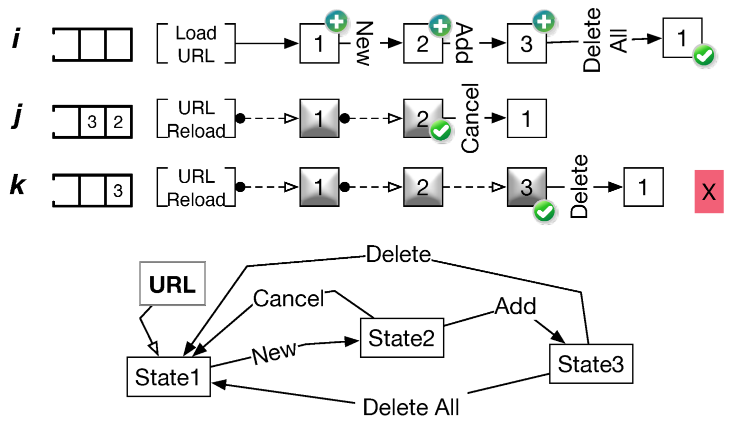


Figure 3.5: Model inferred by FRAGGEN

constant c_0 becomes irrelevant.

Termination: We designed FRAGGEN to be configured to limit the amount of duplication in order to avoid generating inflated models. By default, FRAGGEN does not exercise any actionable which is found to be similar to an already explored actionable. It will *terminate* once all unique actionables are exercised. However, this duplication in exploration effort can sometimes be necessary to fully explore app functionality. Therefore, we provide configuration to continue exploration using unexplored actionables that were previously skipped because they were similar to already explored actionables. FRAGGEN would continue to use Equation 3.2 and Equation 3.3, for prioritization.

Memoization and data-fluid fragments

During the model inference process, recall that after every exploration step, FRAGGEN decides if the resulting browser state should be retained in the model by comparing it to the existing states using the $Classify$ function. Our exploration strategy also relies heavily on the $Classify$ function to search for diverse actions as seen in Equation 3.1.

From algorithm 1, it can be seen that $Classify$ for a given pair of fragments is recursive in nature that may require comparing the corresponding child fragments. To improve its performance, we apply a classic technique for recursive algorithms called memoization, which aims to avoid recalculation of results for sub-problems.

Here, instead of storing just the comparison results for every pair of fragments, we employ a map-based implementation with unique fragments as map keys and a list of their duplicates as the values.

Whenever a new state is added to the model, we update the map by comparing each of the fragments in the new state to existing unique fragments. If any of the fragments in the new state is not a clone (*Cl*) of existing unique fragments, it is added to the map as a new unique fragment. Otherwise, it is added as a duplicate of the unique fragment it is found to be a clone of.

Further, we utilize this memoization technique using the map of fragments to identify *data-fluid* fragments. A fragment is data-fluid if it is found to be an Nd_2-*data* near-duplicate of at-least one other fragment in the map. Intuitively, we are trying to recognize the parts of the dynamic DOM, specific to the web app being tested, that are likely to have data changes. Any clone of a data-fluid fragment is also data-fluid. For example, in Figure 3.4, $\mathcal{F}_6$ in $\mathcal{S}_3$ and $\mathcal{F}_6$, $\mathcal{F}_7$ in $\mathcal{S}_5$ are identified to be *data-fluid*.

In practice, there could be multiple causes for changing data in fragments; such as a DOM element that displays current time. Indeed, this ability to mutate dynamic DOM in real-time is the primary reason that developers are able to create highly interactive and responsive web apps. However, such changes also pose a challenge to web testing in the form of fragility of the automatically generated test oracles which fail in response to every change in the dynamic DOM. In the next section, we describe how FRAGGEN mitigates this challenge by making use of memoization and data-fluid fragments to generate robust test oracles.

3.3.3 Test Generation

UI Test cases, such as the one shown in Listing 4.1, are sequences of UI events derived from inferred web app models as discussed in Section 3.2.1. We designed FRAGGEN to generate tests using *exploration paths* which are essentially inferred model iterations as shown in Figure 3.5, and contain at least one unexplored action in each new path.

Recall (from Section 3.2.1) that a model iteration starts by reloading the URL and ends when the browser state does not have any unexplored actions remaining.

As the exploration paths or model iterations cover all states and transitions, test cases generated by FRAGGEN also cover all the states and transitions in the model.

Each exploration path translates to a test case that starts with loading URL, similar to the test case shown in Listing 4.1. For each transition ($\mathcal{A}$ -¿ ($\mathcal{S}_{src}$, α, $\mathcal{S}_{tgt}$)) in an exploration path ($\mathcal{P}$, definition 15), we generate a test step to perform the action (α) and generate assertions that compare corresponding recorded model states with the browser states by using the function $Classify$ (algorithm 1).

In the remainder of this section, we describe the challenges in generating robust test assertions and how FRAGGEN utilizes fragment analysis to tackle this challenge.

Test assertions

An important but often neglected aspect of regression testing is the creation of effective test assertions or test oracles, which are key in detecting app regressions. Existing test generation techniques generate fragile/brittle test assertions [157], which results in many false positive test failures.

Generating web test assertions is considered difficult [20], primarily because they require a reliable state comparison to handle near-duplicates. Previously, researchers have relied on manual specification of DOM invariants [97] specific to a web app. Although effective, such a procedure is not scalable for complex large applications and requires significant human effort as well as domain knowledge.

Our test assertions use the fragment-based state abstraction to identify changes between recorded states in the model and the states observed during test execution. We designed FRAGGEN's test execution to utilize data-fluid fragments identified during model inference to assign importance to the changes detected by our state comparison. As a result, FRAGGEN can produce fine-grained warnings with different levels of severity based on the characteristics of detected changes. Test failures can then be declared based on the severity of warnings, reducing false positive test failures or test breakages caused by unimportant changes.

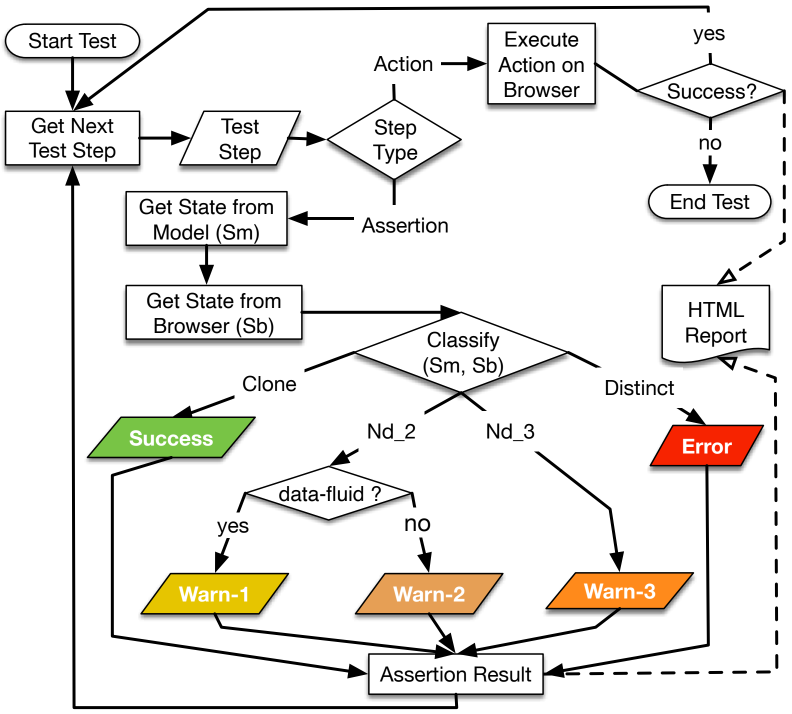


Figure 3.6: Test execution flowchart. The function $Classify$ is defined in algorithm 1

Test execution

Figure 3.6 shows the flowchart of our test execution. For each action step, that exercises actionable (α) in the transition ($\mathcal{A}$), the test execution continues if successful or stops otherwise. For each assertion step, we invoke our fragment-based comparison ($Classify$ in algorithm 1) between the browser state ($\mathcal{S}_b$) and the model state ($\mathcal{S}_m$) specified for the particular test step. Based on the output of $Classify$ and using *memoization*, FRAGGEN can create three levels of warnings in addition to success and error for each assertion. Warn-3 has the highest severity while Warn-1 is the lowest. To the best of our knowledge no prior technique exists to generate such warnings for UI test cases. Existing techniques are only capable of creating a binary decision for a given test oracle and generate either an error or a success status.

Using data-fluid fragments. We lower the severity of a warning from Warn-2 to Warn-1 when we detect that the changes observed belong to a fragment that is data-fluid. For example, a web element displaying current time on a web page changes in every concrete instance of the web page and should be given a lower severity warning compared to a change to the title of a web page, which has never been found to change during model inference. It is important to recognize that both these changes would just be textual changes in the DOM for any state comparison technique.

Configuration. In our experiments, we consider any warning other than Warn-1 to be an assertion failure. However, we provide the capability to configure the assertion failures based on warning levels. Our empirical evaluation assesses how the usage of memoization and identification of data-fluid fragments can help make our assertions robust to near-duplicates while retaining the ability to detect bugs.

Report. FRAGGEN also generates an HTML test report that shows the test execution results for easier manual analysis. The report provides a visualization of changes and their severity. Our replication package contains a test report for every test execution in the evaluation.

3.3.4 Implementation

For page fragmentation, existing implementations of VIPS rely on web rendering frameworks that are no longer maintained and render modern pages incorrectly. Therefore, we ported one implementation [118] to WebDriver API in Java, which is also used by CRAWLJAX so that pages rendered on modern browsers can be fragmented.

We use our ported version of the VIPS [26] algorithm for decomposing a web page into fragments, and compare the structural and visual aspects of the fragments using APTED [113] and Histogram [150] respectively.

We modified the latest version of CRAWLJAX [96] to employ our state abstraction for state equivalence and use our exploration strategy. Finally, our test generator is implemented as a CRAWLJAX plugin and generates JUnit test cases using the WebDriver API. Our tool, FRAGGEN, is publicly available [169].

3.4 Evaluation

Our empirical evaluation aims to assess FRAGGEN through (1) its ability to detect near-duplicates, (2) the adequacy of its inferred web app models, and (3) the suitability of its generated tests for regression testing. We do so by answering the following research questions.

RQ$_1$: *How effective is* FRAGGEN *in distinguishing near-duplicates from distinct states compared to current whole-page techniques?*

RQ$_2$: *How do the models generated by* FRAGGEN *compare to the models generated by current techniques?*

RQ$_3$: *Are the tests generated by* FRAGGEN *suitable for regression testing?*

- **RQ$_{3a}$:** *How do the generated test cases perform in regression testing scenarios?*
- **RQ$_{3b}$:** *How effective and tolerant are the generated test oracles?*

In RQ$_1$, we assess the effectiveness of competing techniques in detecting near-duplicates through state-pair classification. In RQ$_2$, we measure the model quality which directly influences the completeness and redundancies in the generated tests. In RQ$_3$, we assess the suitability of generated test suites in regression test scenarios by measuring their reliability in addition to the effectiveness in detecting app changes. For RQ$_{3a}$, we execute the generated tests on the same web app version but different platform/browser versions to analyze their usability in regression test scenarios, and then evaluate their effectiveness in detecting app changes for a different version of the web app. Finally, for RQ$_{3b}$, we perform mutation analysis of recorded web states to assess the effectiveness and robustness of generated test oracles.

3.4.1 Subject Systems

To address our research questions, we needed to manually analyze captured application states which requires a certain level of control over app behaviour and ability to replicate app states under similar experimental conditions.

Table 3.2: Experimental subjects

App	v_0	v_1	Framework	LOC
addressbook [116]	*8.2.5*	*8.2.5.1*	*PHP, JavaScript*	32K
petclinic [13]	*6010d5*	*4aa89ae*	*Java, Spring MVC*	6K
ppma [14]	*0.6.0*	*0.5.2*	*Yii, JavaScript*	556K
dimeshift [46]	*261166d*	*44089e*	*Backbone.js, JQuery*	10K
claroline [34]	*1.11.10*	*1.11.9*	*PHP, JavaScript*	340K
phoenix-trello [115]	*60c874d*	*c1cdf30*	*Phoenix, Elixir, ReactJS*	5K
pagekit [110]	*1.0.16*	*1.0.14*	*Symfony, Vue.js*	275K
mantisbt [15]	*1.1.8*	*1.2.1*	*PHP, JavaScript*	120K

In addition, for RQ_1 and RQ_2, we require a manually labelled ground-truth for state-pair classification and unique states in a web app respectively.

To this end, we selected eight open-source web apps with an available ground-truth [147] and used in prior web testing research [22, 23, 144, 145, 173].

Our eight subjects shown in Table 5.1 cover a diverse set of several popular back-end and front-end web app frameworks such as *Symfony, Yii, Spring MVC, Backbone.js, Vue.js, Phoenix/React, JQuery, Bootstrap* and *AngularJS*. Some of our subjects such as *MantisBT, Claroline, PageKit* are quite complex and immensely popular with a sizeable active user base. For example, *Claroline* is an award winning learning management system (LMS) used in more than 100 countries and is available in 35 languages; *MantisBT* is one of the most popular open source issue tracking systems in active use.

3.4.2 Competing techniques

Based on the results of our recent empirical study [173], we choose two of the best whole-page techniques, (1) RTED [112], a DOM tree differencing technique which inferred the best models on an average, and, (2) Histogram [150], which was the best performing visual algorithm outperforming RTED on several subjects. From here on, we refer to RTED as *Structural* and Histogram as *Visual* when presenting our evaluation.

Table 3.3: Manually classified state-pair dataset

Subject	Pairs	Clones	Near-Duplicates			Distinct
			Nd_2	*Nd_3*	*Total*	
addressbook	8515	26	52	2295	2347	6142
petclinic	11175	2	1433	180	1613	9411
claroline	17766	2707	69	2	71	14988
dimeshift	11628	375	570	0	570	10683
pagekit	9730	0	904	3044	3948	5782
phoenix	11175	1	25	4580	4605	6569
ppma	4851	64	467	0	467	4320
mantisbt	11325	2	1117	0	1117	10206
Total	86165	3177	4637	10101	14738	68101

3.5 State-pair Classification (RQ_1)

To address RQ_1, we compare FRAGGEN to whole-page techniques (Section 3.4.2) in terms of their ability to classify a given state-pair as either clone, near-duplicate, or distinct.

3.5.1 Procedure and Metrics

Dataset. We use an existing dataset (Table 3.3) of 86,165 state-pairs that are manually labeled as either clone (*Cl*), Nd_2-*data*, Nd_3-*struct*, or distinct (*Di*) as ground truth. It contains 4,637 Nd_2-*data* and 10,101 Nd_3-*struct* near-duplicates along with 3,177 clones and 68,101 distinct state-pairs belonging to eight subjects used in this work. For the two competing whole-page techniques, *structural* and *visual*, the dataset also includes the computed distance between the two states for each of the 86,165 state-pairs.

Comparison metrics. We use two metrics to compare the competing techniques, 1) the classification F_1 score and 2) the number of detected near-duplicates. We designed our experiment so that, given a state-pair, each technique classifies it to be either clone (*Cl*), near-duplicate (*Nd*) or distinct (*Di*). Using the manually classified ground-truth, we then compute the multi-class classification F_1 score which is the average of F_1 over all three classes (Cl, Nd, Di).

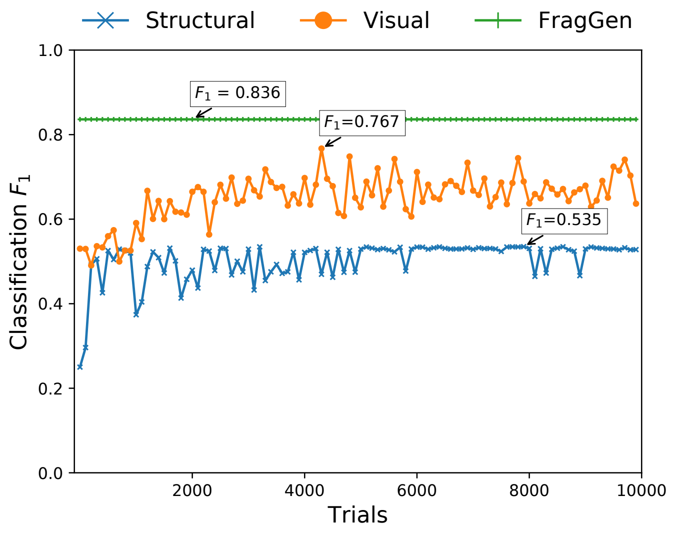
Whole-page technique configuration. The whole-page techniques by default only output a distance between the two states of a given state-pair. In order to compute the output *class* for such techniques, we follow previous work [173] which defined a function Γ as :

$$\Gamma(S_1, S_2, \mathscr{W}, t_c, t_n) \begin{cases} Cl \ : \mathscr{W}(S_1, S_2) < t_c \\ D \ \ : \mathscr{W}(S_1, S_2) > t_n \\ Nd : otherwise \end{cases}$$

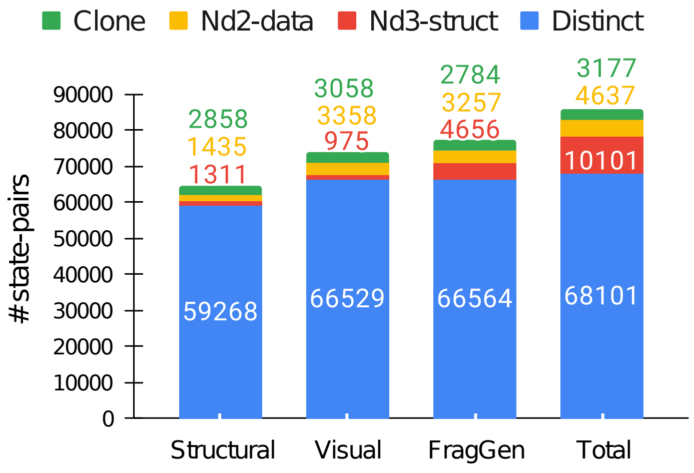
Γ takes as inputs a whole-page comparison technique $\mathscr{W}$ which computes the distance between two given states in a state-pair (S_1, S_2) and outputs a *class* using a pair of thresholds t_c, t_n. Γ classifies a state-pair to be a clone if the computed distance falls below the threshold t_c, distinct if it is above t_n or near-duplicate otherwise. Therefore, the choice of thresholds t_c, t_n, can play a huge role in the overall effectiveness of the techniques.

Threshold determination for whole-page techniques. To obtain optimal thresholds that maximize the classification scores of the two whole-page techniques, using the labelled dataset (shown in Table 3.3) as ground-truth, we employ bayesian optimization [136] to search for thresholds that maximize the multi-class classification F_1 score for Γ. We ran the optimization technique for 10,000 iterations or trials and retrieved the thresholds that provided best F_1 score to be optimal thresholds. In each trial, the optimizer chooses a pair of thresholds from the sample space of distances possible for the corresponding whole-page technique and computes the F_1 score.

FRAGGEN configuration. For each state-pair in the dataset, we invoke the $Classify$ function (algorithm 1) which outputs one of the four classes (*Di*, *Cl*, Nd$_2$-*data*, Nd$_3$-*struct*). For this experiment, we combine the two outputs Nd$_2$-*data* and Nd$_3$-*struct* into a single class *Nd* to compare with the output of Γ.



(a) Finding optimal thresholds using classification F_1



(b) Correctly classified state-pairs for optimal thresholds

Figure 3.7: State-pair classification results on the dataset

Table 3.4: F_1 of inferred models for 60 minute crawls

	addressbook	*claroline*	*dimeshift*	*mantisbt*	*pagekit*	*petclinic*	*phoenix*	*ppma*	*Average*
Structural	0.44	0.83	0.26	0.42	0.40	0.79	0.38	0.23	0.47
Visual	0.42	0.71	0.50	0.28	0.47	0.93	0.19	0.21	0.46
FRAGGEN	0.89	0.82	0.74	0.90	0.74	1.00	0.59	0.71	0.80

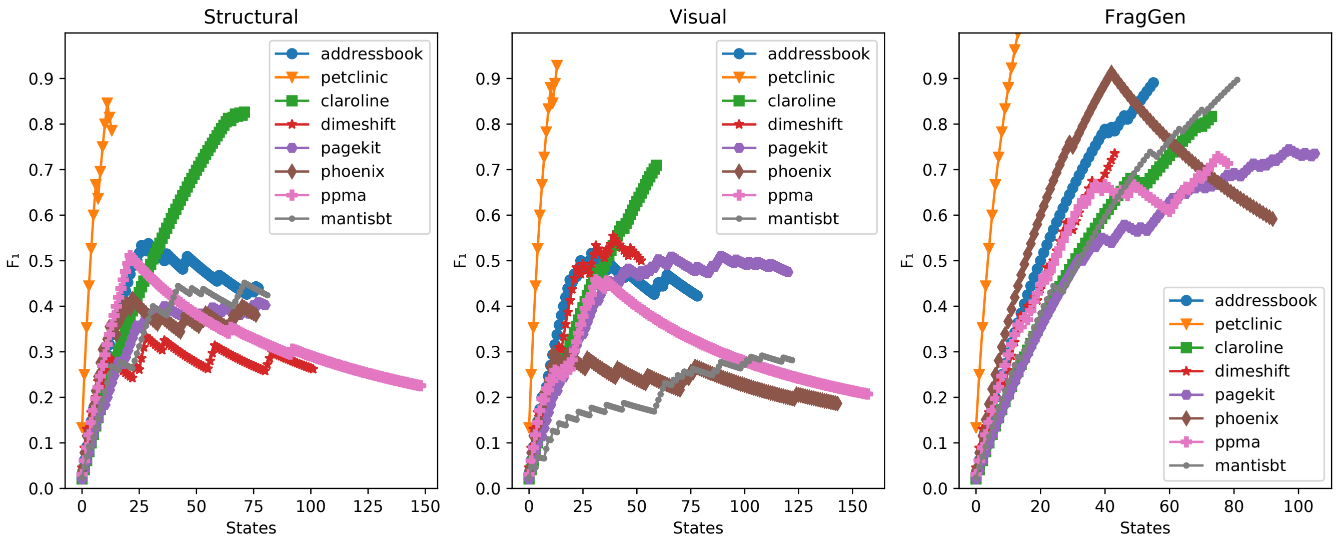


Figure 3.8: F_1 as states are detected and added to the model.

3.5.2 Results

Figure 3.7a shows the classification F_1 being optimized for the structural and visual whole-page techniques by a bayesian optimizer in 10,000 trials. To make the data comprehensible, we divided the 10,000 trials into 100 intervals and plotted a single data-point per interval in Figure 3.7a. We chose the maximum F_1 found in the corresponding interval of 100 trials as the representative data point of the interval.

As seen in the Figure 3.7a, FRAGGEN with an F_1 score of 0.836 performs better than the two whole-page techniques. FRAGGEN's F_1 score is 55% better than structural technique and 9% better than visual technique, which have scores of 0.535 and 0.767 respectively.

Figure 3.7b shows the correctly classified state-pairs for each of the four state-pair classes in the dataset for the optimal thresholds. FRAGGEN correctly classifies the highest number of Nd_3-*struct* as well as distinct (*Di*) state-pairs, while the visual technique is the best at classifying clones (*Cl*) and Nd_2-*data* state-pairs. Structural is the worst performer overall with a slightly better Nd_3-*struct* detection compared to visual. Overall, FRAGGEN correctly classifies 77,261 state-pairs while the visual and structural techniques correctly classify 73,920 and 64,872 state-pairs respectively.

Out of the 14,738 near-duplicates in the dataset, FRAGGEN detects 7,913 which is 188% and 82% better than the structural and visual techniques that detect 2,746 and 4,333 near-duplicates respectively. The result is significant because the whole-page techniques performed worse despite having an obvious advantage of classifying the same dataset that is being used for optimization. As the previous study [173] shows, given a set of state-pairs from an unseen web app, the whole-page techniques are unlikely to perform at a similar level using the same thresholds.

Indeed, the study delves deeper into the threshold selection for generating optimal models; it concludes that 1) the thresholds should be chosen specific to each web app and 2) a ground-truth should be created for any new web app in order to obtain optimal thresholds. Even after such an optimization per web app, as we illustrate with an example in Section 3.2.2, reliance on thresholds creates an inherent limitation for whole-page techniques in separating the Nd_3-*struct* near-duplicates

Table 3.5: Comparison of inferred models (*eight subjects*)

SAF	Model Quality			Labelled Web States				Termination
					Near-Duplicates			
	Pr	*Re*	F_1	Unique	Nd_2	Nd_3	*All*	
Structural	0.46	0.51	0.47	249	44	282	326	2
Visual	0.45	0.52	0.46	244	325	116	441	1
FRAGGEN	0.79	0.83	0.80	411	90	29	119	4

from distinct state-pairs.

In RQ_2, we investigate the quality of models inferred by each of the competing techniques configured with the optimal thresholds. We obtained optimized thresholds that generate the best model for each of the subjects from the empirical study [173].

3.6 Model Inference Comparison (RQ_2)

3.6.1 Procedure and Metrics

Model quality. We measure the quality of an inferred model in terms of its coverage of the app state-space, recall (*Re*), and amount of duplication, precision (*Pr*), by manually analyzing it with reference to a ground truth model for the web app using a methodology established in prior research [173]. The ground truth models for our subject apps taken from a published dataset [147] represent the functionality of a given web app using a minimal set of states and transitions. A ground truth model is a set of unique states. To analyze a given inferred model, each state in the inferred model is manually *mapped* to one of the ground truth states. A state ($\mathcal{S}_m$) in the inferred model is mapped to a state ($\mathcal{S}_g$) in the ground truth if the manual classification of the state-pair ($\mathcal{S}_m$, $\mathcal{S}_g$) is either clone, Nd_2-*data* or Nd_3-*struct*. Every ground truth state that is mapped to at-least one state in the inferred model is *covered* by the inferred model. We then compute *Pr* of the inferred model as the ratio of *covered* ground truth states to the total states in the inferred model, and *Re* as the ratio of *covered* ground truth states to the total number of ground truth states.

Experiment set-up. For each of our subject apps, we generate models using each technique by setting a maximum exploration time of 1 hour to be the *stopping criteria*. We use *Google Chrome (v82.0)* browser, and reset the subject app after every crawl to remove any back-end changes done by previous run. For a fair comparison, we configure each technique to use exactly the same crawl rules (e.g., form fill data). A technique can also *terminate* exploration before the stopping criterion is invoked by the crawler. Table 3.5 shows the number of times a technique terminated on its own.

3.6.2 Results

Table 3.5 shows the overall statistics for all the eight subject apps for FRAGGEN compared to structural and visual techniques. For recall, on average, FRAGGEN covered 83% of the state space, which is 60% higher than visual, the next best technique. For precision, at 79%, FRAGGEN produced models with 71% higher precision than structural, which itself performed slightly better than visual. In total, FRAGGEN added only 119 near-duplicates in all models, compared to the 326 by structural and 441 by visual, which are nearly 3 and 4 times higher, respectively. FRAGGEN also discovered 411 unique states overall, while the existing whole-page techniques detected 37% less app states in aggregate terms.

Overall, the F1 measure of FRAGGEN is 0.80, while that of structural and visual whole-page techniques are 0.47 and 0.46, respectively. Table 3.4 shows the details for all the subject systems. FRAGGEN consistently produced models with better F_1 except for Claroline where structural technique's model is marginally better.

When F_1 of the models is plotted against the states being added to the model, as shown in Figure 3.8, it can be seen that the F_1 score for existing techniques does not improve after an initial exploration period. As a result, the final models generated by existing techniques can sometimes deteriorate over time as more and more near-duplicates are added. FRAGGEN, however, keeps improving the model quality when given more time as it diversifies the exploration to discover unseen states while avoiding the addition of near-duplicates to the model.

One of the main reasons for this trend is that existing techniques exercise simi-

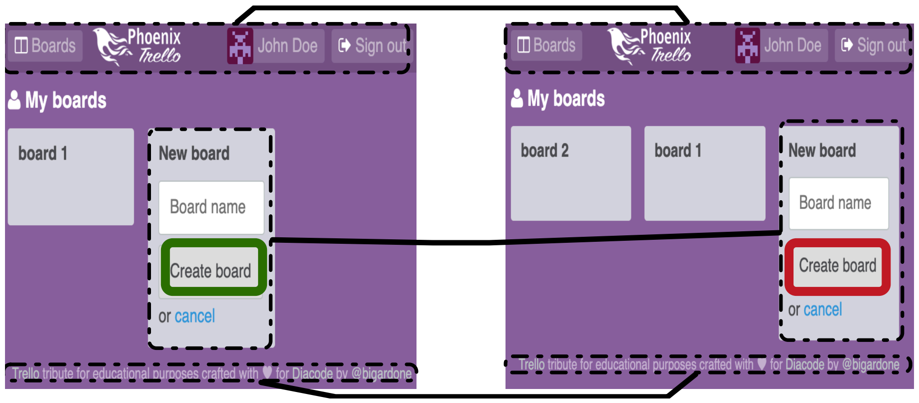


Figure 3.9: Detected near-duplicates in phoenix

lar actions repeatedly, and in dynamic web apps, this often results in the creation of near-duplicates and infinite loops, as mentioned in Section 3.2.2. Consider a real example from our subject Phoenix, where the action "create board" is available in two model states as shown in Figure 3.9. In both states, the action is functionally similar as it creates another board and therefore, need not be exercised more than once. However, existing techniques, which rely on whole-page comparison, cannot infer such similarities and keep creating boards and Nd_3-*struct* near-duplicate states as they repeatedly exercise the "create board" action.

On the other hand, FRAGGEN is able to identify similar actionables through fine-grained fragment-based analysis to avoid creating Nd_3-*struct* states and successfully diversify the exploration. Indeed, as the Table 3.5 shows, FRAGGEN added just 29 Nd_3-*struct* states overall while *structural* and *visual* techniques added 282 and 116 respectively.

The trend is prominently noticeable in Figure 3.8, where FRAGGEN is able to improve the model continuously by avoiding addition of Nd_3-*struct* near-duplicates to the model and seeking out unique actionables to exercise. As a result, FRAGGEN also terminates exploration for 4 out of 8 subjects apps as shown in Table 3.5, whereas structural and visual techniques terminate only 2 and 1 times respectively.

3.7 Regression Testing Suitability (RQ_3)

As part of assessing the suitability of generated tests in regression scenarios, in RQ_{3a}, we execute generated tests on different browser/platform and app versions, in order to evaluate the model inaccuracies and fragility of test cases in addition to the effectiveness of techniques in detecting application changes. In RQ_{3b}, we further evaluate the robustness and effectiveness of test oracles by simulating evolution through mutation of recorded web states.

3.7.1 Test Breakages (RQ_{3a})

Procedure and Metrics

The goal of our evaluation in RQ3a is to assess how FRAGGEN compares against existing techniques that generate regression tests automatically. There are not many techniques available that generate regression tests for web applications currently. We compare tests generated by FRAGGEN against tests generated by CRAWLJAX, which also employs exploration paths to generate tests from the crawl model.

While FRAGGEN uses its fine-grained fragment analysis to generate test oracles, for the whole-page techniques, CRAWLJAX is configured to generate test oracles that use the same whole-page comparison as the one used for model inference, namely, structural (RTED) and visual (Histogram).

Table 3.7 shows the three regression test scenarios used in our evaluation, where in ε_1 and ε_2, we execute tests on the same app version but vary browser/-platforms from the crawl, allowing us to evaluate the validity and robustness of the generated test suite. We then execute tests (ε_3) on a different version of the web app to determine the effectiveness of the test suites in detecting real application changes.

Table 3.6: Regression test run results

	Whole-page techniques												FragGen			
	Visual				**Structural**				**Average**							
Test Execution	ε_1	ε_2	ε_3	***All***	ε_1	ε_2	ε_3	***All***	v_0	v_1	***All***		ε_1	ε_2	ε_3	***All***
Action Success %	83.6	83.2	63.0	76.6	89.1	88.8	69.0	82.3	86.2	66.0	79.5		*99.5*	*99.9*	93.3	97.6
Oracle Success %	*0.0*	*0.0*	*0.0*	*0.0*	53.1	51.3	22.8	*42.4*	26.1	11.4	21.0		*98.6*	*97.5*	64.8	87.0

Table 3.7: Regression test execution set-up

Execution	app_version	Platform	Browser
Crawl	v_0	MacOS-14 [82]	chrome-82
TestSuite ε_1	v_0	MacOS-14	chrome-83
TestSuite ε_2	v_0	RHEL-7 [125]	chrome-84
TestSuite ε_3	v_1	RHEL-7	chrome-84

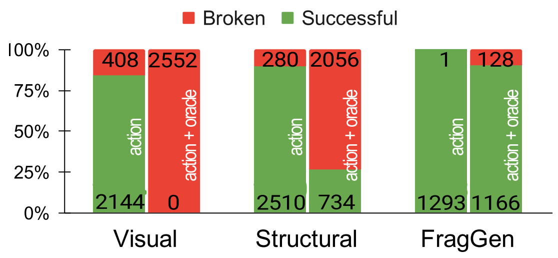


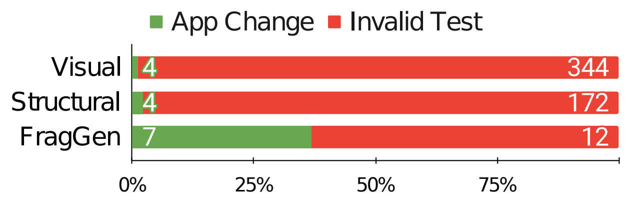
Figure 3.10: Test breakages on the same app version ($\varepsilon_1, \varepsilon_2$)

RQ$_{3a}$ Results

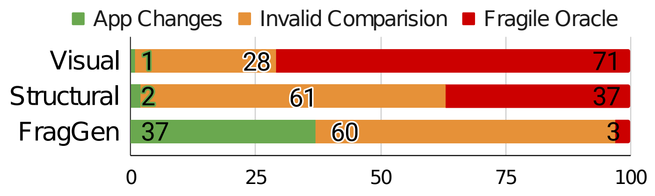
As seen in Table 3.6, whole-page techniques generated test actions that succeeded only 86.2% on an average. A potential cause for failure of remaining nearly 14% actions on the same version could be limitations in handling the near-duplicates for existing techniques. As Figure 3.10 show, breakage of these test actions resulted in breakage of 16% and 10% of tests in visual and structural test suites respectively without even considering the test oracle fragility. When test oracles are considered in declaring test breakages, all visual test oracles fail during test executions causing 100% test breakage, while 52% structural oracles fail breaking nearly 74% of tests in the same app version.

In contrast, FRAGGEN can execute test actions on the same version of the web app with nearly 100% success rate. As Table 3.6 shows, in $\varepsilon_1, \varepsilon_2$, FRAGGEN's test oracles also have a 98% success rate on average, showing greater adaptability to changing execution environments compared to the whole-page techniques.

Next, we evaluate test suite effectiveness in detecting app changes on a different version of web app using the test execution (ε_3). We follow the standard web testing practice where tests created for an existing version (v_0) are executed on a



(a) Test action failures (ε_3)



(b) Random test oracle failures (ε_3)

Figure 3.11: Manual analysis of failed test actions and oracles on a different version of the web apps

new version (v_1) of web app in order to detect faults through manual analysis of test failures. We mitigate the potential bias in manual analysis of test failures by labelling *all* observed application changes to be faults.

When we manually analyzed all test failures in ε_3 as shown in Figure 3.11a, we found that both structural and visual test suites detected 4 app changes each and produced 172 and 344 false positives respectively. In contrast, FRAGGEN was able to detect 7 app changes with only 10 false positives, significantly reducing the manual effort required to identify app changes and maintain test suites.

We then manually analyzed 100 randomly selected oracle failures for each technique to label the cause of failure. In this manual analysis, if the test state matches the expected state, we categorize the failure as either an *app change* if we notice a change in application behaviour or a *fragile oracle* as the failure represents a limitation of state comparison. Our analysis depicted in Figure 3.11b, shows that 71 visual and 37 structural oracles failed because of fragility in state comparison, whereas FRAGGEN generated only 3 fragile test oracles. Of the 100 examined failures, while the structural and visual oracles only detected 2 and 1 app changes respectively, FRAGGEN could detect 37 app changes.

When the test state and expected state do not match for the failure being analyzed, we categorize it as an invalid comparison, where typically test execution does not reach the target app state due to failure of earlier test actions or invalid test sequences due to changed app behaviour. Further root cause analysis of such failures is difficult even with domain knowledge, which makes it challenging to directly evaluate test oracle robustness through regression test executions. We mitigate this problem in RQ_{3b}, where the test oracles are compared on a synthetic dataset generated through mutation of recorded web pages.

These results show that the tests generated by FRAGGEN are significantly more robust than existing techniques in regression testing scenarios involving the same version of the web app, and show greater efficacy in detecting evolutionary app changes with significantly lower manual effort.

Table 3.8: Mutation operators for DOM nodes

Type	**Tags or Attributes**	**Description**	**Example** *(original : mutant)*
Attribute	{id, class, title}	Modifies mentioned attribute value of any node.	[<div id=”a”>] : [<div id=”aMut”>]
Tag	{span, h1-h6, p}	The tag name is changed to a similar tag.	[<h1>xyz</h1>] : [<h2>xyz</h2>]
Subtree	{div, table, tr, td, ul, li, p}	Deletes all children of selected container node.	[<tr><td>xyz</td></tr>] : [<tr></tr>]
Text	{h1-h6, p, b, i, }	The text content of selected leaf node is changed.	[<h1>abc</h1>] : [<h1>abcMut<h1>]

3.7.2 Effectiveness and Robustness of Test Oracles (RQ_{3b})

In RQ_{3b}, we simulate evolutionary app changes using mutation analysis to evaluate the suitability of generated test oracles for regression testing.

Mutation Analysis Methodology

We define four mutation operators for DOM nodes as shown in Table 3.8 in order to generate mutant web pages. Similar mutation operators for GUI artifacts have been proposed in prior web [100, 104, 120], Android [77] and GUI [7, 107] mutation analysis research. Since RQ_{3b} only aims to evaluate test oracles, we find this approach of mutating recorded web pages to be adequate and more flexible, instead of mutating the actual web app source code as prior work does.

Given a crawl model and a test execution trace, our mutation analysis tool applies a random mutation to one of the model states and compares the mutant to the corresponding states in the test trace using each of the three competing techniques.

We partition the generated mutants into either app changes or irrelevant to functionality based on their visibility in the GUI. We categorize (1) the mutants not visible on the page as equivalent mutants that test oracles should tolerate and ignore, and (2) all visible mutations as app changes that should be detected by effective test oracles. As Table 3.9 shows, we use 3,042 Visible mutations to compute effectiveness. Robustness is computed using 6,013 equivalent mutants that are of three kinds : *None* - where no mutation has been applied on the state; *AttributeMutator* - an attribute is mutated; and, *Invisible* - where a hidden web element is mutated.

Note that we recognize the possibility of invisible DOM or attribute changes affecting web page functionality. However, we expect that such changes manifest in failure of test actions, which is outside the scope of RQ_{3b}, as our set-up does not execute test actions on these mutated states.

Equivalent mutants of the kind *None* may still have DOM changes that result from a variety of reasons such as autogeneration of pages in back-end. We still consider them to be equivalent because the changes do not relate to either 1) change in source code which was unmodified or 2) change in test execution which remained consistent.

Finally, we consider a test oracle to be suitable for regression testing if it has

Table 3.9: Effectiveness and Robustness of test oracles

Score	Mutation		**Structural**	**Visual**	FRAGGEN	
	Type	#			*No-Mem*	*With-Mem*
	Visible	*3042*	2235	2908	3001	3001
Effectiveness			**73.5**	**95.6**	**98.7**	***98.7***
	None	*4620*	198	1964	921	136
	Attribute	*770*	10	585	166	157
	Invisible	*623*	553	577	275	274
	Total	*6013*	761	3126	1362	567
Robustness			**87.3**	***48.0***	***77.3***	**90.6**

high robustness as well as effectiveness.

FRAGGEN configuration. As described in Section 3.3.2, during model inference, FRAGGEN identifies data-fluid fragments and employs this knowledge to generate test oracles. In Table 3.9, *No-Mem* refers to test oracles if only fragment-based comparison is used; and *With-Mem* shows the test oracles that employ knowledge of data-fluid fragments.

As shown in Figure 3.6, without the knowledge of data-fluid fragments, we cannot lower the severity of warnings for changes that are typical for the web app under test and therefore may not necessarily be application bugs.

Results

As Table 3.9 shows, our mutation analysis experiment resulted in a total of 3,042 randomly generated visible mutations that should be detected by the oracles and 6,013 equivalent mutants that the oracles should be tolerant to.

Overall, amongst the existing techniques, *visual* test oracles detect 95.6% of visible mutations while *RTED*, the structural comparison technique which only considers HTML tags for equivalence, is tolerant to 87.3% of equivalent mutants encountered during the experiment. However, visual oracles are fragile, with a

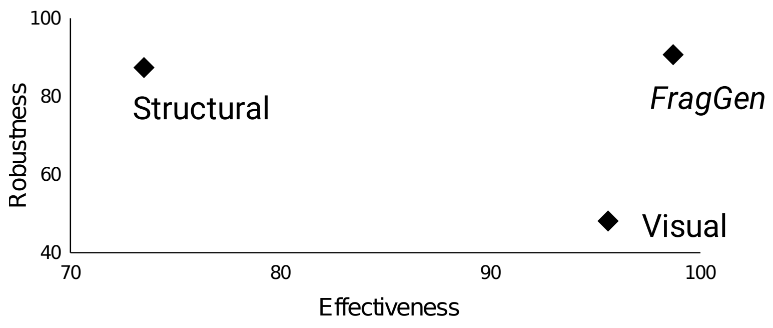


Figure 3.12: Effectiveness vs Robustness of test oracles

robustness score of only 48%, whereas structural oracles fail to detect 26.5% of the visible mutants.

In contrast, test oracles generated by FRAGGEN are able to detect 98.7% of visible mutants in both the configurations. When *memoization* knowledge is applied in the *With-Mem* configuration, FRAGGEN has the highest robustness of all the competing techniques by detecting 90.6% of the equivalent mutants. In the *No-Mem* configuration, when only fragment-based state comparison is used, the robustness drops to 77.3%, which is still 61% more than visual. The difference in numbers between the two configurations is because of handling the *None* category, where no mutation has been applied to the recorded web pages. It can be concluded therefore, that FRAGGEN is able to make use of memoization and data-fluid fragments to improve the robustness of test oracles.

As the Figure 3.12 shows, existing whole-page oracles can either be highly effective or highly tolerant but cannot balance both the aspects, causing either a large number of false positives or false negatives respectively. FRAGGEN's test oracles on the other hand, use fine-grained fragment analysis with memoization to outperform existing techniques in both effectiveness and robustness at the same time, making them suitable for regression testing of modern web apps.

3.8 Discussion

Granularity of fragments. During our experiments, we found that determining equivalency of certain fragments that are too small such as $\mathcal{F}_3$, shown in dotted lines in Figure 3.4, using DOM or visual characteristics results in drawing false equivalence between semantically different web pages. In order to avoid this, using a predefined threshold, FRAGGEN prunes out the smaller fragments and does not use their equivalence in determining page similarity.

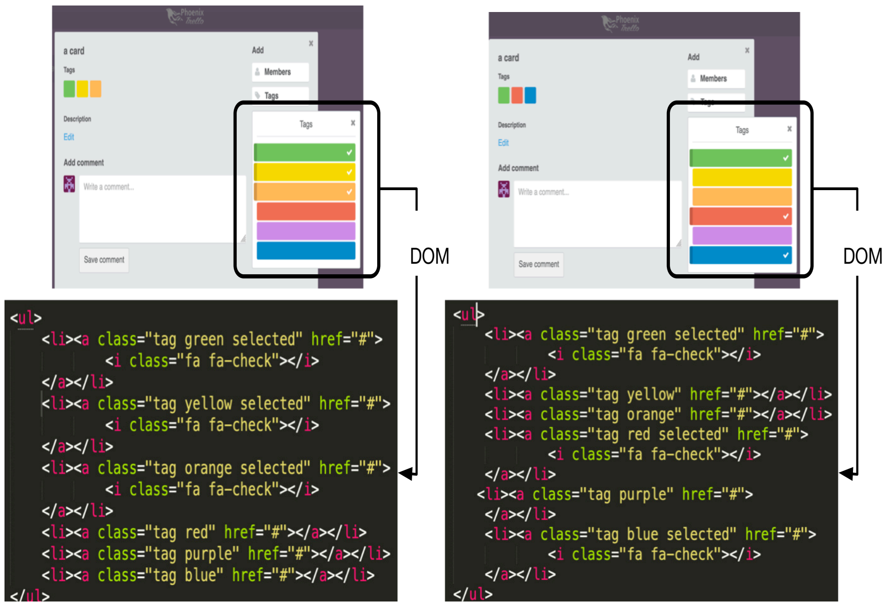
One such example is shown in Figure 3.13, where each individual tag can have two states – selected or not. Therefore each tag can be semantically equivalent to other depending on this state. However, as we avoid comparing the smaller fragments of individual tags, we cannot determine equivalency of parent fragments which differ structurally upon selection of different tags. The number of possible near-duplicates we cannot detect in such cases can explode in number, and is the reason for the reduced precision of the web app model for *phoenix*.

Even-though smaller fragments with even one web element can be separate functional entities, such fine-grained comparison needs semantic inference beyond simple DOM and visual characteristics, which we leave for future work.

Test oracle generation. FRAGGEN currently identifies data-fluid fragments assuming that during model inference, only the events fired by the crawler can cause persistent changes. Such assumption can fail on live web applications where multiple users can concurrently induce back-end changes. However, since our technique is meant for test environments, we consider our assumption reasonable.

Model Inference in other domains. While our FRAGGEN implementation is specific to web apps, conceptually our approach to model generation based on fragments (instead of whole pages or whole screens) can be applied to other domains such as mobile apps or desktop applications.

Threats to validity. Using a limited number of web apps in a controlled setting in our evaluation poses an external validity threat and further experiments are needed to fully assess the generalizability of our results; we have chosen eight subject apps used in previous web testing research, pertaining to different domains in order to mitigate the threat. Threats to internal validity come from the manual labelling of web pages and changes, which was unavoidable because no automated method could provide us with the required ground truth. To mitigate this threat, we performed the labelling by following a process established in prior work. For reproducibility of our findings, we made our tool publicly available [169] along with



```
<ul>
    <li><a class="tag green selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
    <li><a class="tag yellow selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
    <li><a class="tag orange selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
    <li><a class="tag red" href="#"></a></li>
    <li><a class="tag purple" href="#"></a></li>
    <li><a class="tag blue" href="#"></a></li>
</ul>
```

```
<ul>
    <li><a class="tag green selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
    <li><a class="tag yellow" href="#"></a></li>
    <li><a class="tag orange" href="#"></a></li>
    <li><a class="tag red selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
   <li><a class="tag purple" href="#">
    </a></li>
    <li><a class="tag blue selected" href="#">
            <i class="fa fa-check"></i>
    </a></li>
</ul>
```

Figure 3.13: Undetected near-duplicate fragments in phoenix

usage instructions and used subject systems.

3.9 Related Work

In web model inference, CRAWLJAX [96] explores dynamic web apps with client-side JavaScript by triggering actions and analyzing the DOM. WebMate [38] crawls web apps using a state equivalence based on actionables in web pages. FRAGGEN is built upon CRAWLJAX framework by adding a new fragment-based state abstraction and modifying several core components that drive state exploration.

For regression test suite generation from web app models, Mesbah et al. proposed test generation based on model coverage [96] while Marchetto et al. [89] used coverage criteria based on semantically interacting events. FRAGGEN uses the exploration paths that cover all the events and states in the inferred model for test generation.

Instead of crawl models, Biagiola et al. use *page objects* derived from crawl models, and employed search-based [21] techniques, diversification [23] of test events to generate test cases. These page objects used in existing work are derived from crawl models generated by the baseline Crawjax that was manually configured with thresholds by the researchers for their experiments. In contrast, we generate test cases directly from crawl models without any manual threshold selection using the paths followed by our exploration strategy during model inference. Although the generation of page objects from crawl models is largely automated, some manual effort such as specifying guards to avoid test dependencies is required for optimal results. Indeed, the superior crawl models generated by FRAGGEN should improve the test cases generated by the existing techniques based on page objects.

Prior research used histogram [32] to detect cross browser differences and manually specified DOM invariants [95] to create robust test oracles. FRAGGEN, however, automatically generates robust fine-grained test oracles at fragment level by leveraging model inference.

To assess the quality of web test suites, prior work mutated source code artifacts such as JSP web pages [120, 121], and client-side JavaScript [100, 104]. We assess the quality of test oracles by mutating the recorded web pages.

Sun et al. [149] employed page fragment-based exploration for efficient information retrieval while we use fragmentation for model inference and test oracle generation.

FeedEx [98] employs a combination of client-side JavaScript code coverage, DOM and Path diversity to prioritize re-exploration of already explored actions. WebExplor [181] by Zheng et al. also proposes a method to prioritize web elements that were already explored by rewarding discovery of new states in previous result of the same action. Both these existing techniques assign the same priority to all unexplored actions. State exploration strategy is a concern for UI testing in general and Degott et al. [40] also prioritize exploratory actions by first exploring all available actions in a mobile app and analyzing the results of execution. In contrast, FragGen is able to make use of fragment based equivalence of unexplored actions to diversify exploration.

Bajammal et al. [19] look at the testing of canvas elements which can have visual changes but no DOM changes. In contrast, FRAGGEN would consider such changes to be non-functional in nature.

To the best of our knowledge, we are the first to employ page fragmentation to establish state equivalence for web app testing. Our novel state comparison combines both structural and visual analysis of the identified fragments to effectively detect near-duplicates. Our tool, FRAGGEN, uses this novel state comparison to infer precise models, as well as generate effective and reliable regression test suites.

3.10 Conclusions and Future Work

Automated model inference and test generation for complex dynamic modern web apps is a challenging problem because of the presence of near-duplicates that cannot be detected by whole-page analysis employed by existing techniques. We developed a novel technique, FRAGGEN, which uses smaller page fragments to detect near-duplicates and diversify web app exploration to generate precise models while covering a high percentage of application state space. FRAGGEN is also able to generate oracles that are suitable for regression testing as they are highly effective in detecting visible app changes while being tolerant to minor changes unrelated to functionality.

As part of the future work, we plan to improve our state comparison technique through inference of web page semantics when structural and visual characteristics are inadequate.

Chapter 4

Mutation Analysis for Assessing End-to-End Web Tests

4.1 Introduction

Modern web apps are highly dynamic in nature and contain a heterogeneous collection of server-side and client-side components that interact in real-time to update the web page in response to user requests. Consequently, testing web apps programmatically is challenging and is often performed in an end-to-end (E2E) fashion by exercising the GUI functionality of web apps. Given the short release cycles of web apps, automated regression testing using UI tests plays a significant role in the validation of web app changes.

Because of their importance, companies currently invest manpower in creating and maintaining such UI tests suites. However, despite this reliance on UI test suites to validate web app functionality, currently, there is no universal tool to determine their fault-finding capabilities. Therefore, in practice, UI test suite adequacy is determined by coverage of common use case scenarios, and certain server-side and client-side code. However, such coverage metrics are generally considered to be limited for assessing test effectiveness [65, 180].

Instead, mutation analysis, which mimics programmer errors by making small changes to the application has become an accepted norm in establishing the fault revealing capabilities of test suites. Existing mutation analysis tools for web apps are not universally applicable as they are designed for specific programming languages or web development frameworks. Therefore, currently, there exists no mutation analysis framework to assess the actual effectiveness of web-based UI test suites.

In this chapter, we propose MAEWU, a **M**utation **a**nalysis framework for **E**nd-to-end **w**eb **U**I test suites. MAEWU mutates the dynamic DOM of the web app in the browser in order to bypass the limitations of a source code-based mutation analysis employed by existing techniques. Consequently, MAEWU only requires the URL of the web app and its test suite to perform mutation analysis; it neither requires access to the source code of the web app nor employs any proxy to instrument the client-side code.

Using the dynamic DOM state as an artifact for mutation is conceptually novel because we mutate the output instead of the actual source code written by programmers. While DOM mutation allows for universal applicability in all web apps, it also poses a unique challenge vis-a-vis its availability during test execution. First, in traditional mutation analysis, a mutation applied to a source code artifact is preserved over multiple invocations during test execution. The same does not apply for a mutated DOM in a browser which can disappear upon navigation or a page reload. In addition, as web pages are dynamically generated, applying a mutation consistently to each appearance of the browser state is challenging as a state equivalence between concrete instances needs to be established. Second, a typical modern web page is essentially a set of individual UI components, where each component can appear in multiple different pages. As a result, mutating a web element such as a navigation link necessitates identifying all its instances across web pages. We make use of an automatic page fragmentation technique and a tree comparison technique in order to accomplish this task. A similar challenge exists for other kinds of GUI testing such as desktop applications [107] or Mobile apps [41]. However, researchers in those areas are able to rely on mutating the source artifacts because of uniformity in technologies used to generate the corresponding UI. For example, Android apps use Java for handling UI interactions and layout files to define UI structures, which can be mutated to cover various classes of bugs. In web testing, however, such homogeneity of languages exists (e.g., a web app could be built using a combination of JavaScript, HTML, CSS, and PHP).

One of the foremost requirements for an effective mutation analysis tool is the set of mutation operators designed to generate artifact-specific transformations that can mimic programmer errors. While existing techniques [120, 135] have proposed several DOM specific mutation operators, they predominantly rely on mutating the source code artifacts of programming languages like Java or JavaScript. Existing DOM operators also do not cover the wide range of possible DOM transformations that may mimic application faults. As a matter of fact, to the best of our knowledge, there exists no prior research to establish the relationship between UI or DOM changes in the browser and application faults for web apps. Existing work [60] on categorization of web app faults is limited to the location (e.g., server, client) of the bug. Therefore, in this work, we manually analyze 250 bug reports from open source web app bug repositories to identify the UI manifestations of real faults, and design 16 mutation operators (MOs), which manipulate the interactive behaviour and appearance of web elements to mimic real faults.

We evaluate MAEWU on six open source web apps. Our results show that MAEWU (1) generates non-equivalent mutants, (2) consistently applies the mutants dynamically across test executions. The report generated show the mutation score of a given UI Test suite along with the failed mutants to help improve the test suite.

We summarize the contributions of our work as follows:

- We define a set of mutation operators for dynamic mutation of web pages in the browser and remove the need for modifying the source code to evaluate the efficacy of end-to-end test suites.
- We developed a mutation analysis framework, called MAEWU, that is universally applicable to end-to-end test suites of all web applications regardless of the back-end and front-end programming languages used for app development. MAEWU [170] is publicly available.
- We also provide a dataset of 250 labelled bug reports from open source web apps, and 300 labelled mutants that can be used by future researchers to determine the efficacy of UI test suites and mutation analysis techniques.

4.2 Background

In this section, we provide a brief overview of the current state of practice in web testing. Selenium Web Driver is one of the most popular tools for automated UI testing. It provides API in most popular programming languages such as Java and Python to remotely control web browsers and automate UI interactions. Listing 4.1 shows an example selenium UI test case taken from our subject set. Typically, test cases start by fetching the home page of the web app by using the provided url. Thereafter, a series of test actions are performed and test oracles are used to verify the resulting browser state according to a business case scenario of the web app.

Listing 4.1: Selenium JUnit Test Case

```
private WebDriver driver = new ChromeDriver();

@Before
public void setUp() {
        driver.get(app_url);
}

@Test
public void testCollabtiveLoginUser() throws Exception {
        driver.findElement(By.id("username")).sendKeys("username001");
        driver.findElement(By.id("pass")).sendKeys("password001");
        driver.findElement(By.cssSelector("button.loginbtn")).click();
        driver.findElement( By.xpath("//*[@id=\"mainmenue\"]/li[2]/a")).click();
        assertTrue( driver.findElement(By.cssSelector("body")).getText()
                        .matches("^[\\s\\S]*username001[\\s\\S]*$"));
```

As the web app evolves, such UI tests created for an earlier version of web app are used to validate existing functionality and detect any regression bugs that may have been introduced in the newer version. An adequate UI test suite should therefore cover the entire functionality of the web app through a combination of test actions and oracles to aid early detection of regression bugs through test failures.

As web apps can be incredibly complex with heterogeneous components written in multiple server-side and client-side programming languages, their bugs can be equally daunting to detect and fix. Typically, individual server-side components and client-side JavaScript are tested through unit testing while UI testing is used as a form of end-to-end testing to validate high level use case scenarios from an end-user point of view. Several researchers [60] have attempted to characterize web application bugs in terms of their location, significance and so on by analyzing the bug repositories of open source web apps. As UI tests only have access to the browser state, a UI test suite can only reveal application bugs have a direct manifestation in the UI. Such UI bugs are the focus of our mutation analysis for ascertaining the quality of the test suites.

4.3 UI Manifestation of Real Faults in Web Applications

In traditional mutation testing, mutation operators are designed to perform code changes that imitate programmer errors that cause application bugs. As majority of the modern web pages are automatically generated, mutating them may not directly imitate programmer errors. On the other hand, relying on mutation of source code artifacts written by programmers is impossible given the fragmented nature of web development ecosystem. Therefore, we decided to design mutation operators to imitate the manifestation of application bugs on the UI of web pages. In order to do so, we first needed to understand the UI manifestation of web app bugs.

Manual analysis of real faults in web applications in the existing research [60] has focused on the location and root-cause analysis of faults in source code. Marchetto et al. [88] also defined 32 categories of faults that can be used for fault seeding in the web app source code. However, their work relied on introducing faults specific to program constructs and technologies in use at the time of publication.

In contrast, in this work, we are focused only on the front-end manifestation of bugs regardless of their root-cause and specific web development frameworks. Therefore, we collected real bugs reported for ten popular open source web apps shown in Table 4.1 with a minimum of 1000 downloads in sourceforge [139] or greater than 100 stars in GitHub. In total we collected 6331 reports tagged to be bugs from bug repositories. We then randomly selected 250 bug reports to be analyzed manually, where for each bug report, we ascertained the specific UI characteristics that were considered to be faulty, and created tags to reflect them.

Table 4.1: Bug Repositories

Name	**loc**	**languages**	**# Bugs**
tikiwiki [153]	4M	PHP, JavaScript	1975
reactive trader [124]	117K	C#, TypeScript	296
mrbs [103]	187K	PHP, JavaScript	495
pgweb [114]	262K	Go, JavaScript	553
tudu lists [158]	29K	Java, JavaScript	68
addressbook [116]	45K	PHP, JavaScript	173
crater [36]	80K	Laravel, Vue	380
claroline [34]	350K	PHP, JavaScript	336
koel [71]	475K	Laravel, Vue	1297
phplist [117]	1.3M	PHP, JavaScript	758

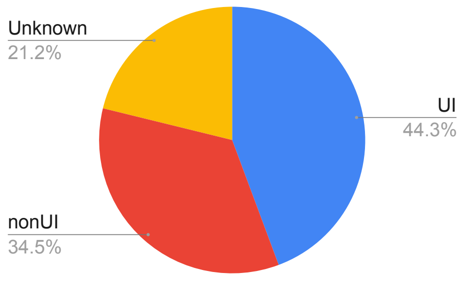


Figure 4.1: Analyzed Bugs

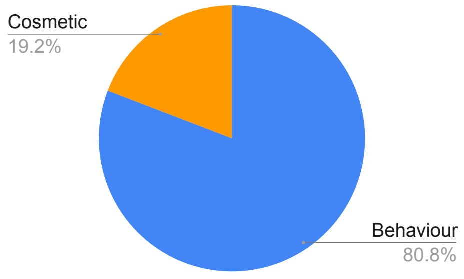


Figure 4.2: Bugs with UI Manifestation

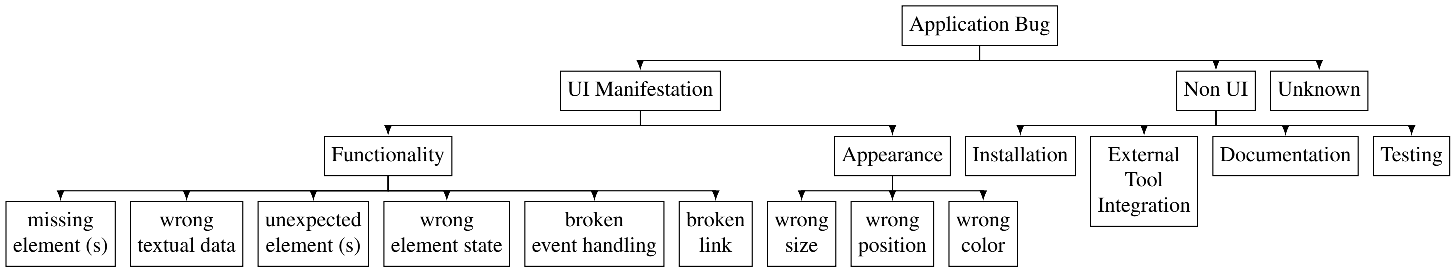


Figure 4.3: Web Application Bug Categories

Our bug report analysis for UI impact resulted in a hierarchy of categories as shown in Figure 4.3.

Of the 250 bug reports we analyzed, we found that nearly 44% mention the impact on UI, while 35% did not have any UI manifestation as shown in Figure 4.1. In order to be able to manually analyze the bug reports, we familiarized ourselves with each of the web apps in terms of the provided UI functionality. However, we still could not assess the UI impact from the report for 21% of the reports. Of the bugs which impact the UI, we found that 81% of them had an effect on the behaviour of the web app as shown in Figure 4.2. Bugs categorized to have broken functionalities were either because of incorrect handling of events associated with web elements or content related faults. Only 19% of the bugs concerned the appearance of the web page, that were either found to be due to incorrect rendering of the web pages or incorrect css styling resulting in overlapped or incomprehensible content.

Using this study on the UI manifestation of real bugs in web applications, we design our mutation operators for dynamic mutation of web pages in the browser.

4.4 Dynamic DOM Mutation

The functionality and data presented to the user through web app GUI is a DOM that is built together by back-end and front-end programs written in languages such as Java, PhP, JavaScript. UI test suites written using browser automation tools such as Selenium therefore indirectly test all these programs while exercising the DOM. Therefore, mutation operators targeting only client side JavaScript or server side JSP cannot assess the quality of the test suites as the resulting mutants only represent a subset of all possible changes that can occur in the DOM. In fact, espousing our sentiment, Mirzaaghaei et al. [101] argue that the coverage of UI test suites should be entirely determined using the GUI of the web app.

Considering the dynamic DOM state as a mutation target is interesting because it is essentially the output of the web app under test. In fact, the dynamic DOM is what end-to-end UI tests assess. UI tests are therefore expected to detect visual/-textual changes in the rendered web page through oracles, and detect behavioural changes by triggering UI actions. Mutation analysis of UI tests therefore will have

Table 4.2: Mutation Operators and their types

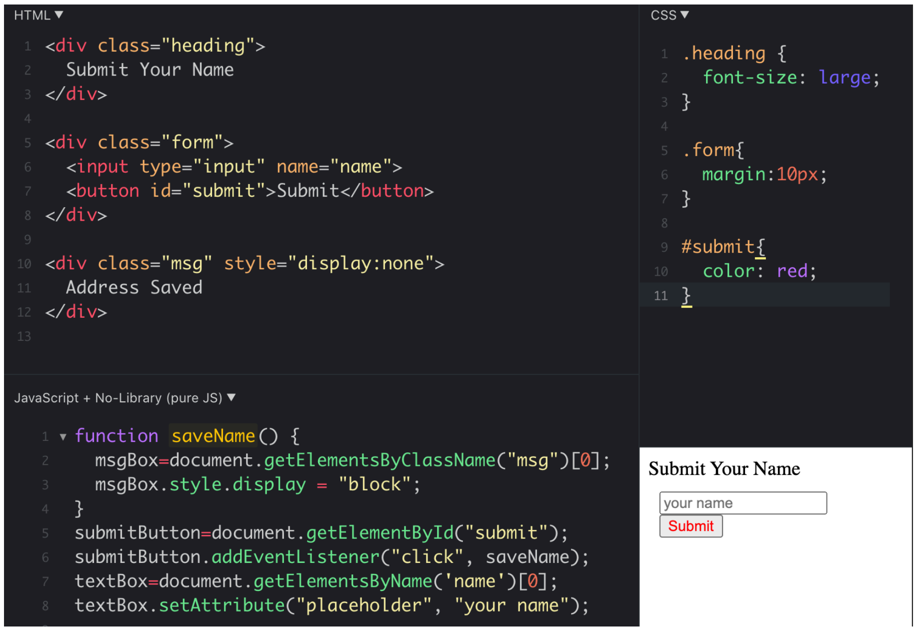
Type	Operator Name	Abbr
Attribute	AttributeAdd	AAM
	AttributeDelete	ADM
	AttributeModify	AMM
EventHandler	EventHandlerAdd	EHAM
	EventHandlerDelete	EHDM
	EventHandlerModify	EHMM
Tree	TreeInsert	TIM
	TreeDelete	TDM
	TreeMove	TMM
Content	ContentInsert	CIM
	ContentDelete	CDM
	ContentModify	CMM
Style	StyleVisibility	SVM
	StyleColor	SCM
	StylePosition	SPM
	StyleSize	SSM

to be designed to satisfy both these aspects of the UI testing to be considered useful.

Taking into account the UI impact of bugs, we designed 16 mutation operators, and placed them into five categories based on the aspect of web elements they target, as shown in Table 4.2. Generated mutants can potentially imitate bug categories corresponding to UI functionality, appearance or both. For example, a changed id or class attribute can potentially impact the behaviour as well as appearance of the corresponding web element.

However, it is important to note the possibility that the mutation *may not impact* the web page functionality or appearance at all in any way. For example, mutating the position of an already invisible element will not change the web page.

In the rest of this section, we describe the mutation operators in greater detail and provide examples using the sample web page shown in Figure 4.4.



```html
<div class="heading">
  Submit Your Name
</div>

<div class="form">
  <input type="input" name="name">
  <button id="submit">Submit</button>
</div>

<div class="msg" style="display:none">
  Address Saved
</div>
```

```css
.heading {
  font-size: large;
}

.form{
  margin:10px;
}

#submit{
  color: red;
}
```

```javascript
function saveName() {
  msgBox=document.getElementsByClassName("msg")[0];
  msgBox.style.display = "block";
}
submitButton=document.getElementById("submit");
submitButton.addEventListener("click", saveName);
textBox=document.getElementsByName('name')[0];
textBox.setAttribute("placeholder", "your name");
```

Submit Your Name

Figure 4.4: Sample Web Page with Source Code

4.4.1 Attribute

In modern web apps, attributes are used to define characteristics or properties of web elements, and therefore changing attribute values can impact both their appearance and behaviour. Some attributes such as *"src"* for ¡img¿ elements are used to define the image urls. Attributes such as *"action"* for ¡form¿ elements can even specify server communication.

For example, consider the sample web page shown in Figure 4.4. Using css rules, appearance of *heading* and *form* is set using *class* selector, while *id* is used for *submit* button. Our three attribute MOs are currently configured to mutate commonly used attributes taken from current HTML standard [5]. However, developers can even use custom attributes to accomplish the same. In any case, being an extensible framework, MAEWU can be configured to use any set of attributes considered to be important for specific web apps, in order to generate interesting mutants.

4.4.2 Tree

Through the tree mutation operators, we aim to alter the DOM structure of the web page. When a tree operator is applied to a web element, the element as well as its children get affected. For example, in our sample web page, applying TreeDelete operator on the *div* of the class form will lead to its deletion as well as its child elements, the *input* element and the submit *button*.

The TreeMove operator can imitate the appearance "position" bugs as well as impact functionality of the moved subtree because of the changed parent through inherited event handling or style rules. The TreeInsert and TreeDelete operators are designed to imitate the functionality bugs– *unexpected-elements* and *missing-elements* respectively.

4.4.3 Content

Of the content related bugs we analyzed as part of the study, some of the root causes included errors fetching data from back-end, parsing form input data, and client-side scripting errors that prevent rendering of data.

In each of these analyzed bugs, the displayed textual content is either incorrect, unexpected, or completely missing from the web pages. Our three mutation operators – *ContentInsert*, *ContentDelete* and *ContentModify* – are aimed at imitating these content bugs.

It is also worth pointing out that most UI test cases typically access only interactive web elements and are likely to miss the content related bugs as a result.

4.4.4 Style

Cosmetic or appearance bugs we analyzed related to unexpected positioning, size and color of specific web elements, primarily caused by wrong css properties computed in runtime. Our analysis of appearance related bugs revealed three computed (from static rules) CSS properties – *color, position, size* – We designed the three style MOs – *StyleColor, StylePosition, StyleSize* – that mutate corresponding css properties in runtime to imitate bugs causing content to be either incomprehensible, place elements in unexpected positions or have unexpected sizes. Our fourth style MO, *StyleVisibility* imitates the functionality bugs *missing-elements* and *unexpected-elements* by toggling the *visible* CSS property of web elements.

4.4.5 Event

We designed three MOs to cover the bugs related to *broken event handling*. Bugs of this category affect the behaviour of the interactive elements like buttons while often having no apparent change in the appearance and visible content of the web pages. In order to imitate such bugs, we created three MOs that insert, remove or modify the event handlers of web elements.

As such, broken event handling can result from a variety of reasons such as bugs on the server side, broken server communication, or even bugs in JavaScript libraries being used on the client side. However, since we are interested in imitating the eventual behaviour observed by the end users, we directly modify the event handlers for the web elements. In order to ensure the JavaScript code for event handlers themselves are valid, we reuse already seen event handlers for other elements within the web app.

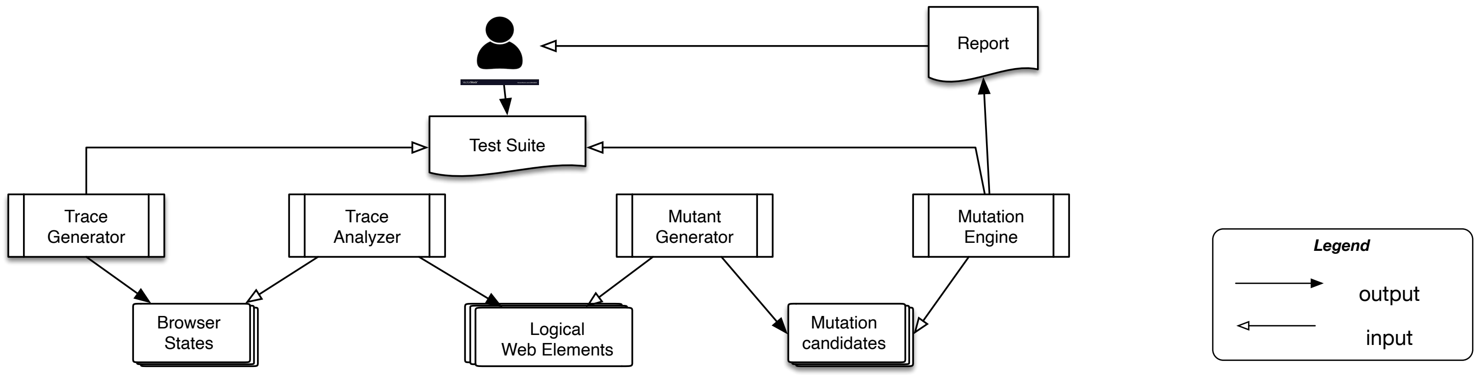


Figure 4.5: Technique Overview.

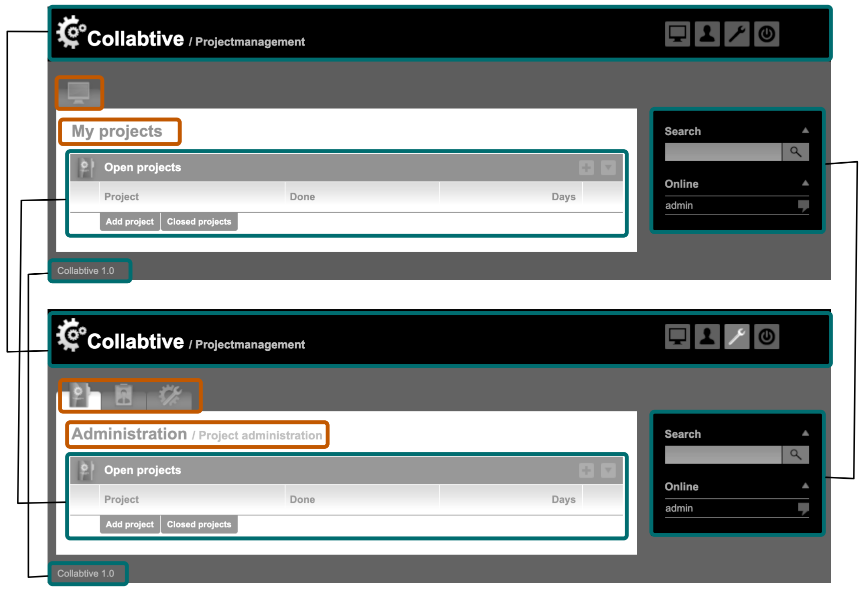


Figure 4.6: Example Web Pages

4.5 Technique

Our technique, MAEWU, aims to assess the mutation score for a given UI test suite and a web app URL by dynamically applying mutation operators to browser states. MAEWU contains four main components 1) Trace Generator (TG), 2) Trace Analyzer (TA), 3) Mutant Generator (MG), and 4) Mutation Engine (ME).

A high level architecture of MAEWU shown in Figure 5.2 indicates the inputs and outputs for each of the components. The mutation analysis performed by MAEWU can be divided in two main processes. While the first process concerns generation of a set of candidate mutants, the second part concerns evaluating the test suite efficacy using the candidate mutants.

To generate the candidate mutants, given a test suite, TG collects the test trace as a series of dynamic DOM states associated with test steps, and TA analyzes them to identify the reappearance of web elements across states by using a page fragmentation technique. MG then creates candidate mutants by applying mutation operators on the identified web elements. To evaluate the efficacy of the test suite, ME applies each candidate mutant to the dynamic DOM in the browser while executing the test suite.

In the rest of this section, we describe each of our components in detail.

4.5.1 Trace Collection

Given a UI test suite, the trace generator (TG) captures a test suite trace as a sequence of trace elements, recording browser states as defined in 19 before and after each test step. In addition, an "observer" script that runs in the browser records JavaScript accesses to web elements for each browser state during test execution.

Definition 19 (**Browser State** (S))**.** is a tuple ¡D,V, K¿ where D is the HTML source or DOM and V is the screenshot of the web page. K is the JavaScript access map for the browser state recorded by the observer script, where each map entry corresponds to a web element and its accesses.

A trace element as defined in 20 records the browser state transition in the web app caused by the test step execution.

Definition 20 (**Trace Element**)**.** is a tuple (ε, S_b, S_a, α), where the action α is performed upon the web element ε in the browser state S_b results in the transition to the browser state S_a.

4.5.2 Trace Analysis

Once the trace is collected for a web app, the trace analyzer (TA) first extracts a list of all web elements in the recorded browser states and a set of text tokens. The set of text tokens which we call the mutation data tokens (Δ) are extracted from content nodes as well as attribute values. Thereafter, since our approach of web app mutation use web elements from dynamic DOM as mutation artifacts, we designed the trace analyzer (TA) to cluster equivalent web elements into logical web elements (ω) in order to achieve a consistency in mutation. Formally, logical web elements are defined in 21.

Definition 21 (**Logical Web Element** (ω))**.** is a tuple ¡E, κ¿, where E is the set of concrete web elements $\{\varepsilon_1, \varepsilon_2, ..\}$ in which any two web elements $\varepsilon_i, \varepsilon_j$ are equivalent to each other. κ is the combined list for JavaScript accesses of all web elements

Algorithm 2: Extract Logical Web Elements

```
Input: states = [$S_1, S_2$..]                      /* Set of Browser States */
Output: LWE                    /* Set of Logical Web Elements ($\omega$) */
1  LWE=[]
2  foreach $S \in states$ do
3      elems = getWebElems($S$)            /* all elements of state */
4      foreach $\varepsilon \in elems$ do
5          foreach $\omega$ in LWE do
6              if belongsTo($\varepsilon$,$\omega$) then
7                  $\omega$.add($\varepsilon$)              /* belongs to cluster */
8                  added $\leftarrow$ true
9          end
10         if not added then
11             $\omega_{new} \leftarrow [\varepsilon]$                 /* create new $\omega$ */
12             LWE.add($\omega_{new}$)
13     end
14 end
15 Function belongsTo($\varepsilon$, $\omega$):
16     $\varepsilon' \leftarrow$ lwe[0]
17     F1 $\leftarrow$ getFragment(node1)            /* fragments from VIPS */
18     F2 $\leftarrow$ getFragment(node2)
19     if TreeComp(F1, F2) = 0 then                     /* Tree Edit Distance */
20         if XPath(node1) = XPath(node2) then
21             return True                 /* same relative XPath */
22         return False
23     return False
24 End Function
```

in E, where access for each element is extracted from the access map K of its parent state (S).

Consider the two example pages in Figure 4.6 that contain several web elements in common. If we decide to mutate the "search box" in page1, we need to ensure the same mutation is applied to it in the page2 as well if a reliable mutation score for the UI test suite is to be computed. A similar problem does not arise for the traditional source code mutation since the applied mutation is available for every instantiation of the corresponding line of code, regardless of the dynamic program state. For example, a mutation applied to a HTML source artifact in the server will be available each time the artifact is accessed from the browser.

However, establishing the equivalence of web elements is challenging because of the dynamic nature of modern web apps. Existing research shows that techniques relying on attributes such as ids to compare web elements tend to be unreliable, because such attributes are often generated dynamically. Similarly using

XPath locators in web pages is also not desirable because we want to compare individual web elements across different web pages. For example in Figure 4.6, the XPath for ”add Project” in the two pages is not the same.

Our solution is based on the observation that a single web page does not necessary provide a singular functionality. Instead, each web page UI is stitched together dynamically and contains independent UI components such as navigation bars that reappear in different web pages. We associate the ownership of web elements to smaller page fragments instead of the entire web pages, and use the equivalence of these fragments to establish similarity of web elements.

Our element extraction algorithm shown in algorithm 2 uses a popular page segmentation technique VIPS [26] to generate smaller page fragments, compare fragments using a tree comparison technique [112], and, uses relative XPaths of web elements inside these fragments for establishing their equivalence.

4.5.3 Generating Mutation Candidates

Given a set of logical web elements (ω) and available mutation data tokens (Δ), Mutant Generator (MG) generates a set of all possible mutation candidates by selecting appropriate mutation operators based on the characteristics of the web element and a random mutation data token if required.

Definition 22 (**Mutation Candidate** (C))**.** is a tuple (ω, O, δ) where ω is the logical web element upon which the mutation operator O is applied using the optional mutation data δ.

However, modern web pages are notoriously heavy [2, 4], where an “average” web page is 2MB in size, can contain more than 600 web elements of 32 different types. Without a notion of significance or importance associated with each mutant to allow for a selection strategy, the total number of mutants to analyze can quickly become unmanageable especially given the resource intensive nature of UI testing.

In this work, we employ a biased-spread random mutant selection strategy where the probability (Equation 4.1) of selecting a mutant depends on its score (Equation 4.2). Our score for a candidate combines 1) four static features of the web element – $isLeafNode$, $hasText$, $isInteractive$, $isDisplayed$, 2) three dynamic features based on the collected web element trace – $numRepetitions$, *numTestAccesses, numJavaScriptAccesses*, and 3) its relationship to already selected mutants. The score is positively impacted by high static (*St*) and dynamic (*Dn*) scores, and negatively impacted by the spread score (*Sp*) based on the presence of already selected mutants for the same web element.

The selection probability is defined as

$$Pr(C^a_j) = \frac{score(C^a_j)}{\sum_{i=1}^{n} score(C^a_i)} \tag{4.1}$$

where a, b and c are constants such that $(0 < a, b, c < 1)$, the set of 'n' available candidates ($\{C^a_{1..n}\}$) and the set of 'm' already selected candidates ($\{C^s_{1..m}\}$), and the candidate score is

$$score(C^a) = a * St(\omega^a) + b * Dn(\omega^a) - c * Sp(C^a, \{C^s_{1..m}\}) \tag{4.2}$$

Based on their relevance to the corresponding MO, the static features of web element capture the usefulness of a given candidate based on its likelihood of imitating a bug. For example, a candidate with the MO "ContentDelete" and a *visible* page heading are likely to imitate bugs related to missing content. On the other hand, the dynamic features capture the importance of a given web element based on the frequency of its appearance and its extent of usage during test execution. While the static features we define are inspired from AST based features for program statements [154], dynamic features are similar to ranking based on execution traces in source code mutation [100]. Our spread score (*Sp*), inspired from the spread-random mutant selection strategy which selects only one mutant per source code statement, decreases the candidate score instead of filtering them out.

We compute Static (*St*) and dynamic (*Dn*) scores as a sum of all corresponding feature values, where values for a static feature (boolean) is 1 if true or 0 otherwise. Finally, the probability of selecting a mutant (Equation 4.1) is then computed as the ratio of a candidate score to the total score of all candidates.

4.5.4 Mutating Dynamic DOM and Mutation Score

For each mutation candidate ($C =< O, \omega >$), the Mutation Engine (*ME*) resets the web app and runs the test suite while applying the mutation operator (*O*) to all the

Table 4.3: Experimental Subjects

Subject			***Test Suite***		
	Version	*Loc*	*Cases*	*Loc*	*Loc*
AddressBook	8.0.0	16298	27	49	1325
Claroline	1.11.10	352537	40	46	1822
Collabtive	3.1	264642	40	48	1935
MantisBT	1.1.8	141607	41	43	1748
MRBS	1.4.9	34486	22	51	1114
PPMA	0.6.0	575976	23	54	1232
Total		866995	196	47	9176

concrete instances (ε) of the logical web element (ω).

Existing techniques on mutation analysis for web applications mutate the source code and compare the output DOM to determine the mutation kill score [120, 135] where there are observable changes. This often involves manual analysis of source code [100] to determine equivalent mutants as well. In our analysis, the mutant is considered to be *killed* by the test suite if any of the test cases fail either because of a failing test action or a test oracle.

4.6 Evaluation

To assess the efficacy of our mutation testing approach, we answer the following research questions.

RQ1 How efficient is MAEWU in generating non-equivalent mutants?

RQ2 How useful are the generated mutants for improving end-to-end UI test suites?

4.6.1 Experimental Setup

We use six open-source web apps as our subject systems, each with a manually written JUnit Selenium UI test suite used in previous web testing research [22]. Table 4.3 lists the name, version and size of our subjects and the corresponding test suite characteristics. All our experiments were run on a Red Hat Enterprise Linux Server (RHEL-7) and Chrome-v84 web browser.

4.6.2 Competing Techniques

We found two existing mutation analysis tools for web app UI testing – WebMuJava and AjaxMutator. WebMuJava is developed by Praphamontripong et al. [120, 121] for JSP and Java Server based web apps. It is not publicly available. Nishiura et al. [105] developed AjaxMutator to mutate Ajax and DOM API calls used in client-side JavaScript of web apps. We explain the reason for not including AjaxMutator below.

Issues using AjaxMutator

AjaxMutator uses the Rhino JavaScript parser to extract mutation targets for four specific features of client-side JavaScript – event registration; timer; Ajax calls; and DOM API to *append* and assigning *attribute*. The current implementation for AjaxMutator takes a *single* JavaScript file as input and generates mutants. It then runs a given Selenium Test Suite for *all* the generated mutants to compute the mutation score. However, we were unable to use it on our subjects.

The first issue we faced is regarding the input JavaScript file expected by AjaxMutator. All of our test subjects, which are modern web apps contained JavaScript in multiple files and libraries along with "InlineHTML" within other program files such as PHP, JSP. In order to get AjaxMutator to work on our subjects, we wrote a file parser to extract JavaScript from source files. However, Rhino could not parse these extracted JavaScript files, with sizes exceeding 50K lines for four of our subjects where upon we spent considerable amount of time trying to clean the files manually without success.

Secondly, for the two subjects we could generate mutants, we found no clear mechanism to reliably apply the generated mutants into the web app when we use this extracted JavaScript file. Because of this limitation, we could not even assess the resulting impact of these mutants on the actual functionality of the web app.

4.6.3 Procedure and Metrics

For each of our subjects, we configure MAEWU with the URL, the accompanying test suite, and a maximum limit of 50 mutants to be selected. Once MAEWU generates the mutants and computes the mutation score, we manually analyzed the impact on the behaviour of the web app for each of these selected mutants.

Analyzing mutants

In order to verify the impact of the mutation on the web page, we compare the live mutated state to the original state first in terms of visual appearance, and second, by exercising the functionality offered by target web element. If required, we also analyze the client-side JavaScript to understand the impact of the mutation.

We classify the mutants that impact neither the functionality nor the appearance of the web page to be equivalent. We then manually label the generated mutants to assess their 1) perceived bug severity, and 2) mutant stubbornness to determine their quality.[111]

The mutation score, bug severity score and stubbornness are computed only for non equivalent mutants.

Computing Mutation Score

We compute the mutation score as the percentage of killed non-equivalent mutants to the total number of non-equivalent mutants for each of the subject apps.

Computing Bug Severity

Based on the mutant impact on the UI, we compute a bug severity score shown in Table 4.4 using 18 manually labelled boolean features adapted from previous work [47]. In the interest of space, we skip discussing the actual adaption which is available along with the full decision tree to compute severity in our tool repository [170].

Computing Mutant Stubbornness

Existing work defines stubbornness through either, source code features [39] that make certain mutants difficult to kill or, by their relationship to the test suites [56,

Table 4.4: Bug Severity based on User Perception

Description of Severity	**Severity Score**
I did not notice any fault	0
I noticed a fault, but I would return to this website again	1
I noticed a fault, but I would probably return to this website again	2
I noticed a fault, and I would not return to this website again	3
I noticed a fault, and I would file a complaint	4

Table 4.5: Mutant Stubbornness

Required UI Testing effort	**Stubbornness Score**
No action needed	0
Locate the element in the page	1
Perform action on the element	2
Assert content, attribute or CSS property of the element	2
Perform action and Assert a property in the resulting browser state	3
Perform action and navigate to a different page to assert the effect of the action	4

177] such as the number of tests that can kill the mutant. The difficulty in finding the right program input [39] to kill the mutant is the common theme in categorizing mutants to be stubborn.

In this work, we model our stubbornness score based on the amount of effort required for a UI tester to kill the mutant. The stubbornness score lies between 0 and 4 as shown in Table 4.5. On the one extreme are easy-to-kill mutants that cause test failures by virtue of just reaching the mutated browser state. For example, it the

Table 4.6: Mutant Generation per subject

	addressbook	*claroline*	*collabtive*	*mrbs*	*mantisbt*	*ppma*	*total*
# candidates	13K	72K	329K	92K	61K	21K	586K
# selected	50	50	50	50	50	50	300
Lwe size	45	37	74	69	45	69	56
Static Score	2.82	3.1	2.88	2.98	3.12	3.1	3
Dynamic Score	63	44	77	73	65	79	67
# Web Elements	2.2K	1.8K	3.6K	3.4K	2.2K	3.4K	17K

Table 4.7: Mutants Generated by MAEWU

	# Mutants	*Non-Eq*	*Equiv*	*% Non-Eq*	*% Equiv*	*# Killed*	*Mutation Score*
addressbook	50	48	2	96	4	14	28
claroline	50	46	4	92	8	6	12
collabtive	50	43	7	86	14	6	12
mantisbt	50	48	2	96	4	6	12
mrbs	50	47	3	94	6	9	18
ppma	50	46	4	92	8	15	30
Total	300	278	22	93	7	56	19

mutation results in a blank page. On the other end of the extreme are the mutants that can result in back-end changes that infect other browser states while keeping the mutated state unaffected in terms of appearance or functionality. For example, a wrong transformation of user input in the infected state that is saved and retrieved from a database in another page.

Next, we discuss the results of our analyses and to save space, we will use the abbreviated operator names for the rest of the chapter as defined in Table 4.2.

4.6.4 Results

RQ1 - Efficacy

Table 4.6 shows the total number of candidates extracted by the Mutant Generator, and the characteristics of the 50 selected candidates per subject. On an average, the logical web element (ω) size for each selected candidate is 56, with *collabtive* having the highest repetition of concrete web elements at 78. Overall, nearly 17K concrete web elements were selected to be mutated in order to apply these 300 mutations in the browser at runtime. The average static score which determines the quality of mutant based on the web element and operator characteristics is 3, whereas the average dynamic score is 67. Note that all the selected logical web elements were *covered* by the test suites either directly by performing action on them during test execution or indirectly by causing a JavaScript to access these elements in the browser as recorded by our ObserverScript.

The results of our manual classification of these mutants based on their impact on UI functionality and appearance is shown in Table 4.7. On an average, 93% of the mutants generated by MAEWU were found to be non-equivalent. As shown in Table 4.8, SCM operator is responsible for 7 out of the 22 equivalent mutants that we found in total. In each of these 7 equivalent mutants, we found that the mutation applied by SCM using the css property "color" has been overridden by a css rule for the child elements. AAM generated 5 equivalent mutants because the added attributes like "id" were not used for any of the JavaScript in the page, rendering that attribute addition meaningless.

The most interesting and hard to classify equivalent mutants were created by EHAM and EHDM operators which generated 3 and 2 equivalent mutants respectively. In our implementation, we used the JavaScript "element.addEventListener" API to add and replace event listeners. We provide a random event handler function that is recorded by the Trace Generator for the subject. However, we found two reasons for EHAM having no impact on app functionality. First, because the functions being called within the new event listener were not available in the browser state being mutated, and therefore they just fail silently with no change to functionality in case of EHAM. Second, our mutation script could not override the default behaviour of the elements as defined by the browser. In future, we plan to automatically detect when such default behaviour impacts JavaScript manipulation of event listeners and select candidates accordingly.

The rest of the operators had a close to or equal to 100% success rate in generating non equivalent mutants.

Table 4.8: Mutant quality Per Operator

	AAM	*ADM*	*AMM*	*EHAM*	*EHDM*	*EHMM*	*SCM*	*SPM*	*SSM*	*SVM*	*TDM*	*TIM*	*TMM*	*CDM*	*CIM*	*CMM*	*Total*
#Mutants	9	10	13	15	4	4	29	26	37	37	27	30	17	17	6	19	300
#Non-Eq	4	10	12	12	2	4	22	26	36	35	27	30	17	17	6	18	278
#Killed	0	2	5	0	0	0	0	7	6	8	13	1	6	3	0	5	56
Non-Eq %	44	100	92	80	50	100	76	100	97	95	100	100	100	100	100	95	93
Mutation Score	0.00	0.20	0.42	0.00	0.00	0.00	0.00	0.27	0.17	0.23	0.48	0.03	0.35	0.18	0.00	0.28	0.20
Bug Severity	2.0	2.3	2.5	2.5	2.5	2.5	0.7	1.3	0.8	2.2	2.4	1.8	1.4	2.0	1.8	2.0	1.7
Stubbornness	2.5	2.8	3.1	3.0	3.0	2.8	2.0	2.0	2.0	1.9	1.5	2.0	2.1	2.0	2.0	2.1	2.1

RQ2 - Usefulness

Table 4.8 shows the results of our manual analysis of mutants as well as the mutation scores for the test suites for each of our mutation operators.

Our manual analysis revealed that all three event handlers MOs have a predictably high bug severity score because they are designed to break the behaviour of the web elements which is perceived to be the most severe fault in a web page. While maintaining a high severity score, EHAM, EHDM and EHMM also created mutants with the highest stubbornness score because killing these mutants required both performing action as well as verifying the result of the action. A mutation score of 0 for the UI test suites being evaluated is also indicative of the difficulty in killing these mutants.

Interestingly, ADM and AMM also have a high bug severity score similar to event handler MOs because they are also capable of affecting the behaviour. For example, deletion of the "href" and "input.type" attribute was particularly effective in breaking the element behaviour. However, the test suites were able to kill these mutants more frequently with 42% of AMM mutants being killed because they impact common interactive web elements that are often used by the test suites.

TDM operator has also generated mutants with a high bug severity primarily because it causes the pages to lose information and functionality by deleting parts of the page. However, these mutants are very easy to kill with a stubbornness score of only 1.5. Often, to kill the mutant, it is enough for a test script to try and locate the web element

Indeed, other MOs that can cause loss or change of information – SVM, CDM, CIM and CMM – have a high bug severity score and generate stubborn mutants because a tester has to either perform an action or assert the expected property of the target web element in order to kill the mutant. Interestingly, however, TMM which also can cause information loss through DOM hierarchy manipulation often caused only cosmetic defects, and therefore has a lower severity score.

In terms of usefulness, SCM generated the mutants with worst severity of 0.7 while generating majority of the equivalent mutants in our entire experiment. SSM and SPM have a similarly low severity score overall but generated mutants that impacted the behaviour of the page by blocking or reducing the accessibility of content or functionality of other web elements.

Overall, we found that the operators that are able to generate mutants of high bug severity and high stubbornness are most useful in exposing the weaknesses of the UI test suites. On an average, MAEWU generated mutants with a severity of 1.7 and a stubbornness score of 2.1, while exposing the limitations of existing test suites for our subjects which have a low mutation score of 0.20.

4.6.5 Discussion

Web App Dynamism and Test Fragility

In RQ2, we used two factors, bug severity and stubbornness score, in understanding if a mutant can be useful in improving existing UI test suites. However, an important aspect of modern web apps that we did not take into consideration in the current work is the presence of dynamic data. For example, content or element properties like 'id' that are dynamically generated and should not be considered to be bugs. So, generating mutants that may be similar to these dynamic changes to the web page will not be useful in determining the fault revealing capabilities of the test suites. Indeed, such mutants may deter practitioners from employing the framework, as these mutants are similar to equivalent mutants.

However, one of the biggest challenges of maintaining an end-to-end UI test suite for web apps is the fragility of web element locators [74], which require costly manual analysis and test maintenance. An interesting idea would be to use the MOs designed in this work, and select mutants that can reveal fragile test locators and test oracles.

Bug Severity for modern web apps

The existing study on the bug severity [47] used in our evaluation was primarily based on older web apps with limited dynamism. The study classifies any css related problem to be cosmetic in nature and gives a very low severity score. However, modern web apps rely heavily on fluid layout models in order to make the web app functionality accessible on multiple device and display configurations. While SCM, SSM, SPM generated mutants of very low severity on a fixed display configuration, these can be valuable in validating layout features and revealing layout bugs. Therefore, we believe a bridging study to better model the appearance related bugs that impact modern web apps is needed to determine bug severity for such mutants.

4.7 Related Work

For mutation analysis of web apps for assessment of UI test suites, there are only two existing research papers. Praphamontripong et al. [120, 121] define and implement mutation operators for JSP and Java Server based web applications in a tool called webMuJava which extends general Java based mutation operators. Nishiura et al. [105] defined mutation operators specific to client-side JavaScript of web applications used for DOM manipulations. To overcome the limitations of existing work owing to their usage of source code mutation, we propose to mutate dynamic DOM to mutate GUI functionality. Maezawa et al. [83] validates ajax code using three delay based mutation operators.

In addition, for web testing in general, Shahriar et al. [135] defined 11 mutation operators on PHP and JavaScript source code to find bugs related to cross-site scripting. Walsh et al. [161] implement CSS mutation operators that change the CSS rules related to the size (e.g., width) of web elements in order to simulate cross-browser page rendering faults. Mirshokraie et al. [100] developed mutation operators specific to JavaScript in web apps in addition to generic JavaScript operators. We do not consider either of these tools to be competing techniques to our work because they are neither intended to assess UI test suites, nor necessarily applicable to all web apps.

In broader area of mutation analysis for GUI applications, Alegroth et al [7], apply mutations to desktop Java application to evaluate GUI testing approaches. Oliveira et al. [106] developed scripts to automate mutant generation for seven of the 18 mutation operators defined in [7] to show that GUI mutation operators are better than traditional method level Java mutation operators in seeding GUI faults in applications. Linares-Vásquez et al., [77, 102] created a taxonomy of Android bugs with the purpose of defining source-code mutation operators for Android apps. Deng et al. [41, 42] defined mutation operators to change core components of

Android apps (e.g., intents, event handlers, XML files and activity lifecycle). Additionally, Luna et al. [81] presented Edroid, a tool that uses 10 mutation operators oriented to validate changes in the GUI.

In both the fields, namely desktop GUI and mobile testing, source code mutation has been employed to a good effect because of the homogeneity of the programs under test.

4.8 Threats to validity

External validity threats concern the generalization of our findings since we used a limited number of subject apps and analyzed only 300 mutants overall. We have chosen six subject apps used in previous web testing research, pertaining to different domains, and fully randomized the mutants to be analyzed in order to mitigate the threat. Threats to *internal validity* come from the manual labelling of mutant categories and features, which was unavoidable because no automated method could provide us with the required ground truth. The manual bug analysis, mutant analysis and labelling was performed by the first author, and the methodology was developed together by the two authors by analyzing example bugs, mutants independently and establishing a discussion to resolve conflicts. For bug severity analysis, we used a labelling methodology outlined in prior work to mitigate the threat to validity. For reproducibility of our findings, we made our tool publicly available [170] along with usage instructions and used subject systems.

4.9 Conclusion and Future Work

Despite the significance of UI test suites in validating web app functionality, currently, no mutation analysis tool exists for ascertaining their fault-finding capabilities. Existing tools for web app mutation testing rely on source code mutation and therefore cannot be universally applied because of the heterogeneous web app development ecosystem. In this work we developed MAEWU, an extensible mutation analysis framework for web apps which mutates the dynamic DOM during test execution. Given only the web app URL and a UI test suite to assess, MAEWU is able to automatically extract unique web elements in the web app and generate non-equivalent mutants that imitate UI manifestation of real web app bugs, select useful mutants based on the web element characteristics and perform mutation analysis to reveal the limitations of the test suite. As part of the future work, we plan to improve the mutant selection strategy by incorporating human feedback to compute mutant score.

Chapter 5

Carving UI Tests to Generate API Tests and API Specifications

5.1 Introduction

Software applications routinely use web APIs for establishing client-server communication. In particular, they increasingly rely on web APIs that follow the REST (REpresentational State Transfer) architectural style [53] and are referred to as RESTful or REST APIs. A typical REST API call starts with an HTTP request made by the client, e.g., the front-end of a web application running in the browser, and ends with a response sent by the server or the back-end of the application. To help clients understand the operations available in a service and the request and response structure, REST APIs are often described using a specification language, such as OpenAPI [109], API Blueprint [12], and RAML [123].

Web application testing is typically performed at multiple levels, each employing different techniques/tools and with different end goals. Unit-level testing of client- and server-side components focuses on validating the low-level algorithmic and implementation details and achieving high code coverage. In contrast, UI-level testing (also called end-to-end testing) focuses on covering navigation flows from the application's web UI, exercising various tiers of the application in end-to-end manner. In between unit and UI testing, API testing places the focus of testing on the operations of a service as well as sequences of operations; it exercises the server-side flows more comprehensively than unit testing but without going through the UI layer. API-level testing is guided by code-coverage goals as well as API-coverage goals (e.g., [91]).

For web applications that use RESTful APIs whose specifications are available, a number of automated testing techniques and tools (e.g., [16, 17, 35, 55, 61, 68, 72, 78, 92, 133, 143, 160, 163]) could be leveraged for API-level testing. These tools take as input an API specification, and automatically generate test cases for exercising API endpoints defined in the specification. However, in practical scenarios, using these tools may not always be possible.

First, for applications that do not have RESTful APIs, such tools are inapplicable. This rules out large classes of web applications, such as Java Enterprise Edition as well as legacy web applications, which could benefit just as well from automated API-level testing. Second, for web applications with RESTful APIs, API specifications may not be available. This can occur because of different reasons, often because the APIs are meant for use by the specific web application only or applications within an enterprise, and not exposed for invocation by external clients. Thus, formal API documentation is considered less important and not done due to development pressures or other factors. Moreover, even when API specifications are available, they can be obsolete and inconsistent with API implementations [90]. As a web application and its APIs evolve, the specifications—which can be large and complex—often fail to co-evolve due to the maintenance effort involved.[1]

In this work, we address the challenges of enabling automated API-level test generation universally for web applications, irrespective of whether they use RESTful web services, and automatically inferring OpenAPI specifications for web applications that use RESTful APIs. We present a dynamic technique that executes the web application via its UI to automatically create (1) API-level test cases that invoke the application's APIs directly, and (2) a specification describing the application's APIs that can be leveraged for development and testing purposes.

Although prior work has investigated carving unit-level tests from system-level

[1]Although there exist tools for automatically documenting REST APIs (e.g., SpringFox [142] and SpringDoc [141]), which can reduce the cost of keeping API specifications up-to-date with API implementations, their applicability is limited (e.g., to web applications implemented using Spring Boot [140]).

executions using code-instrumentation techniques (e.g., [50, 67, 165]), no technique exists for carving API-level test cases from UI paths or test cases.

Our technique monitors the network traffic between the browser and the server, while navigating the application's UI, and records the observed HTTP requests and responses. Then, it applies filtering to exclude the requests (and their responses) that are considered unnecessary for API testing. Next, it builds an API graph from the filtered requests and analyzes the graph to infer a specification that captures API endpoints (or resource paths), the applicable HTTP methods (e.g., GET, POST), and the request/response structure for each API operation (combination of HTTP method and API endpoint). A key feature of our technique is that it infers path parameters or variables for API endpoints from concrete endpoint instances observed during the navigation of UI paths. Moreover, it uses a novel algorithm for directed API probing and API graph expansion to discover more concrete endpoint instances that would otherwise be missed by UI path navigation alone.

The generated API specification can serve as documentation for server-side APIs of a web application (even if the APIs are not RESTful) and also be fed as input into an existing API testing tool for automated test generation (e.g. [16, 17, 61, 92, 160]) or used for checking inconsistencies in existing API specifications (for RESTful APIs).

The "carved" API test cases are derived from UI paths. Because these tests bypass the UI layer, they execute much more efficiently and are less prone to brittleness than UI-level tests. Yet, they cover the same server-side code as the UI paths from which they are derived and exercise the APIs in ways that they would be invoked from the UI.

We implemented our technique in a tool called APICARV that takes as input a UI test suite (generated or manually written) and carves API test cases and an OpenAPI specification.

We conducted an empirical study on seven open-source web applications to evaluate the technique's effectiveness in carving API tests and inferring OpenAPI specifications. With respect to test carving, our results are two-fold. First, they quantify the expected benefits of carved API tests: the tests attain similar coverage as the UI test paths from which they are derived, but at a fraction of the execution cost of UI tests: on average, more than 10x reduction in test execution time. Second, our results illustrate that carved API tests can increase the coverage achieved by two automated API test generators, EvoMaster [16, 51] and Schemathesis [62, 131]: on average, 52% (99%) gains in statement (branch) coverage for EvoMaster and 29% (75%) statement (branch) coverage gains for Schemathesis. Finally, for OpenAPI specification inference, our results show that the technique computes API endpoints (or resource paths) with 98% precision and 56% recall against the ground truth of existing API specifications. These results demonstrate the benefits of our technique.

The contributions of this work are:

- A first-of-its-kind approach for carving API test cases from UI paths that enables API-level testing for web applications, irrespective of the frameworks they use.
- A novel technique for inferring API specifications for web applications that use RESTful services.
- An implementation of the techniques in a tool called APICARV that is publicly available [171].
- Empirical assessment of APICARV, demonstrating the tool's effectiveness and the benefits of carved tests.

5.2 Background and Motivating Example

REST APIs [53] are typically described in a specification (e.g., in OpenAPI [109] format, previously known as Swagger) that lists the available service operations, the input and output data structures for each operation, and the possible response codes. Listing 5.1 shows a snippet of the OpenAPI spec for the REST API of a web application called `realworld` [54] (one of the applications used in our evaluation). The spec lists path items (under `paths:`), where a *path item* consists of a resource path (also referred to as "API endpoint") together with one or more HTTP methods (or "Operations"). The path item illustrated in Listing 5.1 shows the resource path (line 6), the HTTP method (line 7), the parameters specification (lines 8–13), and the response specification (lines 14–20).

Listing 5.1: Example OpenAPI specification.

```
info:
  title: Conduit API
servers:f
  - url: http://localhost:3000/api
paths:
/articles/{id}:            /* path item */
    get:                   /* HTTP method */
      parameters:
        - name: id
          in: path
          required: true
          schema: Integer
          example: 2
      responses:
        200:
          description: OK
          content:
            application/json:                              /* MIME */
              schema:
                ref: '/schemas/SingleArticleResponse'
```

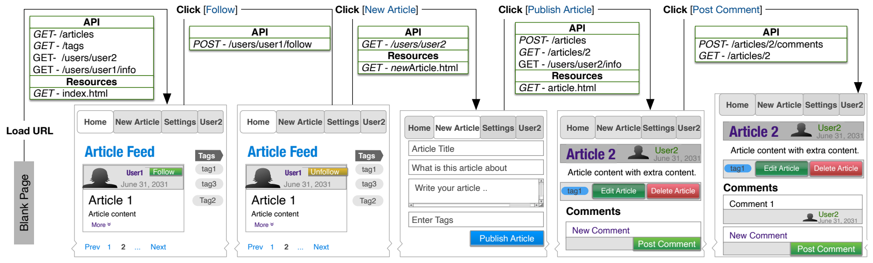


Figure 5.1: Example illustrating a sequence of UI actions and states along with the API calls that are triggered by UI events.

The response specification lists the status code (line 15) and the response data format (line 18) and the structure (lines 19–20). The structure definition contains a reference to a schema defined elsewhere in the document (omitted here).

The resource path `/articles/{id}` (line 6) is specified as a URI template [59], with path parameter `id`. Such a resource path describes a range of concrete URIs via parameter expansion. A concrete URI instance in an HTTP request targeting that endpoint contains an integer value for `id` (e.g., `/articles/2`). More generally, a path item can contain four kinds of parameters—path, query, cookie, and header. Path and query parameters are related to the URI, whereas header and cookie parameters are associated with HTTP request headers.

Figure 5.1 shows a UI test path for the `realworld` [54] web application. The test performs five UI actions that navigate through different application states. Each action exercises a specific functionality. For example, "`Click[follow]`" invokes the functionality to *follow* a given user. The figure also shows the server-side APIs invoked by the browser for each UI action. The UI states are then updated based on the server response. For instance, "`Click[follow]`" invokes the API "`POST[/users/user1/follow]`" and the UI state is updated, to show that "follow user" succeeded.

From the perspective of functional testing of the server-side APIs of a web application, the UI actions and the API calls invoke the same functionality and, therefore, would have the same code coverage and fault-detection abilities. However, invoking the APIs directly, instead of going through the UI layer has advantages: API calls exercise the service-side functionality much more efficiently and are less prone to the brittleness usually associated with UI tests [58], while exercising the APIs in the manner they are invoked from the UI. Thus, carved API tests can be convenient for developers to use in the course of their development activities. This is not to say that carved API tests are an alternative to, or replacement for, the UI tests. UI testing has an important role to play in covering end-to-end flows through all the application tiers; however, such testing is more suitable for system-level or acceptance testing in practice, and less so for supporting developers in their server-side development activities. Our first goal in this work, therefore, is to enable API-level testing such that it is universally applicable for all web applications irrespective of the web frameworks they use.

The second goal of our work is to infer an API specification, such as the one illustrated in Listing 5.1, automatically for the server-side APIs of a web application. The inferred specification documents the APIs and can also be used as input to automated API testing tools (e.g., [16, 17, 61, 92, 160]). API specification inference is applicable to web applications that implement RESTful APIs. Although API specifications could also be inferred for other types of web applications and could serve as useful documentation of server-side APIs, they would be less effective as inputs to automated API testing tools.

The core challenge in specification inference is how to compute resource paths with path parameters accurately (e.g., the `{id}` component of resource path

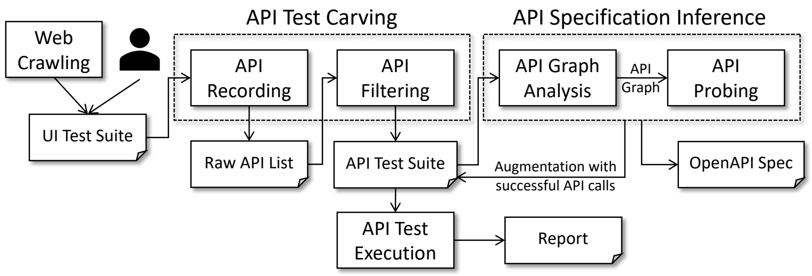


Figure 5.2: Overview of our technique APICARV.

`/articles/{id}`). The concrete URI instances in the requests observed at runtime contain integer values for `id`, such as `/articles/2`. The technique has to determine which segments of concrete URIs represent path parameters. Moreover, a resource path can have multiple path parameters, which adds to the complexity of the problem. In the next section, we present a dynamic-analysis-based carving technique for addressing these challenges.

5.3 Approach

Figure 5.2 presents an overview of our technique called APICARV. The input is a suite of UI test cases for a web application—the test cases could be automatically generated (e.g., created via automated web crawling) or implemented by developers. The output consists of an API-level test suite, along with a test-execution report, and for applications that use RESTful APIs, an OpenAPI specification describing the server-side APIs of the web application. The API test suite is composed of carved test cases that are augmented with API calls made during specification inference. The test-carving phase of the technique involves API recording and API filtering. The specification-inference phase constructs an API graph from the API test suite, and analyzes the graph to create an OpenAPI specification. A key step during specification inference is *API probing*, which attempts to expand the set of resource paths observed during UI test execution and discover additional information for creating more accurate specifications as well as augmenting the carved test suite. Next, we describe the two phases of the technique in detail.

5.3.1 API Test Carving

APICARV performs API test carving in two steps. In the first step, API recording, the technique monitors API calls that are triggered through the execution of the UI test suite and logs the raw API calls. To record API calls, we add network listeners to the browser executing the UI tests, which capture the raw outgoing and incoming HTTP traffic. As Figure 5.1 illustrates, a UI action can result in multiple API calls being executed, e.g., `Click[Publish Article]` triggers one POST and three GET requests. These requests, together with their corresponding responses, are logged during API recording.

In the second step, API filtering, the technique applies a series of filters—operation filter, status filter, and MIME filter—to the raw API calls to remove the calls that are irrelevant for API test and specification carving. The *operation filter* is based on HTTP method checking and is designed to omit methods that are unrelated to resource manipulation. This filter removes all calls with HTTP methods TRACE and CONNECT. The *status filter* checks the response status codes and excludes calls with unsuccessful requests, indicated by 4xx and 5xx response codes. Finally, the *MIME filter* checks the MIME type of the response payload and retains only those calls whose response payloads contain JSON or XML data (i.e., MIME types `text/json` or `text/xml`). For example, the resource-related API calls shown in Figure 5.1, which are irrelevant for API-level testing, are removed during filtering.

In the implementation of our technique, the filtering step is configurable. The three filters described here proved to be adequate for our experimentation. However, the user can configure filtering to prevent omission of certain requests considered essential for API testing or provide custom filters to omit additional types of API calls not covered by the three filters. Filtering configuration may also be needed based on web application characteristics; e.g., the MIME filter would be relevant if the application consists of RESTful APIs.

The output of the filtering step consists of sequences of API calls from which the carved API test suite is created.

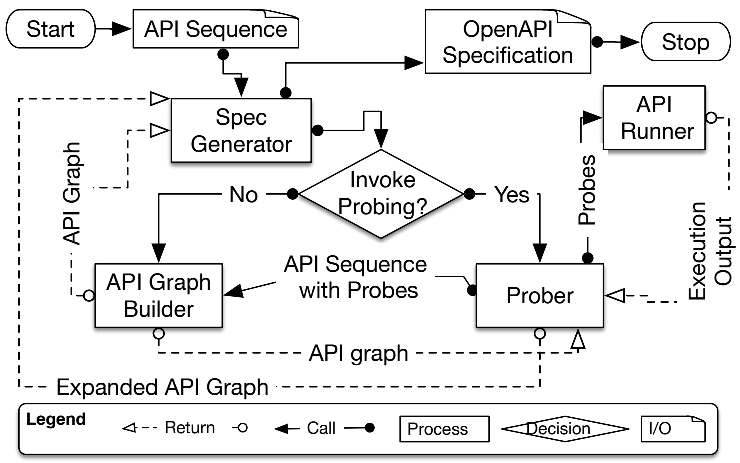


Figure 5.3: The specification-inference flow (`InferSpec`).

5.3.2 API Specification Inference

As discussed in 5.2, the core problem in specification inference that our approach addresses is computing path parameters for resource paths. The technique has to detect the path components of concrete URI instances that represent parameters, while handling paths with multiple parameters and compensating for server-side state changes as a result of UI actions that can potentially impact server responses for URIs.

Figure 5.3 presents a flow chart, `InferSpec`, illustrating specification inference. `InferSpec` takes as input the API sequences created by API carving and produces as output an API specification. It builds an API graph to represent the discovered resource paths and analyzes the graph to create the API specification. `InferSpec` also analyzes the graph to generate API probes (i.e., concrete API requests) that are executed against the application to discover additional valid API calls that are missing in the initial set of API sequences.[2] This step is intended to address the incompleteness of the initial API sequences and, thereby, improve the accuracy of the inferred API specification. As an additional benefit, the successful probes can be used for augmenting the carved API test suite, potentially increasing

[2]In the implementation of the technique, API probing can be limited by upper bound on exploration time or number of probes executed.

Algorithm 3: API graph construction

```
Function BuildAPIGraph:
    Input: apiset ← [$\mathcal{A}_1 ... \mathcal{A}_n$]                    /* Set of API Calls */
    Init: $\mathcal{G} \leftarrow \phi$          /* API Graph for the given set of API calls */
    foreach $\mathcal{A} \in apiset$ do
        $path \leftarrow \mathcal{A}.\mathcal{R}q$.URL.path
        SegArray ← $path$.split()         /* split path into segments */
        parent ← $root$                /* a dummy starting node */
        foreach $Seg_i \in SegArray$ do
            parentPath ← join($Seg_0, \ldots, Seg_{i-1}$)
            $v \leftarrow \phi$                      /* path parameter inference later */
            if $i$ = $SegArray.size$ then
                end-point ← True
            else
                end-point ← False
            end
            pathSeg ← ($Seg_i$, $i$, parentPath, SegArray.size, endpoint, $v$)
            segExists ← for-all $v_s \in \mathcal{N}$ AreEqual(pathSeg, $v_s$)
            if $segExists$ then
                $\mathcal{G}$.addEdge(parentNode, $v_{sim}$)
                parentNode = $v_{sim}$
            else
                $\mathcal{G}$.addNode(pathSeg)
                $\mathcal{G}$.addEdge(parentNode, pathSeg)
                parentNode = pathSeg
            end
        end
    end
    return $\mathcal{G}$
end
Function AreEqual ($v_1$, $v_2$):
    Output: True or False
    if $(v_1.n \neq v_2.n) \vee (v_1.d \neq v_2.d)$ then
        return False             /* different name or path index */
    end
    if $v_1.p = v_2.p$ then
        return True      /* same name, path index, and parent path */
    end
    if $IsEndPoint(v_1) \wedge IsEndPoint(v_2)$ then
        return CompareResponses($v_1.l$, $v_2.l$)
    else
        return False              /* different parent path */
    end
end
```

its coverage.

API Graph Construction

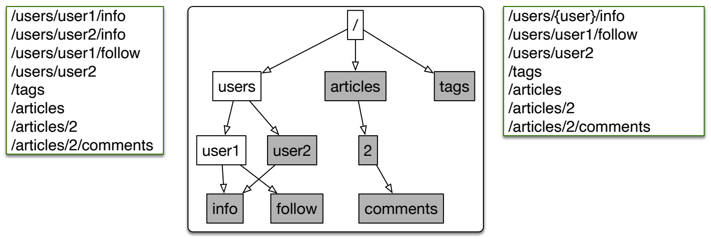
Algorithm 3 presents the algorithm for building the API graph. Before describing the algorithm, we introduce some terminology.

Definition 23 (**Path Segment**)**.** Given a URI U, a *path segment* ν is a tuple (n, d, p, e, l, v), where n is the segment string, d is the index of the segment in U, p is the parent path for ν, e is a boolean indicating whether ν is the final segment of U (and, therefore, an API endpoint), l is the response payload (if e is true), and v is the path parameter inference result for ν.

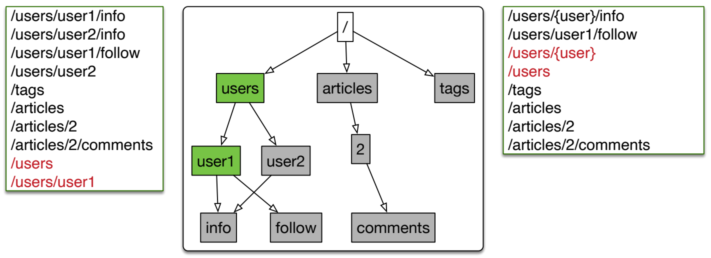
Definition 24 (**API Graph**)**.** An *API graph* $\mathscr{G} = (\nu_r, \mathscr{N}, \mathscr{E})$ is a directed acyclic graph, where ν_r is the dummy root node of the graph, $\mathscr{N}$ is a set of nodes, and $\mathscr{E}$ is a set of edges. Each node in $\mathscr{N}$ is a path segment and a pair of consecutive segments in a URI is connected by an edge in $\mathscr{E}$.

Definition 25 (**Graph Path**)**.** A *graph path* in an API Graph is a sequence of path segments $(\nu_r, \ldots, \nu_x)$ that connects the root node ν_r to any graph node ν_x. A complete path is a path from ν_r to a node where $\nu.e$ is true.

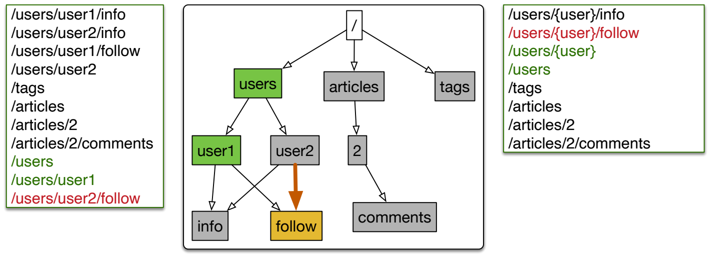
An API graph is constructed for a set of URIs (from API calls). For example, each API graph illustrated in Figure 5.4 represents the URIs shown to the left of the graph. Each graph node, except the root node, represents one or more segments from the URIs, and each complete path represents a URI.



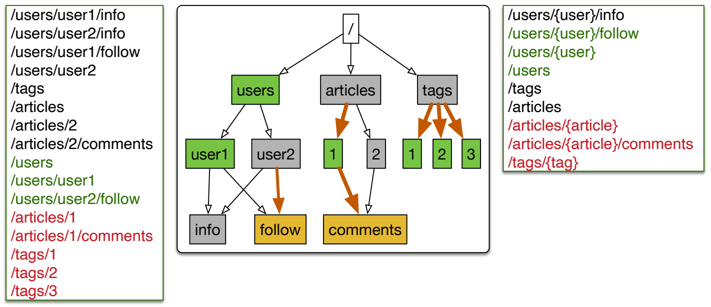
(a) API calls carved from the UI execution



(b) Probing Stage 1: probes for intermediate nodes



(c) Probing Stage 2: probes from bipartite analysis



(d) Probing Stage 3: probes from response analysis

Each subfigure shows the API graph (middle) built from API calls (left) and the inferred specification (right). The color coding of API calls (spec paths) indicates calls recorded (paths created) during UI navigation (black) and probes (paths) created in the previous stage (green) and the current stage (red). The color coding of nodes illustrates nodes created after UI navigation (gray), nodes with responses discovered via probing (green), and nodes with extra responses because of probing (yellow). The edge colors highlight edges created by probes in the previous stage (green) and the current stage (red).

Figure 5.4: Illustration of specification inference.

An API graph is constructed for a set of URIs (extracted from API calls). To illustrate, consider the API graphs shown in Figure 5.4. Each API graph represents the URIs shown to the left of the graph. A graph node, except the root node, represents one or more segments from the URIs, and each complete path represents a URI.

Function `BuildAPIGraph` (lines 1–27) of Algorithm 3 iterates over a given set of API calls ($\mathcal{A}$) and builds an API graph by parsing each request URI into a path in the graph. For each URI, the algorithm splits the URI into segments and then builds a path segment for each segment (lines 7–14). If a path segment v is not similar to any of the existing path segments in the graph, v is added to the graph (lines 15–23).

The similarity of two path segments is determined by the function `AreEqual` (lines 28–40), which first compares the names and path indexes of the two segments (line 29). If either of these do not match, the path segments are considered to be different. For example, as shown in Figure 5.4, each string segment in a URI has its own node in the graph. If the names and path indexes match, the function next compares the parent paths of the segments (using string comparison) and considers the segments to be equivalent if the parent paths match (lines 32–34). Otherwise, if the segments represent endpoints, the function `CompareResponses` is called to determine segment equivalence (lines 35–36). Note that a response object is available for a path segment only if there exists an API call that ends at the segment, making it an endpoint.

`CompareResponses` (not shown in Algorithm 3) relies on the structural similarity of responses instead of matching the entire responses. For a response with JSON or XML data, it ignores the *values* and builds a tree with *keys* in the data. It then asserts the structural similarity of the trees to determine response similarity. For example, consider the requests [`GET /users/user1/info`] and [`GET /users/user2/info`] in Figure 5.4) with responses {`"id":1, "name":"user1", "role":"user"`} and {`"id":2, "name":"user2", "role":"user"`}. To compare these two responses, the technique builds trees using the keys `[id, name, role]` and invokes a tree-comparison technique (APTED [113]) to check their equivalence.

Algorithm 4: Generating API specification from API graph

```
1  Function ExtractOpenAPI:
       Input: $\mathscr{G}$                /* API Graph after path variable inference */
       Output: uriTemplates $\leftarrow \phi$
2      $\mathscr{G} \leftarrow$ MergeLeafNodes($\mathscr{G}$)
3      foreach do $v \in \mathscr{G}$
4          if $v_i.e$ = True then
5              paths $\leftarrow$ GetGraphPaths($v_i$)
6              if paths.size ¿ 1 then
7                  template $\leftarrow$ getURITemplate(paths)
8              else
9                  template $\leftarrow$ paths[0]
10             end
11             uriTemplates.add(template)
12     end
13     return uriTemplates
14 end
15 Function MergeLeafNodes:
       Input: $\mathscr{G}$                /* API Graph for the given set of API calls */
16     foreach $v_i \in \{v_n\}$ do
17         foreach $v_j \in \{v_n\}$ do
               /* Assign a variable when nodes have matching responses
                   */
18             if ($v_i$.e=True) $\wedge$ ($v_j$.e=True) $\wedge$ (CompareResponses($v_i$, $v_j$)=True) then
19                 $v_i.v = v_j.v$ = variableMap.get()
20         end
21     end
22     return $\mathscr{G}$
23 end
24 Function GetGraphPaths:
       Input: $\mathscr{G}$, $v_x$                        /* An API Graph and a node in it */
       Output: paths $\leftarrow \phi$
25     foreach $\zeta_i \in \mathscr{G}$.getPathsto($v_x$) do
26         path $\leftarrow \phi$
27         foreach $v_i \in \{v_n\}$ do
28             if $v_i.v$ != $\phi$ then
29                 path.add($v_i$.name)
30             else
31                 path.add($v_i.v$)        /* v is set by MergeLeafNodes */
32             end
33         end
34         paths.add(path)
35     end
36     return paths
37 end
```

API Specification Generation

Algorithm 4 presents the steps involved in generating API specification from the API graph. Our goal in specification inference is to make the specification precise in terms of the number of path items for each API endpoint. An ideal specification should have exactly one path item describing an API endpoint; URI templates with path parameters make it possible to do so.

The algorithm first merges leaf nodes in the API graph (line 2). The function `MergeLeafNodes` (lines 15–23) performs response comparison to determine whether two nodes with different names belong to the same API endpoint. In addition, the algorithm makes use of the graph structure by getting paths that reach the same endpoint node in the API graph (lines 24–37). Finally, after computing a list of URIs that belong to similar endpoint nodes in the API graph, the technique performs a simple path-index-based match per URI segment to extract a template (lines 5–11).

For example, in Figure 5.4a, for the leaf node `info`, `GetGraphPaths` returns two graph paths for URIs `/users/user1/info` and `/users/user2/info`, which the technique considers equivalent and extracts the template `/users/{user}/info` with path parameter `{user}`.

For the example in Figure 5.4b, the graph paths for URIs `/users/user1` and `/users/user2` end at different segments. However, `MergeLeafNodes` performs response comparison to determine that these segments are equivalent and sets a variable to represent the path parameter (line 19 of Algorithm 4). Then, `GetGraphPaths` uses that variable and returns the string `/users/{user}` for both URIs. Finally, that returned string is used as the template with path parameter `{user}`. We use variable name `user` here for readability; our implementation creates variable names such as `var0`.

API Graph Expansion via Probing

The API graph created from the set of API calls seen during UI navigation is limited by the completeness of UI tests, which can affect the precision and completeness of the inferred API specification. To address this, APICARV expands the initial API graph via systematic API probing.

The technique creates four types of probes: intermediate, bipartite, response, and operation. Intermediate and bipartite probes are built via API graph analysis, response probes are based on HTTP response analysis, and operation probes aim to discover unseen operations for known API endpoints. After building probes,

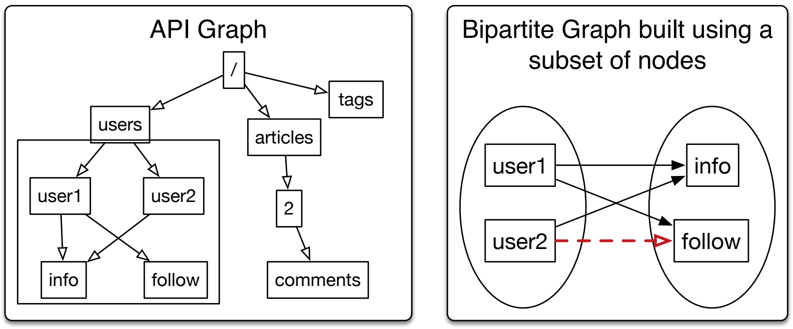


Figure 5.5: Bipartite analysis for API probe generation.

the technique sends them to the server using a scheduling algorithm that avoids data dependencies. The successful probes (i.e., probes with response codes other than 4xx or 5xx) are used for enhancing the API graph and augmenting the API test suite. `InferSpec` uses the expanded API graph to generate a potentially more accurate API specification.

Intermediate probes These probes are created for API graph nodes that do not have an associated server response (i.e. $v.e$ is false). For example, in Figure 5.4b, the probes `/users` and `/users/user1` are intermediate probes built for the nodes `users` and `user1` that do not have an associated response. In this case, the two endpoints are indeed valid. As a result, `InferSpec` adds a new path item `/users` and computes path variable `user` for the existing path `/users/user2`, which it replaces with the template `/users/{user}`.

Bipartite probes These probes are generated by building a bipartite graph from join nodes (i.e., nodes that have more than one predecessor) in the API graph. To illustrate, consider the example in Figure 5.5, where join node `info` has two predecessors (`user1` and `user2`). For this node, the technique constructs the bipartite graph shown in the figure: the left part of the graph contains all predecessors of the join node and the right part contains all successors of nodes in the left part. The technique then computes missing edges that would make the bipartite graph complete, i.e., each node on the left is connected to each node on the right. In this example, one missing edge makes the bipartite graph complete. From this analysis, probe `/users/user2/follow` is generated. As shown in Figure 5.4c, this new probe lets the technique infer the new path variable `user` and convert concrete resource path `/users/user1/follow` to path template `/users/{user}/follow`, thereby improving the specification.

Response probes Response probes are generated by analyzing the server responses for existing API calls. For each response object, the technique builds probes from keys and values extracted from the response. Suppose the response for `GET /tags` is `[{ id: 1, name: tag1, author: user1}, { id: 2, name: tag2, author: user1} ..]`. Using this response, we build probes such as `/tags/1, /tags/id, /tags/tag2, /tags/author`. As shown in Figure 5.4d, by analyzing the `articles` object, we build probes `/articles/1/comments` and `/articles/1`, which result in inference of path templates `/articles/{article}` and `/articles/{article}/comments`. Similarly, response analysis of `tags` object helps us infer `/tags/{tag}`.

Operation probes Operation probes are generated by analyzing API calls based on coverage of HTTP methods per known API endpoint. We consider seven HTTP operations—GET, POST, PUT, PATCH, OPTIONS, HEAD, and DELETE—in our analysis. For example, if the existing set of API calls contains `GET /tags/1` and `PATCH /tags/1`, we generate five probes for the endpoint, each covering one of the remaining HTTP operations (e.g., `DELETE /tags/1`).

Probe Scheduling HTTP requests can cause server-side state updates and, in general, the server response for a request can vary based on other requests. Moreover, the resources corresponding to a URI could be dynamic and only available in certain server-side states. Therefore, API probing should be performed at appropriate server states; our technique achieves this via probe scheduling, based on API graph analysis.

For each probe, we first check if the URI has a corresponding endpoint node in the graph. Consider the example in Figure 5.6. Node `user1`, which is the endpoint

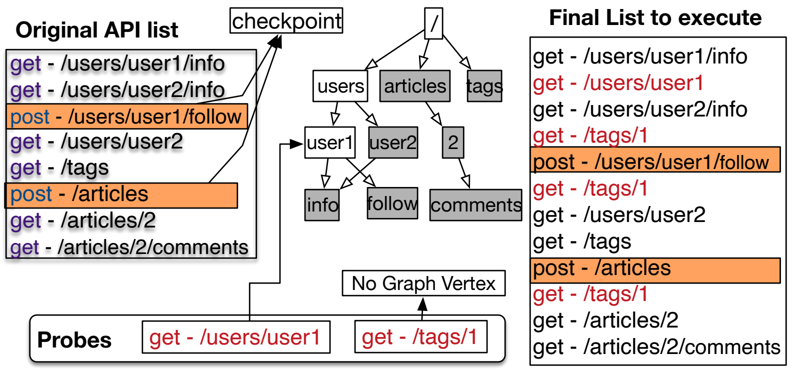


Figure 5.6: Example for illustrating probe scheduling.

node for URI `/users/user1`, already exists in the graph, whereas the corresponding node for `/tags/1` does not exist. If the endpoint node for a probe exists, we schedule the probe immediately before the last request in the original list whose URI includes the segment/node. If the endpoint node does not exist, we schedule the probe after each checkpoint request. A *checkpoint request* is an HTTP request that can change the server-side state. We define two types of checkpoints: cookie-based and operation-based. Cookie-based checkpoints are the HTTP requests for which the server responds with a `set-cookie` field. Operation-based checkpoints correspond to HTTP requests capable of modifying resources, i.e., HTTP methods PUT, POST, DELETE, and PATCH. In Figure 5.6, the two POST requests are checkpoint requests. As the final list in Figure 5.6 shows, `GET /users/user1` is scheduled only once, immediately after the node is discovered in the graph, whereas `GET /tags/1` is scheduled three times (corresponding to three potential server-side states), once before any checkpoint and once each after the two checkpoints in the original list. After the probes are executed, we keep only one instance of a successful probe in cases where multiple instances succeed.

5.4 Implementation

We implemented our technique in a tool called APICARV. We use Crawljax [96] to generate [172] UI test cases automatically. The API recorder module uses the Chrome Devtools Protocol [57] along with Selenium [134] to instrument the

Table 5.1: Web applications used in the evaluation.

Application	Framework	# API Endpoints	# Operations	LOC
booker	Spring-boot, ReactJS	15	24	8K
ecomm	Spring-boot	21	22	6K
jawa	Spring-boot, AngularJS	5	8	20K
medical	Spring-boot, VueJS	20	28	5K
parabank	Spring-mvc, AngularJS	27	27	60K
petclinic	Spring-boot, AngularJS	17	36	39K
realworld	Express, NextJS	12	19	12K
Total		117	164	150K

browser during UI test execution to record API calls. Our implementation and experimental dataset are publicly available in a replication package [171].

5.5 Empirical Evaluation

We investigated the following research questions in the evaluation of APICARV.

RQ$_1$: How do carved API tests compare with UI tests in terms of code coverage and execution efficiency?

RQ$_2$: How effective is APICARV in generating OpenAPI specifications?

RQ$_3$: Do carved API tests improve the coverage achieved by automatically generated API test suites?

5.5.1 Experiment Setup

We performed the evaluation on seven open-source web applications; Table 5.1 lists the applications and their characteristics. All of the applications implement RESTful APIs for their services and have OpenAPI specifications available, which serve as ground truth for measuring the accuracy of the inferred API specifications.

Table 5.2: Statistics about different analysis stages in APICARV runs on the subject applications.

	API Filtering		Probing				Generated Test Suites							
							Carver				Carver + Prober			
	Recorded	Filtered	Generated	Executed	Succeeded	Checkpoints	Paths total	Paths success	Requests	Time (s)	Paths total	Paths success	Requests	Time (s)
booker	3610	613	529	85295	57	203	14	10	443	21	24	20	500	21
ecomm	7838	1187	440	4906	4	18	61	60	1175	21	61	60	1179	15
jawa	1568	198	60	279	13	7	9	4	110	6	18	13	123	6
medical	315	122	1277	39309	17	32	24	23	117	15	26	25	134	17
parabank	13072	574	694	43833	26	68	25	23	572	125	29	27	598	124
petclinic	1536	294	1399	5144	102	42	20	20	290	5	50	50	392	7
realworld	1471	398	7510	72259	225	9	62	36	365	79	116	91	590	103
Total	29410	3386	11909	251025	444	379	215	176	3072	273	324	286	3516	294

For UI test generation, we configured Crawljax to run for 30 minutes. We also created 14 manual tests for three subjects (`booker`, `medical`, `ecomm`), which required dedicated action sequences and input data. Thus, our evaluation uses automatically generated and developer-written UI test cases.

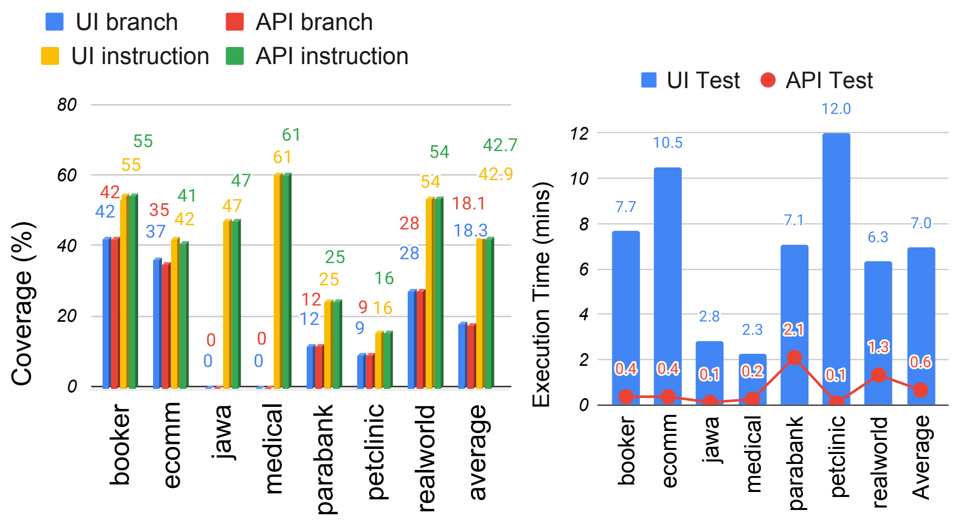
For investigating RQ$_3$, we used two popular automated test generators for REST APIs—EvoMaster [51] and Schemathesis [131]. EvoMaster can be used in white-box and black-box modes; in the white-box mode, it is applicable to REST APIs implemented in the Java language. For our study, we used EvoMaster in its black-box mode so that it can be applied to non-Java API implementations in our subjects. A recent empirical study [70] showed these two tools to be the top-performing tools, in terms of code coverage achieved, among the black-box testing tools for REST APIs. We configured EvoMaster to run for one hour; for Schemathesis, we used its default configuration settings. We ran each tool 10 times to account for randomness and report coverage data averaged over the 10 runs. To measure code coverage, we used JaCoCo [66] for Java-based APIs and Istanbul [11] for JavaScript-based APIs.

5.5.2 Quantitative Analysis of APICARV Stages

Before discussing our results on the research questions, we present empirical data on different stages of APICARV and provide a quantitative analysis of the stages. Table 5.2 presents data about the filtering, probing, and test-generation stages.

Columns 2–3 of the table show the number of API calls available after recording and filtering, and highlight the importance of filtering: i.e., a large proportion of the raw API calls recorded get filtered out. These calls basically retrieve resources related to UI rendering in the browser and can be ignored for testing the functionality of server-side APIs. On average, over 88% of the raw API calls belong to the category of irrelevant calls. The proportion of such calls ranges from over 72% (for `realworld`) to over 95% (for `parabank`). Thus, API filtering is an important component of APICARV; moreover, as discussed in Section 5.3.1, the filtering component can be configured to be more strict (removing more of the raw API calls) or less stringent (removing fewer calls).

Columns 4–7 present information about the probing stage: probes generated,



(a) Code coverage **(b)** Execution time

Figure 5.7: Coverage rates and execution times of UI tests and carved API tests.

probes executed, and checkpoints in the filtered API list. On average, the number of probes generated is over three times the number of filtered API calls, and the number of probes executed is 21 times the number of generated probes. Recall from the discussion of probe scheduling in Section 5.3.2 that some probes are scheduled for multiple executions based on occurrences of checkpoints in the initial API list. The number of generated probes varies considerably, ranging from 0.3 times the initial API calls (for `jawa`) to almost 19 times the initial API calls (for `realworld`). A similar large variation can also be seen in the number of probes executed. Finally, 444 probes were successful and are added to API test suite generated at the end of probing.

Columns 8–15 of Table 5.2 present data about test generation, broken down by tests created during carving and probing. It can be seen that the total number of successful paths, which are the number of valid resource paths discovered, increases from 176 in the Carver test suite to 286 in the Prober test suite. The Prober is, thus, able to discover 110 additional valid resource paths across the applications. These 110 path invocations come at the cost of only 21 seconds. In other words, the Prober test suite is able to successfully exercise 62.5% more paths with only 7.6% increase in test-execution time.

5.5.3 RQ$_1$: Coverage Rates and Execution Efficiency of Tests

Figure 5.7a presents coverage rates for UI tests and carved API tests. As the data illustrate, the coverage is identical for all applications, except `ecomm`, for which instruction and branch coverage of API test cases are marginally lower (by 1% and 2% respectively). We suspect that this difference may have been due to API filtering. On average, carved API test suites covered 18.1% branches and 42.7% instructions, which is 0.2% less than the coverage achieved by the UI test suites. Thus, overall, the carved API tests perform very well in matching the coverage rates of UI test cases.

In terms of execution efficiency, however, there is a big difference between the two types of test cases, as Figure 5.7b shows. On average, the UI test suites took seven minutes to run, whereas the API test suites ran in about 0.6 minutes only—more than 10x improvement in execution efficiency. The biggest improvement occurs for `petclinic`, for which the UI test suite took 150 times longer to run than the API test suite. Even with the smallest improvement, which occurs for `parabank`, the API tests executed over 3x faster than the UI tests (2.1 minutes versus 7.1 minutes, respectively).

The carved API tests match the coverage achieved by UI tests while executing significantly (10x) faster than UI tests. Thus, carved API tests can be employed for improving test execution efficiency, without incurring loss in coverage of server-side code.

5.5.4 RQ$_2$: Accuracy of Inferred OpenAPI Specification

Goals and Measures. To measure the effectiveness of APICARV in inferring API specifications, we compute precision, recall, and F_1 scores for the generated OpenAPI specification (S_{gen}) against the existing OpenAPI specification, considered the ground truth (S_{gt}), for each subject. We compute these scores for resource paths and operations (HTTP methods) defined on resource paths, and for the specification generated from the API graphs computed after carving and probing. Precision and recall are computed in the usual way, based on true positives, false positives, and false negatives. A path/operation is considered true positive if it occurs in both

Table 5.3: Precision, recall, and F1 scores achieved for API specification inference.

	Path			Operation				
Tool	Pr	Re	F_1	Pr	Re	F_1	Pr*	F_1*
Carver	1.00	0.49	0.32	0.85	0.46	0.28	1.00	0.31
Carver+Prober	0.98	0.56	0.35	0.48	0.54	0.25	0.95	0.34

S_{gen} and S_{gt}, false positive if it occurs in S_{gen} but not in S_{gt}, and false negative if it occurs in S_{gt} but not in S_{gen}. F_1 score is the harmonic mean of precision and recall. In addition to these metrics, we measure *duplication factor* for S_{gen}. A duplication occurs when multiple paths/operations in S_{gen} correspond to the one path/operation in S_{gt}. We map paths in S_{gen} to paths in S_{gt}, and compute duplication factor as (# mapped paths in S_{gt} / # mapped paths in S_{gen}). The computed value ranges from 0 to 1, with higher values indicating less duplication (the value 1 means there is no repetition of API endpoints in S_{gen}). The presence of duplication causes S_{gen} to contain redundant paths/operations that can be combined.

Results and Analysis. Table 5.3 presents the precision, recall, and F1 scores for specification inference. In terms of resource paths, APICARV achieves 100% precision for specifications created after carving and 98% precision after probing (Column 2). The recall after the carving phase is 49%, which the probing phase improves to 56%—a gain of 14% (Column 3). The probing phase is intended to address incompleteness in the API calls observed during UI navigation; the result shows that it achieves that to some degree and with only a small reduction in precision. The overall recall at 56% is somewhat low, which is a consequence of the incompleteness inherent in dynamic analysis. This could be addressed via improvements in crawling or providing higher coverage UI test suites as input to APICARV. This concern is orthogonal to APICARV's core specification-inference and test-carving techniques.

In terms of operations, the results for recall (Column 6) after the carving phase is 46%, which the probing phase improves to 54%—a gain of 17%. The precision value for operations is high (85%) after carving, but there is a significant drop to 48% after probing (Column 5). Upon closer inspection, we found that this is caused by operation probes, specifically the probes with HTTP methods `OPTIONS` and `HEAD`; these requests are not handled correctly in any of the subjects. Ideally, an `OPTIONS` request should provide available operations for an API endpoint and the corresponding specification for the endpoint should contain `OPTIONS` as an operation. In all of our subjects, while the server returns a success status (200 code) for an `OPTIONS` request, the corresponding operation is not documented in the specification. We consider this to be a specification inconsistency with respect to application behavior; on ignoring this inconsistency, APICARV achieves 95% precision for operations as well, shown as Pr* in Table 5.3 (Column 8).

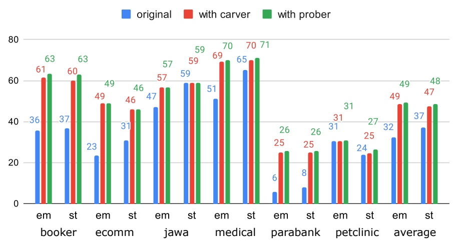
APICARV achieves high precision in inferring resource paths and operations. The probing phase of APICARV increases the recall and F1 scores, while not causing a significant reduction in precision.

A manual analysis revealed that the path and operation precision drops from 1.0 to 0.98 and 0.95 because of one API endpoint found through probing in `realworld`. We verified that the resource path is indeed valid and provides a health-check for the service despite being absent in the specification, a potential inconsistency. Table 5.4 shows the operation inconsistencies that we found per subject. The inconsistencies exposed by the carver are particularly interesting because these `OPTIONS` and `HEAD` requests are actually being used by the client—the application UI layer running in the browser—to communicate with the server. Recall that the carver uses only the requests captured during UI navigation. For example, the UI client of the `ecomm` application uses the `OPTIONS` operation on 11 API endpoints for server communication. These inconsistencies indicate room for potential improvements in the specifications, in particular, by documenting the `OPTIONS` HTTP method for API endpoints.

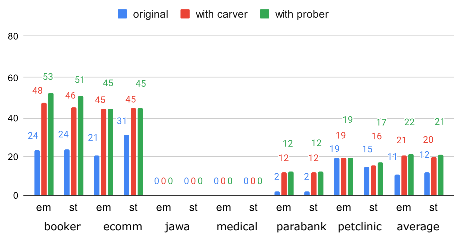
Columns 4–7 of Table 5.4 show the duplication factor computed for the specification generated after the carving and probing phases. It can be seen that the duplication factor does not vary significantly for six of the subjects. Path and operation duplication drops to 0.67 from 0.8 and 0.94, respectively, for `petclinic` because of the challenge in determining similarity of responses (Algorithm 4 lines

Table 5.4: Endpoints inferred, path/operation duplication found, and operation inconsistencies detected.

	Endpoints Inferred		Duplication				Operation Inconsistencies	
			Path		Operation			
	carver	car+pro	carver	car+pro	carver	car+pro	carver	car+pro
booker	8	10	0.89	0.91	0.92	0.94	0	21
ecomm	13	13	0.81	0.81	0.82	0.82	11	14
jawa	2	2	1.00	1.00	1.00	1.00	0	2
medical	15	16	0.88	0.89	0.89	0.90	9	17
parabank	9	9	1.00	1.00	1.00	1.00	0	12
petclinic	8	12	0.80	0.67	0.94	0.67	5	18
realworld	4	5	0.57	0.56	0.57	0.56	0	18
Average/Total	59	67	0.85	0.83	0.88	0.84	25	102



(a) Instruction coverage



(b) Branch coverage

Figure 5.8: Augmentation effectiveness of carved tests: coverage rates of test suites generated by EvoMaster (em) and Schemathesis (st) before augmentation (original) and after augmentation with carved tests and probes.

15–23). Recall that we use tree-based comparison to determine response similarity and, in the case of petclinic, the server provides responses that are structurally dissimilar based on the back-end data differences.

Endpoints covered (Columns 2–3 of Table 5.4) improves for four of the seven subjects, with the largest increase of 50% occurring for `petclinic`. Overall, endpoint coverage increases from 59 to 67, for 14% increase, which is reflected in the path recall values (Column 3 of Table 5.3) as well.

APICARV can detect potential inconsistencies between API implementations and specifications, and could be leveraged for improving specifications.

5.5.5 RQ$_3$: Augmentation Effectiveness of Carved API Tests

Goals and Measures. With RQ$_3$, we investigate the usefulness of carved API tests in enhancing the coverage rates achieved by EvoMaster [51] and Schemathesis [131]. Specifically, we measure instruction and branch coverage of the test suites generated by those tools; then, we augment the test suites in two steps, by adding the carved API tests and the successful probes to the test suites, and measuring coverage gains in each augmentation step.

Results and Analysis. Figure 5.8 presents the results for RQ$_3$. It shows the instruction and branch coverage rates for the original API test suites generated by EvoMaster and Schemathesis and the two augmented test suites. Overall, our augmentation causes coverage increases in most instances, with a few exceptions (e.g., there are no instruction coverage gains for `jawa` with Schemathesis). In terms of instructions, on average, the coverage of EvoMaster test suite increases from 32% to 49%, for a coverage gain of 52%; for Schemathesis, coverage increases from 37% to 48%, for a coverage gain of 29%. For branch coverage, augmentation has a bigger effect because of the low coverage rates of the original test suites. For EvoMaster, branch coverage gain is 99%, increasing from 11% to 22%. For Schemathesis, branch coverage gain is 75%, increasing from 12% to 21%. For both types of coverage, the gains for `booker`, `ecomm`, and `parabank` are substantial.

Moreover, additional coverage from probes, on top of the gains from carved tests, occurs in several instances. For example, the probes provide a considerable increase in branch coverage for `booker`—48% to 53% for EvoMaster and 46% to 51% for Schemathesis.

APICARV can significantly increase coverage achieved by EvoMaster and Schemathesis and, thus, can effectively complement such tools. The probing stage of APICARV can provide small additional gains on top of the gains from API tests carved from UI paths.

```
/visits/{visitId}:
  get:
    parameters:
      -name: visitId
       in: path
```

```
public ResponseEntity getVisit(Integer visitId) {
    Visit visit = this.clinicService.findVisitById(visitId);
    if (visit == null) {
        /* Covered by ApiCarv, Schemathesis and Evomaster */
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    /* Covered by ApiCarv through probing */
    return new ResponseEntity<>(visitMapper.toVisitDto(visit), ↩
        HttpStatus.OK);
}
```

Figure 5.9: Example of an endpoint and the associated service code (from `petclinic`) that requires specific test data.

```
/user/signUp:
  post:
    params: {in: body, schema: {"user": string, "pass": string}}
    responses: {200: {"status": string}}

/user/signIn:
  post:
    params: {in: body, schema: {"user": string, "pass": string}}
    responses: {200:{"status": string, "token": string}}

/cart/add:
  post:
    params: {name: token, in : query, schema : string}
```

Figure 5.10: Example (from the `ecomm` application) of dependencies between API endpoints.

To understand how carved API test suites can complement API testing techniques, we performed an in-depth analysis of the differences in code coverage achieved by the testing tools and APICARV. Our analysis revealed two interesting high-level scenarios where carving API test suites from end-to-end UI test suites could improve the overall effectiveness of API-level testing of web applications.

First, generating appropriate test data to cover API endpoints is a challenging aspect of API fuzzing, and carved API test suites can be leveraged to cover certain endpoints that require specific test data. An example of such a scenario from `petclinic` is shown in Figure 5.9 where covering the `/visits/visitId` endpoint requires providing a value for `visitId` that is already present in the application database. In this instance, APICARV leveraged the analysis of API calls observed during the carving phase (derived from the UI test suite) to find a value of `visitId` that covers line 8 of method `getVisit()` and elicits a successful response from the service (`HttpStatus.OK`); EvoMaster and Schemathesis were unable to craft a request with a valid value for `visitId`.

Second, some API endpoints can have dependencies on other API endpoints. Consider the three API endpoints from the `ecomm` application shown in Figure 5.10. The endpoint `/cart/add` requires a query parameter, `token`, that is generated by the server in response to a call to the `/user/signIn` endpoint. But, to invoke `/user/signIn`, a user must first be registered via the `/user/signUp` endpoint. EvoMaster could invoke `/user/signUp` successfully, but it could not execute the subsequent operations to sign in and add item to cart; Schemathesis could not cover any of these operations. In contrast, the API test suite carved from the UI test suite of `ecomm` could create a successful `POST /cart/add` request by satisfying these dependencies. The API calls in the carved test suite are constructed by the web application UI, which is developed to adhere to the API specification and valid API invocation sequences. Thus, such dependencies are inherently followed in sequences of API calls made along UI paths, and APICARV captures the call sequences by navigating those paths. The API testing tools, although they attempt to discover meaningful or valid API call sequences, could not generate requests that satisfy the endpoint dependencies in this instance.

Modern web applications are commonly tested in the industry using end-to-end UI test suites, which are often manually created. It is efficient and convenient to develop such UI test suites because of the mature UI testing eco-system and the fact that application business logic is easily translated into a sequence of UI actions on the web interface. API tests carved from such UI test suites can, therefore, contain realistic test data and encapsulate the application business logic in a sequence of API calls. Given these characteristics, carved test suites could effectively complement the API tests generated by API testing tools, such as EvoMaster and Schemathesis.

5.5.6 Threats to Validity

Our study may suffer external and internal threats to validity. In terms of external threats, we used seven web applications and two REST API testing tools. Our selection of web applications was constrained to applications that implement RESTful services and have OpenAPI specifications available to serve as ground truth; also, our requirement of measuring code coverage on APIs further constrained the candidate applications. Future evaluation with more and varied web applications will help confirm whether our results generalize. Our selection of REST API testing tools was guided by a recent study [70] that showed EvoMaster and Schemathesis to be the most effective tools, in terms of coverage achieved, among the studied black-box testing tools. Another threat is the use of existing OpenAPI specifications as ground truth. We think this is a reasonable choice for our experiments and we use them with the expectation that they may have some inconsistencies. As for internal threats, there may be bugs in APICARV and our data-collection scripts. We mitigated these threats by implementing automated unit test cases for APICARV and manually checking random samples of our results. We also make APICARV and our experiment artifacts available [171] to enable replication of our results.

5.6 Related Work

To the best of our knowledge, our work is the first to propose carving of API-level tests from UI-level test executions. Several papers have explored carving of unit-level tests from system-level executions via code instrumentation (e.g., [50, 67, 165]). Elbaum et al. [50] present a technique for carving unit-level tests from system tests consisting of Java-based code exercising the application end-to-end. Other techniques [67] selectively capture and replay events and interactions between selected program components and the rest of the application, using simplified state representations or they aim [165] at enhancing replay efficiency by mixing action-based and state-based checkpointing. These approaches highlight different advantages of carved tests compared to the original tests, such as their execution efficiency and robustness to program changes. Carved unit tests are shown [50] to be orders of magnitudes faster than the original executions, while retaining most of their fault-detection capabilities. These benefits also motivate our work, and our evaluation demonstrates the significantly superior execution efficiency of carved API tests compared with UI-level tests, with negligible loss in code coverage.

Dynamic specification mining has mostly focused on mining behavioral models of a program from its execution traces (e.g., [9, 79, 119]). These models capture relations between data values and component interactions, to allow for accurate analysis and verification of the software. More relevant to our work are the approaches recently suggested for mining OpenAPI specifications. Several works propose inferring OpenAPI specifications from web API documentation pages. AutoREST [28] infers API specifications from HTML-based documentation via selection of web pages that likely contain information relevant to the specification. It applies a set of rules to extract relevant information from the pages and construct the specification. D2Spec [176] uses machine-learning techniques to extract the base URL, path templates, and HTTP methods from crawled documentation pages. A different approach that uses dynamic information is taken in [49], which generates web API specifications from example request-response pairs. Closest work to ours is SpyREST [137], which intercepts HTTP requests and applies a simple heuristic for identifying path parameters, by considering numeric path items and using regular expression matching. In contrast, our approach infers path parameters via API-graph analysis and API probing. We tried to execute SpyREST for comparison with APICARV's specification inference, but its service failed to work.

5.7 Conclusion and Future Work

We presented APICARV, a first-of-its-kind technique and tool for carving API tests and specifications from UI tests. Our evaluation on seven open-source web applications showed that (1) carved API tests achieve similar coverage as the UI tests that they are created from, but with significantly less (10x) execution time, (2) APICARV achieves high precision in inferring API specifications, and (3) APICARV can increase the coverage achieved by automated API test generators.

There are several directions in which our approach could be extended in future work, including development of techniques for improving the inferred specifications to take them closer to developer-written specifications, enhancing the specifications with information (e.g., example values) that can be leveraged by automated REST API test generators, and improving the recall of specification inference via novel crawling techniques aimed at discovering the server-side APIs of a web application.

Chapter 6

Concluding Remarks

Testing modern web applications in practice requires significant human effort, and consumes the bulk of the application development resources. Currently, tools and techniques that enable and support automatic testing practices for other kinds of software are either unavailable or ineffective for web applications due to their heterogeneous nature. While the heterogeneous nature of web ecosystem has forced practitioners to rely on end-to-end UI testing, performing this task automatically has remained an unsolved challenge over the years due to the presence of near-duplicate web pages which are a result of highly dynamic UI. On one hand, a highly interact-able and dynamic UI enables greater web application functionality and offers a rich experience to the end users, but on the other-hand makes UI testing more challenging.

6.1 Contributions

In this dissertation, we introduced novel techniques for automatically testing modern web applications by relying on the analysis of the user interface, making them universally applicable to all web applications. The main contributions of this dissertation are as follows.

- An empirical study (Chapter 2) of existing state comparison techniques that are used for inferring navigational models of web applications, a key first step towards automatic UI test generation. We also performed a qualitative analysis of functional near-duplicate web pages in the wild, where we defined three categories of near-duplicate web pages characterized by the functional differences between the web pages being compared. The results of our empirical analysis showed that existing state comparison techniques are ineffective in detecting near-duplicate web pages and infer sub-optimal navigational models undermining dependent downstream tasks such as model-based test generation.

- A novel technique called FRAGGEN to automatically generate UI test suites for modern web applications (Chapter 3). FRAGGEN introduces a novel state abstraction for web pages based on page fragmentation and utilizes fine-grained fragment analysis to diversify state exploration and generate resilient test oracles. Our evaluation on eight open-source web apps showed that FRAGGEN in comparison to existing techniques detects near-duplicate web pages much more effectively, infers navigational models which cover greater web application functionality, and generates resilient UI test suites that are suitable for regression testing.

- A mutation analysis framework called MAEWU which is universally applicable for any web application (Chapter 4) . MAEWU relies only on the analysis of the application UI, mutates the dynamic DOM in the browser instead of the source code, and therefore determines UI test suite effectiveness for any web application irrespective of the front-end and back-end frameworks it is composed of. Our evaluation on six open-source web apps demonstrate that MAEWU is effective in generating non-equivalent mutants, able to dynamically apply mutations consistently during test execution, and ultimately effective in assessing Web UI test suites in terms of adequacy and facilitates test suite quality improvements.

- A first-of-its-kind technique and tool called APICARV that enables API testing universally for all web applications (Chapter 5) . APICARV takes a UI test suite as input, captures API calls that are executed during UI test execution, carves API test suites, and also generates API specifications for web applications. Our evaluation on seven open-source web apps showed that

APICARV was able to carve API test suites that replicate server-side coverage achieved by the UI test suites in significantly less execution time, generate precise API specifications, and improve overall code coverage achieved by existing tools through test augmentation.

6.2 Research Questions Revisited

We introduced four research questions in Chapter 1 and addressed them in the next four chapters (chapters 2-5). Below, we revisit them and summarize our contributions.

Research Question 1
What are functional near-duplicates and how to detect them?

Automated web testing techniques infer models from a given web app, which are used for test generation. From a testing viewpoint, such an inferred model should contain a minimal set of states that are distinct, yet, adequately cover the app's main functionalities. In practice, models inferred automatically are affected by near-duplicates, i.e., replicas of the same functional webpage differing only by small insignificant changes. In Chapter 2, we presented the first study of near-duplicate detection algorithms used for web app model inference. In our study, we first characterized functional near-duplicates in the wild by classifying a random sample of 1500 state-pairs, from 493*k* pairs of webpages obtained from over 6,000 websites, into three categories, namely clone, near-duplicate, and distinct. We then systematically computed thresholds that define the boundaries of these categories for each of the ten existing near-duplicate detection techniques we gathered from three domains, namely, information retrieval, web testing, and computer vision. We then use these thresholds to evaluate the techniques first on the 1500 state-pairs from the random sample, and then a further 97,500 state-pairs extracted from nine open-source web apps. Our study highlights the challenges posed in automatically inferring a model for any given web app. Our findings show that even with the best thresholds, no algorithm is able to accurately detect all functional near-duplicates. We then purpose these near-duplicate detection techniques to infer navigational models on our nine open-source web apps and found that lack of effectiveness in detecting near-duplicates undermines the overall effectiveness of model inference for all the existing techniques. As a result, the inferred models were found to be sub-optimal, which would render model-based testing techniques such as test generation ineffective.

Research Question 2

How to produce accurate web app models and effective regression test suites? The findings from the empirical study provided a strong motivation for us to design a technique that is 1) capable of detecting functional near-duplicates in an effective manner and 2) does not require threshold selection, which proved to be a costly manual fine-tuning process during our empirical study described in Chapter 2. We designed a model-based test generation technique, FRAGGEN, described in Chapter 3, which relies on the insight that a web page is not necessarily a single functional entity, but a set of functionalities. It is worth noting that all existing techniques rely on whole-page comparison by treating the web page as a singular entity. FRAGGEN eliminates the need for thresholds, by employing a novel state abstraction based on page fragmentation to establish state equivalence. FRAGGEN also uses fine-grained page fragment analysis to diversify state exploration and generate reliable test oracles. Our evaluation shows that FRAGGEN outperforms existing whole-page techniques by detecting more near-duplicates, inferring better web app models and generating test suites that are better suited for regression testing. On a dataset of 86,165 state-pairs, FRAGGEN detected 123% more near-duplicates on average compared to whole-page techniques. The crawl models inferred by FRAGGEN have 62% more precision and 70% more recall on average. FRAGGEN also generates reliable regression test suites with test actions that have nearly 100% success rate on the same version of the web app even if the execution environment is varied. The test oracles generated by FRAGGEN can detect 98.7% of the visible changes in web pages while being highly robust, making them suitable for regression testing.

Research Question 3

How to measure the effectiveness of regression test suites?

In Chapter 2 and Chapter 3, we focused on the problem of state-equivalence and model inference. Next, we address the challenge of assessing the adequacy of UI test suites in Chapter 4. When we studied the existing work related to this challenge, we found that there exists no reliable and universal method to ascertain the fault-finding capabilities for UI test suites of web apps. Mutation testing or mutation analysis, a well known fault-based testing technique is considered a reliable method to determine test adequacy. For other software, where mutation testing is practiced, existing methods generate mutants by making small changes to source code imitating programmer errors. However, for web apps, mutation testing based on source-code is difficult to employ universally because of the heterogeneous nature of web development ecosystem. Existing attempts to enable mutation testing for web apps employed source-code mutation and therefore were only applicable to specific frameworks. With the target of developing a universal mutation analysis framework for web apps, we developed MAEWU, described in Chapter 4, a mutation analysis framework that mutates the dynamic DOM in the browser instead of source code. As a result, MAEWU can be employed to determine test suite adequacy for any given web app regardless of the back-end and front-end components it might be composed of. We proposed 16 mutation operators that mutate the behavior and appearance of web elements to mimic the nine categories of web app faults found through an analysis of 250 bug reports. We evaluated our dynamic mutation analysis framework on six open-source web apps. The results from our empirical evaluation demonstrated that MAEWU is effective in assessing Web UI test suites in terms of adequacy and facilitates test suite quality improvements.

Research Question 4

How to enable API testing universally for all web apps?

While the previous research questions focus on automatic UI testing, through this research question, we aim to tackle the challenge of API testing in Chapter 5. API testing can play an important role, in-between unit-level and UI-level testing for modern web applications which make extensive use of API calls to update the UI state in response to user events or server-side changes. Existing API testing tools require API specifications (e.g., OpenAPI), which often may not be available or, when available, be inconsistent with the API implementation, thus limiting the applicability of automated API testing to web applications. In Chapter 5 we present an approach that leverages UI testing to enable API-level testing for web applications. Our technique navigates the web application under test and automatically generates an API-level test suite, along with an OpenAPI specification that describes the application's server-side APIs (for REST-based web applications). A key element of our solution is a dynamic approach for inferring API endpoints with path parameters via UI navigation and directed API probing. We evaluated the technique for its accuracy in inferring API specifications and the effectiveness of the "carved" API tests. Our results on seven open-source web applications show that the technique achieves 98% precision and 56% recall in inferring endpoints. The carved API tests, when added to test suites generated by two automated REST API testing tools, increase statement coverage by 24% and 29%, and branch coverage by 75% and 77%, on average. The main benefits of our technique are: (1) it enables API-level testing of web applications in cases where existing API testing tools are inapplicable and (2) it creates API-level test suites that cover server-side code efficiently, while exercising APIs as they would be invoked from an application's web UI, and that can augment existing API test suites.

6.3 Reflections and Future Directions

In this thesis, we advanced the research geared towards automatic UI test generation for modern web apps, specifically addressing the challenges in the effective detection of near-duplicates and the generation of optimal navigational models using state exploration. We also took the first steps towards enabling mutation analysis and API testing universally for all web apps regardless of the back-end and front-end frameworks they are developed upon. However, much work remains to be done in order to make fully automatic web application testing effective, efficient, and practicable.

State comparison. While our fragment-based state comparison technique comfortably outperforms existing techniques, further work is needed in order to be able to detect functional near-duplicates with 100% accuracy. Specifically, we use the dynamic DOM tree and visual analysis of the captured screenshot as a proxy for functional equivalency. However, in practice, determining functional equivalence of web pages requires a certain amount of domain knowledge of the web application and semantic understanding of the presented content in the web page, which human testers typically rely upon. Natural language processing (NLP) and deep learning (DL) provide a suitable avenue to explore the applicability of page semantics in determining the functional similarity of web pages, a relatively unexplored area of web testing. In this thesis and most of the existing research, so far, state comparison for functional testing of web applications has been viewed as a singular challenge that can be addressed with a single solution. However, there might be benefits in viewing tasks such as model inference and automatic test oracles as separate but related concerns that require different approaches. For example, test oracles are required to detect state changes that could be application bugs while state abstraction used during model inference is only concerned with establishing equivalence for the sake of mapping application functionality. In Chapter 3, we do this separation to an extent by introducing memoization of dynamic fragments which help us relate application changes to bugs with varying degrees of severity. However, we believe this problem warrants further research and employing deep learning to target specific scenarios such as regression testing would prove beneficial. In addition, our research has so far focused on functional changes that always have a corresponding change in the DOM. However, there are certain web elements like canvas elements that may contain visual changes but have no corresponding effect on the DOM. While existing research on testing canvas elements has remained separate from broader functional testing, there might be a benefit in incorporating such elements as a part of the state equivalence for model inference and test generation.

Mutation analysis. In our mutation analysis framework, we use dynamic mutation of DOM in the browser in order to apply mutations to the web application. As we show in Chapter 4, our technique was able to generate mutants that imitate real-world bugs. However, while mutation dynamic DOM would ultimately represent all changes that could occur due to application faults in the UI, whether developer-introduced bugs can cause such UI changes in specific web applications is unknown. Given the resource-intensive nature of mutation analysis, in MAEWU, we took first steps towards assigning importance to possible mutants in order to make mutation analysis efficient. However, future work should look towards improving mutant prioritization and semantic similarities between mutants from a functional perspective in order to make mutation analysis a productive endeavor in improving test suite quality. For example, back-end instrumentation to determine if two web page mutants could be similar because they arise out of similar back-end code changes could help diversify functionality assessed through mutation analysis.

API Testing of web applications. In our API carving work, as described in Chapter 5, we utilize existing UI test suites to generate API test suites and infer API specifications. The core observation that led to this project is the fact that end-to-end UI testing is often considered the de-facto method to validate web application functionality and is often part of acceptance testing for web applications. In the current form, APICARV only focuses on enabling API testing using existing UI test suites. Our experimental set-up in fact utilizes FRAGGEN to automatically crawl the web application to enable API testing. However, usage of automatic UI testing falls short and ends up only covering 50% of the available end-points in the web application. An interesting idea would be to use API coverage as a metric to improve crawling. Associating API end-points with UI actions can also lead to interesting possibilities related to web page semantics. In a way, API end-points represent back-end functionality and equivalence of UI elements or actions could be established through equivalence of API calls triggered by them. However, a single UI action can lead to several API calls and the challenge would be to form a semantic representation for the UI action through further analysis of the API calls and corresponding server responses. Further, our technique can be adapted to other platforms like mobile and desktop applications that have similar client-server architectures. It might be particularly interesting to adopt a similar approach in native mobile applications (especially android), where researchers have developed automatic UI testing techniques [87, 128, 148] that are able to achieve a significantly high code coverage and therefore likely to exercise almost all API endpoints accessible through the UI.

Bibliography

[1] Document Object Model (DOM). https://en.wikipedia.org/wiki/Document_Object_Model. → pages 3, 51, 53

[2] The average web page . https://www.advancedwebranking.com/html/. → page 102

[3] Stratified Random Classifier. https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html. Package: scikit-learn. → page 27

[4] HTTP Archive. https://httparchive.org/reports/page-weight. → page 102

[5] HTML Living Standard . https://html.spec.whatwg.org/. → page 95

[6] S. Afroz and R. Greenstadt. Phishzoo: Detecting phishing websites by looking at them. In *2011 IEEE Fifth International Conference on Semantic Computing*, pages 368–375, Sep. 2011. → pages 9, 16

[7] E. Alegroth, Z. Gao, R. Oliveira, and A. Memon. Conceptualization and evaluation of component-based testing unified with visual gui testing: An empirical study. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation*, ICST 2015, pages 1–10, 2015. → pages 79, 114

[8] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 252–261, March 2011. → page 39

[9] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. POPL 2002, pages 4–16. ACM, 2002. → page 147

[10] and A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004. ISSN 1057-7149. → pages 14, 16, 49

[11] Apache. https://istanbul.js.org/. https://istanbul.js.org/, 2022. Accessed: 2022-09-01. → page 137

[12] APIBlueprint. API Blueprint, 2022. URL https://apiblueprint.org/. Accessed: Sep 1, 2022. → page 117

[13] app1. Angular version of the Spring PetClinic web application. https://github.com/spring-petclinic/spring-petclinic-angular, 2021. → pages 21, 64

[14] app7. PHP Password Manager. https://github.com/pklink/ppma, 2021. → pages 21, 64

[15] app9. Mantis Bug Tracker. https://github.com/mantisbt/mantisbt, 2018. → pages 21, 64

[16] A. Arcuri. Restful api automated test case generation with evomaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(1): 1–37, 2019. → pages 118, 119, 120, 122

[17] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering*, ICSE 2019, pages 748–758, Montreal, QC, Canada, 2019. IEEE. → pages 118, 119, 122

[18] Y.-M. Baek and D.-H. Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 238?249, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338455. → page 39

[19] M. Bajammal and A. Mesbah. Web canvas testing through visual inference. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, ICST 2018, pages 193–203, 2018. → page 84

[20] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015. → page 60

[21] M. Biagiola, F. Ricca, and P. Tonella. Search based path and input data generation for web application testing. pages 18–32, 08 2017. ISBN 978-3-319-66298-5. → page 83

[22] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella. Web test dependency detection. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 12 pages. ACM, 2019. → pages 21, 38, 64, 104

[23] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella. Diversity-based web test generation. In *Proceedings of 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 12 pages. ACM, 2019. → pages 2, 5, 21, 38, 41, 64, 83

[24] L. Blanco, N. Dalvi, and A. Machanavajjhala. Highly efficient algorithms for structural clustering of large websites. In *Proceedings of the 20th International Conference on World Wide Web*, WWW 2011, pages 437–446. ACM, 2011. → page 9

[25] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, Sept. 1997. → page 38

[26] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma. Vips: a vision-based page segmentation algorithm. Technical Report MSR-TR-2003-79, November 2003. URL https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/. → pages 50, 51, 62, 102

[27] F. Calefato, F. Lanubile, and T. Mallardo. Function clone detection in web applications: A semiautomated approach. *J. Web Eng.*, 3(1):3–21, May 2004. → pages 38, 39

[28] H. Cao, J. Falleri, and X. Blanc. Automated generation of REST API specification from plain HTML documentation. In *ICSOC*, volume 10601 of *Lecture Notes in Computer Science*, pages 453–461. Springer, 2017. → page 147

[29] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of*

Computing, STOC 2002, pages 380–388, New York, NY, USA, 2002. ACM. ISBN 1-58113-495-9. → pages 13, 14, 15, 49

[30] T.-C. Chen, S. Dick, and J. Miller. Detecting visually similar web pages: Application to phishing detection. *ACM Trans. Internet Technol.*, 10(2): 5:1–5:38, June 2010. ISSN 1533-5399. → pages 9, 16

[31] S. R. Choudhary, D. Zhao, H. Versee, and A. Orso. Water: Web application test repair. In *Proceedings of 1st International Workshop on End-to-End Test Script Engineering*, ETSE 2011, pages 24–29. ACM, 2011. → page 2

[32] S. R. Choudhary, M. R. Prasad, and A. Orso. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proc. of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST 2012, pages 171–180. IEEE, 2012. → pages 49, 84

[33] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Proceedings of 30th International Conference on Software Maintenance and Evolution*, ICSME 2014. IEEE, 2014. → pages 2, 41

[34] Claroline. Claroline. Open Source Learning Management System. https://sourceforge.net/projects/claroline/, 2021. → pages 21, 64, 91

[35] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato. Automated black-box testing of nominal and error scenarios in restful apis. *Software Testing, Verification and Reliability*, page e1808, 2022. → page 118

[36] Crater. Crater. https://sourceforge.net/projects/crater.mirror/, 2021. → page 91

[37] V. Crescenzi, P. Merialdo, and P. Missier. Clustering web pages based on their structure. *Data Knowledge Engineering*, 54(3):279–299, Sept. 2005. → pages 38, 39

[38] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: A tool for testing web 2.0 applications. In *Proceedings of the Workshop on JavaScript Tools*, JSTools 2012, pages 11–15, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312745. → page 83

[39] X. Dang, X. Yao, D. Gong, and T. Tian. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. *IEEE Transactions on Reliability*, 69(1):334–348, 2020. → pages 106, 107

[40] C. Degott, N. P. Borges Jr., and A. Zeller. *Learning User Interface Element Interactions*, pages 296–306. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450362245. → page 84

[41] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2015, pages 1–10, 2015. → pages 87, 114

[42] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei. Mutation operators for testing android apps. *Inf. Softw. Technol.*, 81(C):154–168, Jan. 2017. ISSN 0950-5849. → page 114

[43] M. N. Dennis Fetterly, Mark Manasse. On the evolution of clusters of near-duplicate web pages. In *Journal of Web Engineering*, volume 2, pages 228–246. Institute of Electrical and Electronics Engineers, Inc., October 2004. → pages 4, 8, 9, 38, 39

[44] G. A. Di Lucca, M. Di Penta, A. R. Fasolino, and P. Granato. Clone analysis in the web era: an approach to identify cloned web pages. In *Proceedings of the International Workshop of Empirical Studies on Software Maintenance - November 2001 - Florence - Italy*, pages 107–113, 2001. → pages 4, 8, 9, 39, 49

[45] G. A. Di Lucca, M. D. Penta, and A. R. Fasolino. An approach to identify duplicated web pages. *2013 IEEE 37th Annual Computer Software and Applications Conference*, 00(undefined):481, 2002. → pages 38, 39

[46] dimeshift. DimeShift: easiest way to track your expenses. https://github.com/jeka-kiselyov/dimeshift, 2021. → pages 21, 64

[47] K. Dobolyi and W. Weimer. Modeling consumer-perceived web application fault severities for testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA 2010, pages 97–106, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588230. → pages 106, 113

[48] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax crawl: Making ajax applications searchable. In *Proceedings of the 2009 IEEE*

International Conference on Data Engineering, ICDE 2009, pages 78–89. IEEE, 2009. → pages 4, 8

[49] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot. Example-driven web api specification discovery. In A. Anjorin and H. Espinoza, editors, *Modelling Foundations and Applications*, pages 267–284. Springer International Publishing, 2017. → page 147

[50] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Transactions on Software Engineering (TSE)*, 35(1):29–45, 2009. → pages 119, 146

[51] evomaster. EvoMaster: A Tool For Automatically Generating System-Level Test Cases, 2022. URL https://github.com/EMResearch/EvoMaster. Accessed: Sep 1, 2022. → pages 120, 137, 143

[52] E. V. Eyk and W. J. V. Leeuwen. Performance of near-duplicate detection algorithms for crawljax. B.S. Thesis, 2014. → page 39

[53] R. T. Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000. → pages 6, 117, 120

[54] Gérôme Grignon, Manuel Vila. The mother of all demo apps. https://github.com/gothinkster/realworld, 2022. Accessed: 2022-01-01. → pages 120, 122

[55] P. Godefroid, B.-Y. Huang, and M. Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 725–736, 2020. → page 118

[56] L. Gonzalez-Hernandez, B. Lindström, J. Offutt, S. F. Andler, P. Potena, and M. Bohlin. Using mutant stubbornness to create minimal and prioritized test sets. In *2018 IEEE International Conference on Software Quality, Reliability and Security*, QRS 2018, pages 446–457, 2018. → page 106

[57] Google. Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/, 2022. Accessed: 2022-01-01. → page 134

[58] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *Proceedings of 31st International Conference on Software Engineering*, ICSE 2009, pages 408–418. IEEE, 2009. → pages 2, 41, 122

[59] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, , and D. Orchard. URI Template. RFC 6570, 2012. URL https://www.rfc-editor.org/info/rfc6570. → page 121

[60] Y. Guo and S. Sampath. Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM 2008, page 303?305, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939715. → pages 88, 90

[61] Z. Hatfield-Dodds and D. Dygalo. Deriving semantics-aware fuzzers from web api schemas, 2021. → pages 118, 119, 122

[62] Z. Hatfield-Dodds and D. Dygalo. Deriving semantics-aware fuzzers from web api schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings*, ICSE-companion 2022, pages 345–346, 2022. → page 120

[63] T. H. Haveliwala, A. Gionis, D. Klein, and P. Indyk. Evaluating strategies for similarity search on the web. In *Proceedings of the 11th International Conference on World Wide Web*, WWW 2002, pages 432–442, New York, NY, USA, 2002. ACM. ISBN 1-58113-449-5. → pages 4, 9

[64] M. Henzinger. Finding near-duplicate web pages: A large-scale evaluation of algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR 2006, pages 284–291. ACM, 2006. → pages 4, 8, 9, 15, 28, 38, 42

[65] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. → pages 5, 86

[66] JaCoCo. JaCoCo Java Code Coverage Library, 2022. URL https://www.eclemma.org/jacoco/. Accessed: Sep 1, 2022. → page 137

[67] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *Proceedings of the 23rd*

International Conference on Software Maintenance, ICSM 2007, pages 234–243. IEEE, 2007. → pages 119, 146

[68] S. Karlsson, A. Čaušević, and D. Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. In *13th International Conference on Software Testing, Validation and Verification*, ICST 2020, pages 131–141. IEEE, 2020. → page 118

[69] J. Kiesel, L. Meyer, F. Kneist, B. Stein, and M. Potthast. An empirical comparison of web page segmentation algorithms. In *ECIR (2)*, pages 62–74, 2021. → page 51

[70] M. Kim, Q. Xin, S. Sinha, and A. Orso. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pages 289–301. Association for Computing Machinery, 2022. → pages 137, 146

[71] Koel. Koel. https://sourceforge.net/projects/koel.mirror/, 2021. → page 91

[72] N. Laranjeiro, J. Agnelo, and J. Bernardino. A black box tool for robustness testing of rest services. *IEEE Access*, pages 24738–24754, 2021. → page 118

[73] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th International Conference on Software Testing, Verification and Validation*, ICST 2015, pages 1–10. IEEE, 2015. → page 42

[74] M. Leotta, A. Stocco, F. Ricca, and P. Tonella. Using multi-locators to increase the robustness of web test cases. In *Proceedings of 8th International Conference on Software Testing, Verification and Validation*, ICST 2015, pages 1–10. IEEE, 2015. → page 113

[75] V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966. → pages 14, 15, 49

[76] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764 – 766, 2013. ISSN 0022-1031. → page 26

[77] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 233–244, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. → pages 79, 114

[78] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao. Morest: Model-based restful api testing with execution feedback. *arXiv preprint arXiv:2204.12148*, 2022. → page 118

[79] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering*, ICSE 2008, pages 501–510. ACM, 2008. → page 147

[80] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, 1999. → pages 14, 16, 49

[81] E. Luna and O. E. Ariss. Edroid: A mutation tool for android apps. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, pages 99–108, 2018. → page 115

[82] MacOS Mojave. MacOS Mojave . https://en.wikipedia.org/wiki/MacOS_Mojave, 2020. → page 75

[83] Y. Maezawa, K. Nishiura, H. Washizaki, and S. Honiden. Validating ajax applications using a delay-based mutation technique. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE 2014, pages 491–502, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. → page 114

[84] S. Mahajan and W. G. Halfond. Finding HTML Presentation Failures Using Image Comparison Techniques. In *Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE 2014, pages 91–96. ACM, 2014. → pages 16, 49

[85] S. Mahajan and W. G. J. Halfond. Detection and Localization of HTML Presentation Failures Using Computer Vision-Based Techniques. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation*, ICST 2015, pages 1–10, April 2015. → page 49

[86] G. S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, WWW 2007, pages 141–150. ACM, 2007. → pages 4, 8, 9, 38

[87] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. doi:10.1145/2931037.2931054. URL https://doi.org/10.1145/2931037.2931054. → page 156

[88] A. Marchetto, F. Ricca, and P. Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *2007 9th IEEE International Workshop on Web Site Evolution*, pages 31–38, 2007. → page 90

[89] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *2008 1st International Conference on Software Testing, Verification, and Validation*, ICST 2008, pages 121–130, 2008. → pages 5, 83

[90] B. Marculescu, M. Zhang, and A. Arcuri. On the faults found in rest apis by automated test generation. *ACM Transactions on Software Engineering Methodology*, 31(3), 2022. → page 118

[91] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Test Coverage Criteria for RESTful Web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 15–21, 2019. → page 118

[92] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés. Restest: Black-box constraint-based testing of restful web apis. In *International Conference on Service-Oriented Computing*, pages 459–475. Springer, 2020. → pages 118, 119, 122

[93] A. M. Memon and M. L. Soffa. Regression testing of guis. *Proceedings of the 9th European software engineering conference*, pages 118–127, 2003. → pages 2, 41

[94] A. Mesbah. *Advances in Testing JavaScript-based Web Applications*, volume 97 of *Advances in Computers*, chapter 5, pages 201–235. Elsevier, 2015. → pages 3, 11

[95] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE 2009, pages 210–220. IEEE, 2009. ISBN 978-1-4244-3453-4. → pages 49, 84

[96] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3:1–3:30, 2012. → pages 4, 5, 8, 14, 15, 17, 41, 62, 83, 134

[97] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35–53, 2012. → pages 2, 5, 41, 42, 60

[98] A. Milani Fard and A. Mesbah. Feedback-directed exploration of web applications to derive test models. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE 2013, pages 278–287. IEEE, 2013. → pages 4, 8, 14, 15, 84

[99] S. Mirshokraie and A. Mesbah. Jsart: JavaScript assertion-based regression testing. In *International Conference on Web Engineering*, ICWE 2012, pages 238–252, 2012. → page 41

[100] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering (TSE)*, 41(5):429–444, May 2015. ISSN 2326-3881. → pages 79, 84, 103, 104, 114

[101] M. Mirzaaghaei and A. Mesbah. Dom-based test adequacy criteria for web applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 71–81, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. → page 93

[102] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk. Mdroid+: A mutation testing framework for android. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 33–36, 2018. → page 114

[103] mrbs. Meeting Room Booking System. https://mrbs.sourceforge.io/, 2021. → pages 21, 91

[104] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden. Mutation analysis for javascript web applications testing. volume 2013, 01 2013. → pages 79, 84

[105] K. Nishiura, Y. Maezawa, H. Washizaki, and S. Honiden. Mutation analysis for javascript web applications testing. volume 2013, 01 2013. → pages 105, 114

[106] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. Definition and evaluation of mutation operators for gui-level mutation analysis. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2015, pages 1–10, 2015. → page 114

[107] R. A. P. Oliveira, E. Alégroth, Z. Gao, and A. Memon. Definition and evaluation of mutation operators for gui-level mutation analysis. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, ICSTW 2015, pages 1–10, 2015. → pages 79, 87

[108] J. Oliver, C. Cheng, and Y. Chen. TLSH – A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13, Nov 2013. → pages 13, 14, 15, 49

[109] Open API Initiative. Openapi specification. https://spec.openapis.org/oas/latest.html, 2022. Accessed: 2022-01-01. → pages 117, 120

[110] pagekit. Pagekit: modular and lightweight CMS. . https://github.com/pagekit/pagekit, 2021. → pages 21, 64

[111] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE 2018, pages 537–548, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. → page 106

[112] M. Pawlik and N. Augsten. Efficient computation of the tree edit distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015. → pages 14, 15, 49, 64, 102

[113] M. Pawlik and N. Augsten. Tree edit distance: Robust and memory-efficient. *Inf. Syst.*, 56:157–173, 2016. → pages 53, 62, 129

[114] PGWeb. PGWEB. https://sourceforge.net/projects/pgweb.mirror, 2021. → page 91

[115] phoenix. Phoenix: Trello tribute done in Elixir, Phoenix Framework, React and Redux. https://github.com/bigardone/phoenix-trello, 2021. → pages 21, 64

[116] PHP AddressBook. Simple, web-based address & phone book. http://sourceforge.net/projects/php-addressbook, 2021. Accessed: 2018-10-01. → pages 21, 45, 64, 91

[117] PHP List. PHP List. https://sourceforge.net/projects/phplist/, 2021. → page 91

[118] T. Popela. Implementace algoritmu pro vizuální segmentaci www stránek. Master's thesis, Brno University of Technology, Faculty of Information Technology, 2012. URL https://www.fit.vut.cz/study/thesis/14163/. → page 62

[119] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *ICSM*, pages 1–10. IEEE Computer Society, 2010. → page 147

[120] U. Praphamontripong and J. Offutt. Applying mutation testing to web applications. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 132–141, April 2010. → pages 79, 84, 88, 104, 105, 114

[121] U. Praphamontripong, J. Offutt, L. Deng, and J. Gu. An experimental evaluation of web mutation operators. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW 2016, pages 102–111, 2016. → pages 84, 105, 114

[122] L. Ramaswamy, A. Iyengar, L. Liu, and F. Douglis. Automatic detection of fragments in dynamically generated web pages. In *Proceedings of the 13th International Conference on World Wide Web*, WWW 2004, pages 443–454. ACM, 2004. → page 38

[123] raml. RAML, 2022. URL https://raml.org/. Accessed: Sep 1, 2022. → page 117

[124] Reactive Trader. Reactive Trader. https://github.com/AdaptiveConsulting/ReactiveTraderCloud/, 2021. → page 91

[125] RedHat Linux 7. RedHat Linux . https://en.wikipedia.org/wiki/Red_Hat_Enterprise_Linux, 2018. → page 75

[126] F. Ricca and P. Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE 2001, pages 25–34. IEEE, 2001. → page 11

[127] D. Roest, A. Mesbah, and A. v. Deursen. Regression testing Ajax applications: Coping with dynamism. In *International Conference on Software Testing, Verification and Validation*, ICSE 2010, 2010. → pages 5, 41, 42

[128] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella. Deep reinforcement learning for black-box testing of android apps. *ACM Trans. Softw. Eng. Methodol.*, 31(4), jul 2022. ISSN 1049-331X. doi:10.1145/3502868. URL https://doi.org/10.1145/3502868. → page 156

[129] S. Roy Choudhary, M. R. Prasad, and A. Orso. X-PERT: Accurate Identification of Cross-browser Issues in Web Applications. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE 2013, pages 702–711. IEEE Press, 2013. → pages 16, 49, 54

[130] S. Sampath. Advances in user-session-based testing of web applications. *Advances in Computers*, 86:87–108, 2012. → page 9

[131] schemathesis. schemathesis, 2022. URL https://github.com/schemathesis/schemathesis. Accessed: Sep 1, 2022. → pages 120, 137, 143

[132] M. Schur, A. Roth, and A. Zeller. Mining workflow models from web applications. *IEEE Transactions on Software Engineering (TSE)*, 41(12): 1184–1201, Dec 2015. → pages 4, 8

[133] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering (TSE)*, pages 1083–1099, 2017. → page 118

[134] Selenium. Selenium web browser automation. https://www.selenium.dev/, 2022. Accessed: 2022-07-01. → pages 11, 134

[135] H. Shahriar and M. Zulkernine. Mutec: Mutation-based testing of cross site scripting. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS 2009, pages 47–53, USA, 2009. IEEE Computer Society. ISBN 9781424437252. → pages 88, 104, 114

[136] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2951–2959. Curran Associates, Inc., 2012. → pages 25, 66

[137] S. M. Sohan, C. Anslow, and F. Maurer. Spyrest: Automated restful api documentation using an http proxy server (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015, pages 271–276, 2015. → page 147

[138] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing Management*, 45 (4):427 – 437, 2009. ISSN 0306-4573. → page 27

[139] SourceForge. SourceForge. https://sourceforge.net/, 2021. → page 90

[140] springboot. SpringBoot, 2022. URL https://spring.io/projects/spring-boot/. Accessed: Sep 1, 2022. → page 118

[141] springdoc. springdoc-openapi, 2022. URL https://springdoc.org/. Accessed: Sep 1, 2022. → page 118

[142] springfox. SpringFox: Automated JSON API documentation for API's built with Spring, 2022. URL https://springfox.github.io/springfox/. Accessed: Sep 1, 2022. → page 118

[143] D. Stallenberg, M. Olsthoorn, and A. Panichella. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2021, pages 117–128. IEEE, 2021. → page 118

[144] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. Clustering-aided page object generation for web testing. In *Proceedings of 16th International Conference on Web Engineering*, ICWE 2016, pages 132–151. Springer, 2016. → pages 15, 21, 38, 64

[145] A. Stocco, M. Leotta, F. Ricca, and P. Tonella. APOGEN: Automatic Page Object Generator for Web Testing. *Software Quality Journal*, 25(3): 1007–1039, Sept. 2017. → pages 21, 64

[146] A. Stocco, R. Yandrapally, and A. 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, ESEC/FSE 2018, pages 503–514, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5573-5. → pages 2, 16, 176

[147] N. D. Study. Near duplicate study crawls, Sept. 2019. URL https://doi.org/10.5281/zenodo.3385377. → pages 64, 70

[148] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 245–256, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351058. doi:10.1145/3106237.3106298. URL https://doi.org/10.1145/3106237.3106298. → page 156

[149] Y. Sun, P. Jin, and L. Yue. A framework of a hybrid focused web crawler. In *2008 Second International Conference on Future Generation Communication and Networking Symposia*, volume 2, pages 50–53, Dec 2008. → page 84

[150] M. J. Swain and D. H. Ballard. Indexing via color histograms. In A. K. Sood and H. Wechsler, editors, *Active Perception and Robot Vision*, pages 261–273. Springer Berlin Heidelberg, 1992. → pages 14, 49, 53, 62, 64

[151] S. Thummalapenta, P. Devaki, S. Sinha, S. Chandra, S. Gnanasundaram, D. D. Nagaraj, S. Kumar, and S. Kumar. Efficient and change-resilient test automation: An industrial case study. In *2013 35th International Conference on Software Engineering*, ICSE 2013, pages 1002–1011, 2013. → page 2

[152] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *2013 35th International Conference on Software Engineering*, ICSE 2013, pages 162–171, May 2013. → pages 2, 41

[153] tikiwiki. TikiWiki. https://sourceforge.net/projects/tikiwiki/, 2021. → page 91

[154] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen. Selecting fault revealing mutants. 25(1):434–487. ISSN 1573-7616. → page 103

[155] A. Tombros and Z. Ali. Factors affecting web page similarity. In *Proceedings of the 27th European Conference on Advances in Information Retrieval Research*, ECIR 2005, pages 487–501. Springer-Verlag, 2005. → page 9

[156] P. Tonella, F. Ricca, and A. Marchetto. Recent advances in web testing. *Advances in Computers*, 93:1–51, 2014. → pages 3, 11

[157] A. Torsel. Automated Test Case Generation for Web Applications from a Domain Specific Model. In *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pages 137–142, July 2011. → pages 5, 60

[158] Tudu Lists. Tudu Lists. https://sourceforge.net/projects/tudu/, 2021. → page 91

[159] J. Upchurch and X. Zhou. Malware provenance: code reuse detection in malicious software at scale. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–9, Oct 2016. → page 15

[160] E. Viglianisi, M. Dallago, and M. Ceccato. Resttestgen: automated black-box testing of restful apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, ICST 2020, pages 142–152. IEEE, 2020. → pages 118, 119, 122

[161] T. A. Walsh, P. McMinn, and G. M. Kapfhammer. Automatic detection of potential layout faults following changes to responsive web pages (n). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2015, pages 709–714, Nov 2015. → page 114

[162] Y. Wang and M. Kitsuregawa. *Link Based Clustering of Web Search Results*, pages 225–236. Springer Berlin Heidelberg, 2001. → page 38

[163] H. Wu, L. Xu, X. Niu, and C. Nie. Combinatorial testing of restful apis. In *ACM/IEEE International Conference on Software Engineering*, ICSE 2022, 2022. → page 118

[164] xpath. Xpath, XML Language, 2022. URL https://en.wikipedia.org/wiki/XPath. Accessed: Sep 1, 2022. → page 56

[165] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC/SIGSOFT FSE*, pages 85–94. ACM, 2007. → pages 119, 146

[166] R. Yandrapally and A. Mesbah. Mutation analysis for assessing end-to-end web tests. In *2021 IEEE International Conference on Software Maintenance and Evolution*, ICSME 2021, pages 183–194, 2021. → page vii

[167] R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra. Robust test automation using contextual clues. In *Proceedings of 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 304–314. ACM, 2014. → page 2

[168] R. K. Yandrapally. Near-Duplicate Study Tools and DataSet For Replication. https://github.com/NDStudyICSE2019/NDStudy, 2019. GitHub Repository. → pages 10, 38

[169] R. K. Yandrapally. FragGen VM and Dataset for replication. https://doi.org/10.5281/zenodo.4007539, 2021. Zenodo Software Upload. → pages 44, 62, 82

[170] R. K. Yandrapally. MAEWU - Mutation analysis framework for E2E web testing. https://github.com/mutationwebapp/maewu, 2021. → pages 88, 106, 115

[171] R. K. Yandrapally. Replication package - Carving UI Tests to Generate API Tests and API Specifications. https://github.com/apicarve/apicarver, 2022. → pages 120, 135, 146

[172] R. K. Yandrapally and A. Mesbah. Fragment-based test generation for web apps. *IEEE Transactions on Software Engineering (TSE)*, pages 1–1, 2022. → pages vi, 134

[173] R. K. Yandrapally, A. Stocco, and A. Mesbah. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ICSE 2020, page 12 pages. ACM, 2020. → pages vi, 42, 48, 49, 54, 64, 66, 69, 70

[174] R. K. Yandrapally, S. Sinha, R. Tzoref-Brill, and A. Mesbah. Carving ui tests to generate api tests and api specifications. ICSE 2023, 2023. → page vii

[175] B. Yang, F. Gu, and X. Niu. Block mean value based image perceptual hashing. In *2006 International Conference on Intelligent Information Hiding and Multimedia*, pages 167–172, 2006. → pages 14, 16, 49

[176] J. Yang, E. Wittern, A. T. Ying, J. Dolby, and L. Tan. Towards extracting web api specifications from documentation. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories*, MSR 2018, pages 454–464, 2018. → page 147

[177] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 919–930, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. → page 107

[178] H. Yee, S. Pattanaik, and D. P. Greenberg. Spatiotemporal sensitivity and visual attention for efficient rendering of dynamic environments. *ACM Trans. Graph.*, 20(1):39–65, Jan. 2001. ISSN 0730-0301. → pages 14, 16, 49

[179] C. Zauner. *Implementation and benchmarking of perceptual image hash functions*. PhD thesis, 2010. → pages 14, 16, 49

[180] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 214–224, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. → page 86

[181] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu. Automatic web testing using curiosity-driven reinforcement learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering*, ICSE 2021, pages 423–435, 2021. → page 84

Appendix A

Publications

I also contributed as a second author in the following research paper during my PhD. However, it is not being included as part of this dissertation.

- ”Visual web test repair” [146]: Andrea Stocco, **Rahulkrishna Yandrapally**, and Ali Mesbah, European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018). 503 - 514. (*acceptance rate: 19%*)