AUTOMATED REPAIR OF PRESENTATION FAILURES IN WEB APPLICATIONS USING SEARCH-BASED TECHNIQUES

by

Sonal Mahajan

A Dissertation Presented to the
FACULTY OF THE USC GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2018

Copyright 2018 Sonal Mahajan
Dedication

To my parents, Pradip and Sharayu, and sister, Payal,
for their endless love, support, and encouragement.
Acknowledgements

Pursuing Ph.D. at USC has been one of the most rewarding, enriching, and challenging experiences of my life. It is my pleasure to acknowledge several people who were instrumental in the course of this pursuit.

Firstly, I would like to thank my advisor, Professor William G. J. Halfond, for his continuous support, feedback, and encouragement at all stages of my Ph.D. study. His enthusiasm and positive attitude towards new ideas dared me to think creatively and differently, and explore them without the fear of failing. The innumerable meetings that we had, and his timeless feedback and criticism helped me strengthen the roots of my dissertation research. He has repeatedly inspired me by setting an example through hard work, and carefully nurtured my technical and communication skills, all of which have helped me grow as a researcher. Besides my advisor, I would like to thank the rest of my dissertation committee: Prof. Nenad Medvidovic, Prof. Sandeep Gupta, Prof. Chao Wang, and Prof. Jyotirmoy Deshmukh, for their support, insightful comments, constructive feedback, and challenging questions. I am thankful to Prof. Phil McMinn from the University of Sheffield for being a great collaborator and imparting his knowledge of search-based techniques, which has helped me significantly improve my dissertation work.

My time at USC would not have been fun without the wonderful camaraderie of my labmates. They contributed towards creating a truly great and fun research environment, and helped me appreciate our different cultures, cuisines, and languages. I would like to thank Abdulmajeed Alameer for being a diligent co-author and a great friend, and for the interesting brainstorming sessions that we shared, Negarsadat Abolhassani for being a dear friend and for the entertaining and witty discussions that we had, Mian Wan for accompanying me in the walks back home and for introducing me to delicious Chinese snacks, Jiaping Gui for the many discussions that we had, especially, during our job search, Yingjun Lyu for the several fun conversations, and Ding Li (now at NEC Labs) for being very supportive and for helping me get through my early days at USC.

My heartfelt acknowledgements go to my teachers in all stages of my education, the individuals who have inspired me to learn and grow. I would like to specially thank Prof. Ilmi Yoon and Prof. Dragutin Petkovic of San Francisco State University for encouraging me to apply and enter the
doctoral program at USC. Their constant guidance and support helped me in making an informed
decision. I am also thankful to Prof. Ashutosh Muchrikar of Cummins College of Engineering for
his valuable support and encouragement during my undergraduate years.

Last but not the least, I would like to thank my family. My parents, Pradip and Sharayu, who
have faithfully believed in me, instilled in me the zeal for perfection and a quest for knowledge,
and supported me in all my pursuits. My sister, Payal, and brother-in-law, Navdeep, who have
always ensured that I don’t feel homesick by providing me with a warm and welcoming home in the
U.S., and always offered great technical insights that have helped me shape up my dissertation
effectively. My little nephew, Neil, who has been a perfect means of relieving my stress and
anxiety with his infectious laughter and loving, playful nature. My grandmother, who has been
particularly supportive and encouraging.
Table of Contents

Dedication ii
Acknowledgements iii
List of Tables ix
List of Figures x
Abstract xi

Chapter 1: Introduction 1
  1.1 Presentation Failures: Definition and Types 2
  1.2 Major Challenges 3
  1.3 Insights and Hypothesis 4
    1.3.1 Insight 1: Search-based techniques can be used for repair 4
    1.3.2 Insight 2: Repair has to quantify and balance two objectives 4
      1.3.2.1 Insight 2A: Presentation failures can be quantified 5
      1.3.2.2 Insight 2B: Aesthetic similarity can be quantified 5
    1.3.3 Hypothesis: 6
  1.4 Contributions 6
  1.5 Overview of Publications 7

Chapter 2: Background 12
  2.1 Web App Basics 12
  2.2 Types of Presentation Failures 13
    2.2.1 Layout Cross Browser Issues (XBIs) 14
    2.2.2 Mobile Friendly Problems (MFPs) 14
    2.2.3 Internationalization Presentation Failures (IPFs) 14
    2.2.4 Mockup-driven Development Problems (MDDPs) 15
    2.2.5 Regression Debugging Problems (RDPs) 15
  2.3 The Process of Debugging Presentation Failures 15
    2.3.1 P1. Detection 16
    2.3.2 P2. Localization 16
    2.3.3 P3. Repair 17
  2.4 Search-Based Techniques 17

Chapter 3: Overview of the Generalized Repair Approach, *FIX 18
  3.1 Design of *FIX 18
    3.1.1 AP1: Initialize 19
    3.1.2 AP2 and AP3: Search and Evaluate 20
Chapter 4: \( \mathcal{X} \text{Fix} \): Repair of Layout Cross Browser Issues (XBIs) 24

4.1 Background and Example ........................................... 25
4.2 Specialization of the Generalized Approach, \( \mathcal{X} \text{Fix} \) .................. 27
  4.2.1 Detection and Localization of Layout XBIs ..................... 28
  4.2.2 Repair of Layout XBIs ........................................ 28
    4.2.2.1 Overall Algorithm ..................................... 30
    4.2.2.2 Search for Candidate Fixes ............................. 33
    4.2.2.3 Search for the Best Combination of Candidate Fixes .... 37
4.3 Evaluation ........................................................... 38
  4.3.1 Implementation ............................................... 38
  4.3.2 Subjects ...................................................... 40
  4.3.3 Methodology .................................................. 41
  4.3.4 Threats to Validity ......................................... 43
  4.3.5 Discussion of Results ....................................... 44
    4.3.5.1 RQ1: Reduction of XBIs .................................. 44
    4.3.5.2 RQ2: Impact on Cross-browser Consistency .............. 45
    4.3.5.3 RQ3: Time Needed to Run \( \mathcal{X} \text{Fix} \) .................. 47
    4.3.5.4 RQ4: Similarity of Repair Patches to Real-world Websites’ Code . 47
4.4 Conclusion .......................................................... 48

Chapter 5: \( \mathcal{M} \text{Fix} \): Repair of Mobile Friendly Problems (MFPs) 50

5.1 Background .......................................................... 52
  5.1.1 Types of MFPs ............................................... 52
  5.1.2 Current Methods for Addressing MFPs ......................... 53
5.2 Specialization of the Generalized Approach, \( \mathcal{M} \text{Fix} \) .................. 54
  5.2.1 Detection of MFPs ........................................... 54
  5.2.2 Repair of MFPs .............................................. 54
  5.2.3 Phase 1: Segmentation ...................................... 55
  5.2.4 Phase 2: Localization ...................................... 57
    5.2.4.1 Identifying Problematic Segments ...................... 57
    5.2.4.2 Identifying Problematic CSS Properties ................ 58
  5.2.5 Phase 3: Repair .............................................. 59
    5.2.5.1 Metrics .................................................. 60
    5.2.5.2 Computing Candidate Mobile Friendly Patches ........... 61
    5.2.5.3 Generating the Mobile Friendly Patch ................. 62
5.3 Evaluation .......................................................... 63
  5.3.1 Implementation ............................................... 63
  5.3.2 Subjects ...................................................... 64
  5.3.3 Experiment One .............................................. 64
    5.3.3.1 Discussion of results .................................. 66
  5.3.4 Experiment Two .............................................. 68
    5.3.4.1 Discussion of results .................................. 68
  5.3.5 Threats to Validity ......................................... 70
Chapter 6: \textit{IFIX}: Repair of Internationalization Presentation Failures (IPFs) 72
6.1 Background .................................................. 73
6.2 Specialization of the Generalized Approach, \*FiX ......................... 75
   6.2.1 Detection and Localization of IPFs .................................. 75
   6.2.2 Repair of IPFs .................................................. 76
   6.2.3 Identifying Stylistically Similar Clusters .................................. 78
      6.2.3.1 Visual Similarity Metrics ........................................ 79
      6.2.3.2 DOM Information Similarity Metrics .............................. 80
   6.2.4 Candidate Solution Representation ...................................... 81
   6.2.5 Fitness Function .................................................. 81
      6.2.5.1 Generating the PUT \* ............................................. 81
      6.2.5.2 Fitness Function Components ................................... 82
   6.2.6 Search ........................................................... 84
6.3 Evaluation ...................................................... 86
   6.3.1 Implementation ................................................... 87
   6.3.2 Subjects .......................................................... 87
   6.3.3 Experiment One .................................................... 87
      6.3.3.1 Presentation of Results ........................................... 89
      6.3.3.2 Discussion of Results ........................................... 89
   6.3.4 Experiment Two ..................................................... 90
      6.3.4.1 Presentation of Results ........................................... 91
      6.3.4.2 Discussion of Results ........................................... 92
6.3.5 Threats to Validity ............................................... 93
6.4 Conclusion ...................................................... 94

Chapter 7: \textit{GFIX}: Repair of Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs) 95
7.1 Background and Motivation ............................................. 97
7.2 Specialization of the Generalized Approach, \*FiX ......................... 98
   7.2.1 Detection and Localization of MDDPs and RDPs ......................... 98
   7.2.2 Repair of MDDPs and RDPs ............................................ 100
      7.2.2.1 Overall Algorithm ............................................... 102
      7.2.2.2 CSS Properties from Visual Symptoms of MDDPs and RDPs ........ 106
      7.2.2.3 Size and Position Analysis ...................................... 110
      7.2.2.4 Color Analysis .................................................. 112
      7.2.2.5 Predefined Values Analysis .................................... 114
7.3 Evaluation ...................................................... 114
   7.3.1 Implementation ................................................... 115
   7.3.2 Subjects .......................................................... 115
      7.3.2.1 Selection of subjects for RDPs .................................. 116
      7.3.2.2 Selection of subjects for MDDPs ................................ 116
      7.3.2.3 Test case generation ............................................. 117
   7.3.3 Experiment One .................................................... 117
      7.3.3.1 Presentation of Results ........................................... 118
      7.3.3.2 Discussion of Results ........................................... 119
   7.3.4 Experiment Two ..................................................... 119
      7.3.4.1 Presentation of Results ........................................... 121
      7.3.4.2 Discussion of Results ........................................... 121
7.3.5 Threats to Validity ............................................... 122
## Chapter 8: Related Work

8.1 Automated repair of software systems ........................................ 124  
8.2 Cross Browser Issues (XBIs) .................................................. 125  
8.3 Mobile Friendly Problems (MFPs) .......................................... 126  
8.4 Internationalization Presentation Failures (IPFs) ....................... 127  
8.5 Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs) .................................................. 127  
8.6 Other presentation failure detection techniques .......................... 129  
8.7 Testing and analysis of web app client-side components ............... 129  
8.8 Graphical User Interface (GUI) Testing ................................... 130  

## Chapter 9: Conclusion and Future Work

9.1 Summary ................................................................. 131  
9.2 Future Work ............................................................ 133  

References ................................................................. 135
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Overview of the specializations of *Fix</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Subjects used in the evaluation of *Fix</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Effectiveness of *Fix in reducing XBIs</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>*Fix’s average run time in seconds</td>
<td>47</td>
</tr>
<tr>
<td>5.1</td>
<td>Subjects used in the evaluation of *MFix</td>
<td>65</td>
</tr>
<tr>
<td>6.1</td>
<td>Subjects used in the evaluation of *GFix</td>
<td>86</td>
</tr>
<tr>
<td>6.2</td>
<td>Effectiveness of *GFix in reducing IPFs</td>
<td>88</td>
</tr>
<tr>
<td>7.1</td>
<td>Mapping between symptoms and CSS properties</td>
<td>107</td>
</tr>
<tr>
<td>7.2</td>
<td>Categorization of CSS properties based on visual impact</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Subjects used in the evaluation of *GFix</td>
<td>115</td>
</tr>
<tr>
<td>7.4</td>
<td>Effectiveness of *GFix in reducing MDDPs and RDPs</td>
<td>118</td>
</tr>
<tr>
<td>7.5</td>
<td>*GFix’s median run time in seconds</td>
<td>118</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Process flow overview of debugging presentation failures .......................... 16
3.1 Overview of the generalized repair approach, *Fix ................................. 18
4.1 Example screenshots showing XBI ....................................................... 26
4.2 Overview of the XFix approach for repairing layout XBIs ......................... 30
4.3 Fitness function components of the XBI repair approach ......................... 36
4.4 High-level overview of the XFix tool ................................................ 38
4.5 Example of repair.css generated by XFix ........................................... 39
4.6 Similarity ratings given by participants in XFix human study .................. 46
4.7 Similarity of XFix generated repair patches to real-world websites’ code .... 48
5.1 Overview of the MFix approach for repairing MFPs ............................... 55
5.2 Example for demonstrating the MFix approach ..................................... 56
5.3 Distribution of the median mobile friendliness score across 10 runs .......... 66
5.4 Breakdown of the running time of MFix ............................................ 67
6.1 Example of an IPF and different ways of fixing it ................................ 77
6.2 Overview of the IFix approach for repairing IPFs ................................ 78
6.3 Example of ancestor elements with fixed width that need to be adjusted together with SimSet elements ......................................................... 82
6.4 Initializing the population ...................................................................... 85
6.5 Two UI snippets in the same equivalence class (from Hotwire) ............... 90
6.6 Similarity ratings given by user study participants ................................. 92
6.7 Weighted distribution of the ratings ..................................................... 93
7.1 Overview of the GFix approach for repairing MDDPs and RDPs ............. 101
7.2 Illustrative example ........................................................................... 102
7.3 Anti-aliasing example ......................................................................... 113
7.4 Examples of UI snippets used in the GFix’s user study ......................... 120
7.5 GFix’s user study results ...................................................................... 121
Abstract

The appearance of a web application’s User Interface (UI) plays an important part in its success. Issues degrading the UI can negatively affect the usability of a website and impact an end user’s perception of the website and the quality of the services that it delivers. Such UI related issues, called presentation failures, occur frequently in modern web applications. Despite their importance, there exist no automated techniques for repairing presentation failures. Instead repair is typically a manual process, where developers must painstakingly analyze the UI of a website, identify the faulty UI elements (i.e., HTML elements and CSS properties), and carry out repairs. This is labor intensive and requires significant expertise of the developers.

My dissertation addresses these challenges and limitations by automating the process of repairing presentation failures in web applications. My key insight underlying this research is that search-based techniques can be used to find repairs for the observed presentation failures by intelligently and efficiently exploring large solution spaces defined by the HTML elements and CSS properties in a web page. Based on this insight, I designed and developed four techniques for the automated repair of different types of presentation failures in web applications. The first technique focuses on the repair of layout Cross Browser Issues (XBIs), i.e., inconsistencies in the appearance of a website when rendered in different web browsers. The second technique addresses the Mobile Friendly Problems (MFPs) in websites, i.e., improves the readability and usability of a website when accessed from a mobile device. The third technique repairs problems related to internationalization in web application UIs. Lastly, the fourth technique addresses issues arising from mockup-driven development and regression debugging. In the empirical evaluations, all of the four techniques were highly effective in repairing presentation failures, while in the conducted user studies, participants overwhelmingly preferred the visual appeal of the repaired versions of the websites compared to their original (faulty) versions. Overall, these are positive results and indicate that my repair techniques can help developers repair presentation failures in web applications, while maintaining their aesthetic quality.
Chapter 1

Introduction

The number of web applications is on a constant rise, with over 1.8 billion websites reported as of December 2017 [1]. The proliferation of web applications has pervaded all aspects of human activities, from banking to social networking and from entertainment to shopping. In fact, based on a recent report released by the Department of Commerce, a total revenue of over 450 billion dollars was generated in the United States from e-commerce sales alone in the year 2017 [4]. The increasing popularity of web applications is driven by companies offering their services and products via interactive and feature-rich websites that end users can access from a wide range of browsers running on a variety of different platforms and devices. To further expand their outreach to international markets, companies often provide translated text and localized media content on their websites in order to effectively communicate with a global audience.

An attractive and visually appealing appearance of a website’s User Interface (UI) plays an important part in its success. A recent study underscores this point by noting that an average visitor forms a first impression of a web page within the first 50 milliseconds of visiting the page [152] — an amount of time that is heavily influenced by the web page’s aesthetics. Companies invest a significant amount of effort to design and implement their websites. It is typical for companies to employ teams of graphic designers and web programmers to get a website’s “look and feel” correct. This effort is important because studies have shown that the aesthetics of a website significantly impact end users’ overall evaluation of a website; particularly, impressions of trustworthiness and usability [152, 88, 87, 151, 71, 67, 70, 133, 145]. Issues degrading the visual consistency and aesthetics of a website can undermine this effort and negatively impact end users’ perception of the website and the quality of the services that it delivers. These issues can also seriously impact a website’s usability or functionality likely leading to a frustrating and poor user experience. Thus, such appearance related issues have the potential to severely impact the website’s success and affect the branding a company is trying to achieve [30]. Unfortunately,
despite the severe impact of such appearance related issues, they are found to occur frequently in modern web applications [137, 47, 73]. Researchers have recognized the need for automating the process of debugging appearance related issues in web applications and have proposed several approaches for various parts of the debugging process. However, these techniques are limited in their applicability as they can only detect the appearance related issues, with the repair remaining a labor intensive manual task. This motivates the need for automating the process of repairing appearance related issues in web applications.

1.1 Presentation Failures: Definition and Types

The appearance related issues in web applications are called presentation failures, which are formally defined as discrepancies between the actual appearance of a website and its intended appearance. Presentation failures are found pervasively in modern web applications [137, 47, 73] and can cause serious usability problems or significantly distort the intended appearance of a website.

Presentation failures can occur in web applications due to several reasons, such as developer errors and differences in the rendering engine of browsers. I now discuss examples of different types of presentation failures. The first type is layout Cross Browser Issues (XBIs), which occur when a web page is rendered inconsistently across different browsers. XBIs tend to arise because of differences in the interpretations of HTML and CSS standards by the different browsers. The second type is Mobile Friendly Problems (MFPs), which occur when users access web pages from a non-traditional sized device, such as a smartphone or tablet, and the pages are not designed to be mobile friendly. Such pages can exhibit a range of usability issues, such as unreadable text, cluttered navigation, or content that overflows the device’s viewport. The third type is Internationalization Presentation Failures (IPFs), which occur when a page is translated from one language to another (e.g., English to Spanish) and cause distortions in the layout of the page. Such distortions are caused by differences in the lengths of text in different languages. The fourth type is Regression Debugging Problems (RDPs). These presentation failures occur when developers perform maintenance tasks, such as correcting a bug or refactoring the HTML structure, and cause the page to appear differently than its original (intended) version. For example, a refactoring may change a page from a table-based layout to one based on ⟨div⟩ tags. During this process, the developers may inadvertently introduce a fault that changes the appearance of the page in an unintended manner. The fifth and last type is Mockup-driven Development Problems (MDDPs). In the mockup-driven style of development, developers use mockups — highly detailed renderings
of the intended appearance of the web page — to guide their implementation of the web pages. Developers may accidentally introduce faults during this process, which causes the implemented page to appear differently than its mockup.

### 1.2 Major Challenges

The detection, localization, and repair of presentation failures poses numerous challenges for developers and testers. First, detection of presentation failures is an expensive task, since typically testers must look at the UI of a website and identify when the UI does not conform with its intended appearance. Second, localization of presentation failures is difficult given the complex layout and styles of modern web pages. Third, developers lack a standardized way to repair presentation failures and generally have to resolve them on a case by case basis.

Existing tools are limited in helping developers to debug presentation failures. Although tools, such as Firebug [22], can provide useful information, developers still require expertise to manually analyze the presentation failures and then repair them by performing the necessary modifications so that the page renders correctly. Automated UI testing techniques, such as X-PERT [137], Google Mobile-Friendly Test Tool (GMFT) [26], and GWALI [49], are only able to detect and localize presentation failures (i.e., they address only the first two of the three previously listed challenges), but are incapable of repairing presentation failures so that the rendering of a web page can conform with its expected appearance. Repairing presentation failures is thus unfortunately strictly a manual process that is labor intensive and guided by a developer’s intuition and experience. Therefore, the accuracy and efficiency of this process can vary significantly by developer. The challenges and limitations of existing techniques motivate my research to automate the process of repairing presentation failures in web applications.

Repairing presentation failures, however, poses several challenges. The first challenge is that the solution space for identifying a repair is very large. Finding a repair for presentation failures means identifying new values for CSS properties of HTML elements that can make the faulty appearance of the web page match its correct appearance as closely as possible. These three UI aspects, i.e., HTML elements, CSS properties, and their values, form the three dimensions of the solution space, which can grow multiplicatively with respect to the number of HTML elements and CSS properties in a page. A typical web page can contain hundreds or thousands of HTML elements, each with several dozen CSS properties that range over a large set of possible values, for example, the `background-color` CSS property can be one of 16 million colors. Therefore, a brute force approach for finding the fix value is not scalable. The second challenge is that the
rendering of a web page is controlled by complex and dynamic interactions between the HTML and CSS of a page, making it difficult to analytically model the interactions between HTML and CSS. This problem is further compounded as the rendering rules are context-sensitive, meaning that the analytical model deduced for one web page rendered in a browser is not generalizable across different web pages and browsers. The third challenge is that a repair must be constructed carefully to modify the problematic UI elements in the page without introducing new presentation failures, which can easily occur due to complex and cascading interactions between HTML and CSS in a web page. This means that any potential repair must be evaluated in the context of not only how well it resolves the targeted presentation failure, but also its impact on the rest of the page’s layout as a whole. This task is complicated because it is possible that more than one element will have to be adjusted to repair a presentation failure.

1.3 Insights and Hypothesis

In this section, I present the key insights underlying my research and the hypothesis that this dissertation tests in order to realize the goal of automatically repairing presentation failures in web applications.

1.3.1 Insight 1: Search-based techniques can be used for repair

To address the repair challenges discussed above, my first key insight is that search-based techniques can be used to repair presentation failures in web applications. An important characteristic of the presentation failures problem domain that motivates this insight is that repairing presentation failures does not require finding a pixel-perfect, optimal solution. Identifying an approximate or a close-to-optimal solution that can resolve the presentation failures while maintaining close aesthetic similarity to the page’s original version is sufficient. Search-based techniques are ideal for this type of problem because they can explore large solution spaces intelligently and efficiently to find a good approximate solution. Such techniques typically work by evaluating the quality of potential solutions until a good enough solution is found or the allocated computation budget is fully exhausted. To evaluate the quality of a solution, search-based techniques use an objective function that can guide the search to likely solutions. The insight of designing such an objective function for repairing presentation failures is described in Insight 2.
1.3.2 Insight 2: Repair has to quantify and balance two objectives

My second key insight is that the best repair has to balance two objectives, or competing constraints: minimize the number of presentation failures in the page under test and maximize the page’s aesthetic similarity to its original version. Search-based techniques are well-suited for executing such multi-objective optimization problems, since they can effectively balance a number of competing constraints to produce an overall best repair. The details for quantifying these two objectives are explained in Insights 2A and 2B, respectively.

1.3.2.1 Insight 2A: Presentation failures can be quantified

My insight for the first objective is that presentation failures in a web page can be quantified by leveraging existing detection techniques, such as X-PERT [137] for XBLs. The detection techniques typically compare a page under test with its intended appearance and report the differences as presentation failures. The number of presentation failures in a page reported by the detection techniques can be used to determine how close a candidate repair found during a search is to making the page failure free. Such an objective function could guide the search to a repair that minimizes the number of presentation failures in a page. When the number of presentation failures converges to zero, it would imply that all of the presentation faults in the page have likely been identified and repaired. One weakness of such an objective function, however, is that it is unlikely to be able to reliably guide the search to solutions that maintain (or enhance) the aesthetic quality of the page. For example, solutions generated by simply hiding the problematic HTML elements or assigning extreme CSS values can resolve the presentation failures in a page but on the other hand significantly disrupt the page’s layout and affect its functionality and usability, thereby likely being unacceptable to developers and end users. This motivates the need for finding a repair that can faithfully maintain (or enhance) the page’s aesthetic quality after repair, which forms the second objective of the objective function.

1.3.2.2 Insight 2B: Aesthetic similarity can be quantified

My insight for the second objective is that the aesthetic similarity of a page before and after repair can be quantified by analyzing the page’s layout and that this quantification can also be used as part of the objective function. The layout of a web page rendered in a browser can be modeled as a graph, with HTML elements in the page forming the vertices and the layout relationships, such as parent-child, sibling, and position, of the elements with respect to each other forming the edges. Such a model can be built for a page before and after applying a repair and the two models
can be compared. The differences extracted from the comparison can serve as the quantification of the closeness of the aesthetic similarity of the page after repair to its original version. This measurement of aesthetic similarity can be used as part of the objective function to guide the search to a repair that minimizes the layout deviation between the before and after repair versions of the page.

### 1.3.3 Hypothesis:

Based on the above two insights, the hypothesis statement of my dissertation is:

```
Search-based techniques can repair presentation failures in a web page with high effectiveness.
```

To evaluate the hypothesis, I developed different repair algorithms using search-based techniques to resolve the different types of presentation failures in web applications discussed in Section 1.1. I then showed the effectiveness of these techniques in resolving the detected presentation failures. I quantified effectiveness as the reduction in the presentation failures reported by existing detection techniques in the before and after repair versions of the pages. I also conducted user studies to understand the reduction in the human-observable presentation failures and quantify the impact of the generated repairs on the aesthetic quality of the pages from a human perspective. The empirical evaluations demonstrated that my repair algorithms can provide fixes for resolving a high number of the observed presentation failures in web pages, while, in most cases, maintaining or enhancing the pages’ aesthetic appeal. These results confirm the hypothesis of my dissertation and indicate that my research is potentially of high usefulness to developers for automatically repairing different types of presentation failures in web applications.

### 1.4 Contributions

The contributions of my dissertation include the design and development of four search-based techniques for the automated repair of different types of presentation failures in web applications and empirical evaluations of the techniques to assess their effectiveness. To the best of my knowledge, my work is the first automated approach for generating repairs for presentation failures, and the first to apply search-based repair techniques to web pages.

1. **Repair techniques** — I designed and developed four techniques listed below for repairing the different types of presentation failures in web applications discussed in Section 1.1. The different techniques are explained in detail in Chapters 4, 5, 6 and 7. For easy understanding of my repair techniques, I explain their common structure in Chapter 3.
(a) $\mathcal{X}$Fix— The goal of this technique is to find potential fixes that can repair layout XBIs detected in web pages.

(b) $\mathcal{M}$Fix— The goal of this technique is to improve the mobile friendliness of web pages by automatically repairing the MFPs detected in the pages.

(c) $\mathcal{I}$Fix— The goal of this technique is to automatically repair IPFs that have been detected in translated versions of web pages.

(d) $\mathcal{G}$Fix— The goal of this technique is to automatically repair the MDDPs and RDPs detected in web pages.

2. Empirical evaluation of the techniques — I conducted empirical evaluations of the four techniques listed above on real-world subject web pages to demonstrate their effectiveness. For measuring effectiveness, I used existing detection techniques to compute the reduction in the number of observed presentation failures before and after repair, and conducted user studies to understand the visual quality of the generated fixes. I also measured the total running time required by the different techniques to generate repairs for each of the subject web pages.

1.5 Overview of Publications

In this section, I provide an overview of the publications that I have developed during the course of this dissertation. The dissertation work is mainly divided into four chapters, Chapters 4, 5, 6 and 7, that correspond to the four techniques I developed to repair different types of presentation failures in web pages. Each of the chapters is based on one or more papers, which have been published or are under submission. The 15 papers corresponding to the four chapters are listed below. For each of the papers, I was the primary author (or one of the primary authors), with contributions including idea, design, and evaluation of the work. All of the papers were co-authored with my Ph.D. advisor, Prof. William G. J. Halfond.

Chapter 4: Repair of Layout Cross Browser Issues (XBIs)

In this chapter, I discuss the repair technique, $\mathcal{X}$Fix, that I designed for repairing layout XBIs in web pages. This chapter had two publications at the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA) in 2017. The first publication was a full paper in the main research track [97] and the second publication was a tool demo paper [98]. The main research track paper received the ACM SIGSOFT Distinguished Paper Award. An extended version of the
main research track paper is currently in preparation to be submitted at a software engineering journal [96]. All of the papers were co-authored with Abdulmajeed Alameer, a fellow Ph.D. student at USC, and Prof. Phil McMinn, a collaborator from the University of Sheffield, UK.


**Chapter 5: Repair of Mobile Friendly Problems (MFPs)**

In this chapter, I discuss the repair technique, *MFix*, that I designed for repairing MFPs in web pages to improve their mobile friendliness. This chapter had one publication in the research track of the IEEE/ACM International Conference on Software Engineering (ICSE) in 2018 [95]. This work was developed in collaboration with Negasadat Abolhassani, a fellow Ph.D. student at USC, and Prof. Phil McMinn.


**Chapter 6: Repair of Internationalization Presentation Failures (IPFs)**

This chapter discusses the repair technique, *IFix*, that I developed for repairing IPFs in web pages. This work was published in the research track of the IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2018, and was a recipient of the *IEEE Distinguished Paper Award* [99]. The paper was co-authored with Abdulmajeed Alameer and Prof. Phil McMinn. To facilitate the repair, I first designed a technique (GWALI) for detecting and localizing IPFs in web pages. This work was originally published at the IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2016, where it received
Chapter 7: Repair of Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs)

In this chapter, I explain the repair technique, \texttt{GFix}, that I developed for repairing MDDPs and RDPs in web pages. This work is currently under submission at a software engineering journal [107]. It was co-authored with Prof. Phil McMinn.

Prior to developing \texttt{GFix}, I developed a detection and localization technique, WebSee. WebSee was originally published as a new ideas paper [102] at the IEEE/ACM International Conference on Automated Software Engineering (ASE) in 2014. It was later extended and published as a full paper in the research track of the IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2015 [103]. Its tool demo paper was also published at the same conference (ICST 2015) [104]. I later developed a technique based on WebSee to identify the visual inconsistencies across web pages of the same web application that was published as a short paper at the Asia-Pacific Software Engineering Conference (APSEC) in 2016 [101]. This work was developed during my summer internship at Infosys Labs in collaboration with Krupa Benhur Gadde and Anjaneyulu Pasala. An extended version of the APSEC 2016 paper is currently in preparation to be submitted at a software engineering journal [100].

I developed two techniques to perform root cause analysis of MDDPs and RDPs to identify the faulty HTML elements and CSS properties that are likely responsible for the observed failures.
The first technique was a preliminary work using search-based techniques that appeared as a new ideas short paper [106] at the Workshop on Search-Based Software Testing (SBST) in 2014. The second technique was based on visual symptoms and probabilistic modeling, and was published as a full paper [105] in the research track of the IEEE International Conference on Software Testing, Verification and Validation (ICST) in 2016. Both the papers were developed in collaboration with Bailan Li, a former Ph.D. student at USC. The ICST 2016 paper was also co-authored with Pooyan Behnamghader, a fellow Ph.D. student at USC.


Chapter 2

Background

This chapter provides the necessary background information that is used throughout the dissertation. Section 2.1 discusses the fundamentals of a typical web application and how a browser renders the User Interface (UI) of a web page. In Section 2.2, I discuss five different types of presentation failures. Finally, in Section 2.3, I discuss the process of debugging presentation failures in web applications.

2.1 Web App Basics

A web application is a client-server software application in which the application code is present on a remote server and is delivered to the client running a web browser over the Internet. Typically, the client-server framework of web applications is modeled as a three-tier architecture, presentation tier, business logic tier, and the data tier. The presentation tier is considered to be the topmost tier, displaying the web application to the end users as HyperText Markup Language (HTML) pages rendered in their browsers. The business logic or the middle tier contains the core functionality or behavior of the web application, and controls the interaction between the presentation and data tier. The data tier houses the database servers where information is stored persistently. When the end users interact with the presentation tier of the web application, the client (browser) submits a Hyper Text Transfer Protocol (HTTP) request message to the business logic tier on the server. The business logic tier processes the request, interacts with the data tier if required, and fetches/generates the HTML page and related resources, and sends this as a response back to the client. The client then renders the received HTML page in the response in the browser.

Modern web applications typically follow the Model-View-Controller (MVC) design pattern in which the application code (Model and Controller) runs on a server accessible via the Internet and
delivers HTML and CSS based web pages (View) to a client running a web browser. The repair techniques presented in my dissertation focus on repairing the View part of the web application.

I now discuss how an HTML page is rendered in a web browser. The rendering engine (also called the layout engine) in a browser is responsible for displaying the requested HTML content. At a high-level, the rendering engine follows four steps to display the HTML page. The first step is parsing the HTML to construct the Document Object Model (DOM) tree. The HTML code is parsed using a tokenization algorithm. The tokenization algorithm parses the HTML into tokens such as start tags, end tags, and attribute name-value pairs. The DOM tree construction is performed in parallel to the tokenization algorithm. Every HTML element produced by the tokenization algorithm, such as `html`, `body`, and `div`, is added to the DOM tree. Each HTML element may be referenced in the DOM tree using a unique expression, called an “XPath”. After the DOM tree is constructed, the second step is render-tree construction. The render tree represents the visual order in which the elements will be displayed. The render-tree is constructed by first parsing the Cascading Style Sheets (CSS) that describe the presentation properties of an HTML page. By parsing the CSS, the style rule for every element to be rendered is computed by calculating the visual properties applying to it. The visual properties are calculated based on the CSS rules, cascade order, and specificity. After the render tree is constructed, the third step is to decide the layout of the render tree. This is done by mapping the DOM tree to the render tree. The bounding box for each element to be rendered on the browser screen is computed using a flow based model. A bounding box gives the physical display location and size of an HTML element on the browser screen. Then the last step is painting the render tree. This is done by traversing the render tree and calling the rendering engine’s paint() function to display content on the screen. The painting is done from the back to the front of the layout, meaning the background color is painted first, followed by the background image, the border, children, and finally outline.

### 2.2 Types of Presentation Failures

A presentation failure is defined as a discrepancy in the actual appearance of a web page and its intended appearance. Examples of presentation failures are Cross Browser Issues (XBIs), Mobile Friendly Problems (MFPs), Internationalization Presentation Failures (IPFs), Mockup-driven Development Problems (MDDPs), and Regression Debugging Problems (RDPs). My dissertation targets the automated repair of these five different types of presentation failures. I discuss each of them in detail below.
2.2.1 Layout Cross Browser Issues (XBIs)

XBIs are defined as inconsistencies in the appearance or behavior of a website across different browsers. XBIs have been a serious concern for web developers for a long time. A simple search on StackOverflow — a popular technical forum — with the search term “cross browser” results in over 23,000 posts discussing ways to resolve XBIs, of which approximately 7,000 are currently active questions [43]. Although XBIs can impact the appearance or functionality of a website, the vast majority — over 90% — result in appearance related problems [137]. A significant class of appearance related XBIs is called layout XBIs, which collectively refer to any XBI that relates to an inconsistent layout of HTML elements in a web page when viewed in different browsers. These are the type of XBIs targeted by my dissertation. Layout XBIs appear in over 56% of the websites manifesting XBIs [137]. The impact of layout XBIs on web pages can range from minor cosmetic differences to severe aesthetic distortions to serious usability problems. Layout XBIs tend to arise from different interpretations of the HTML and CSS specifications, and are not per se, faults in the browsers themselves [5]. Additionally, some browsers may implement new CSS properties or existing properties differently in an attempt to gain an advantage over competing browsers [113].

2.2.2 Mobile Friendly Problems (MFPs)

Many websites are not designed to gracefully handle users who are accessing their pages through a non-traditional sized device, such as a smartphone or tablet. These problematic sites may exhibit a range of usability issues, such as unreadable text, cluttered navigation, or content that overflows the device’s viewport and forces the user to pan and zoom the page in order to access content. Such usability issues are collectively referred as MFPs [26, 13] and can lead to a frustrating and poor user experience.

2.2.3 Internationalization Presentation Failures (IPFs)

Companies often employ internationalization (i18n) frameworks for their websites, which allow the websites to provide translated text or localized media content, to communicate effectively with a global audience. However, because the length of translated text differs in size from text written in the original language of the page, the page’s appearance can become distorted. HTML elements that are fixed in size may clip text or look too large, while those that are not fixed can expand, contract, and move around the page in ways that are inconsistent with the rest of the page’s layout. Such distortions, called IPFs, reduce the aesthetics or usability of a website and
occur frequently — a recent study reports their occurrence in over 75% of internationalized web pages [47].

2.2.4 Mockup-driven Development Problems (MDDPs)

Mock-up-driven development [123, 92, 130] is a popular style of web app development. In this style of development, front-end developers use mockups — highly detailed renderings of the intended appearance of the web application — to guide their development of web application templates. The developers are generally expected to create “pixel perfect” matches of these mockups [24] using web development tools, such as Adobe Muse, Amaya, or Visual Studio. Back-end developers also make changes to these templates by adding dynamic content. Both front-end and back-end developers need to check that their respective changes are consistent with the mockup. Inconsistencies in the appearance of the implemented web application and its mockups are called MDDPs.

2.2.5 Regression Debugging Problems (RDPs)

Developers often perform maintenance on their web pages in order to introduce new features, correct a bug, or refactor the HTML structure. The goal of such regression debugging tasks is to change the structure or HTML code of a page without altering its visual appearance. For example, a developer may refactor a web page to transition it from using a table-based layout to one based on the use of the $\langle$div$\rangle$ tag. During this modification, developers may inadvertently introduce a fault in the code that results in presentation failures called as RDPs.

2.3 The Process of Debugging Presentation Failures

From a high level, the process of debugging presentation failures in web applications involves answering three questions; is there a presentation failure (Detection), where are the faulty HTML elements causing the failure (Localization), and how to correct the faulty elements to prevent the failure (Repair). Figure 2.1 gives an overview of the process of debugging presentation failures in web applications. The process takes two inputs: the page under test (PUT) to be analyzed for presentation failures and an oracle that specifies the visual correctness properties of the PUT. The process of repairing presentation failures is comprised of three phases. The first phase, Detection, compares the rendering of the PUT with the oracle to identify if there exist visual differences. The second phase, Localization, identifies a set of potentially faulty HTML elements in the PUT that are most likely responsible for causing the detected failure. The third phase, Repair, produces and applies fixes to the faulty PUT to resolve the reported presentation failure. The output of
the repair process is a page, PUT’, which is a repaired version of the PUT. I now introduce the three phases of the debugging process in more detail.

![Diagram of debugging process](image)

**Figure 2.1: Process flow overview of debugging presentation failures**

### 2.3.1 P1. Detection

The goal of the first phase, *Detection*, is to report areas of visual differences in the PUT with respect to its appearance oracle. The UIs of modern web applications are highly complex and dynamic. Back-end server code dynamically generates content and client-side browsers render this content based on complex HTML and CSS rules. This makes detecting presentation failures both a labor-intensive and error-prone process. To illustrate, a tester must visually compare the rendering of each page against an oracle, such as a design mockup, to detect that a presentation failure has occurred. Therefore the ability to automatically detect presentation failures is a key first step to automate the localization and repair phases.

### 2.3.2 P2. Localization

The goal of the second phase, *Localization*, is to identify a set of HTML elements in the PUT that may be responsible for the detected presentation failures. Identifying the potentially faulty HTML elements for a presentation failure in modern web applications is challenging since an element’s visual appearance is controlled by a complex series of interactions defined by the page’s HTML structure and CSS rules. The widespread use of HTML rendering features, such as floating elements, overlays, and dynamic sizing, also increases the difficulty of identifying the faulty element as there is often no obvious or direct connection from the rendered appearance to the structure of the underlying HTML. Therefore an accurate localization of potentially faulty HTML elements in the PUT is an important step for effectively repairing the observed presentation failures.
2.3.3 P3. Repair

The third and final phase, Repair, of the debugging process takes the Detection and Localization information to an actionable result, finding a suitable fix for the presentation fault. Although it is possible for developers to manually fix their code given the localization information provided by phase two, this process poses several challenges. First, modern web pages may contain several CSS properties defined for each HTML element that control its appearance. This makes it challenging for developers to accurately determine which CSS properties of the reported faulty elements need to be adjusted in order to repair the presentation fault. Assuming that the relevant CSS properties can be identified, the developers must still carefully construct the repair. Due to complex and cascading interactions between styling rules, a change in one part of the PUT’s UI can easily introduce further issues in another part of the page. This means that any potential repair must be evaluated in the context of not only how well it resolves the targeted presentation failure, but also its impact on the rest of the page’s layout. For these reasons, automated techniques would help developers to more effectively and efficiently repair their faulty web pages. My dissertation work automates this phase of the debugging process to generate repairs for the observed presentation failures in web pages, while maintaining their aesthetic quality.

2.4 Search-Based Techniques

Search-based techniques explore large solution spaces intelligently and efficiently to find an optimal solution. Search-based techniques typically begin by selecting a sample set of one or more candidate solutions from the solution space. A fitness function then assesses their quality and assigns each candidate a score indicating how fit (i.e., good) they are. This score, called the objective score or fitness score, helps in establishing a “direction” of search. If an improvement in the fitness score is observed over the original score, then it indicates that the search is progressing in the correct direction. The search algorithm can then select new candidate solutions that are in this promising direction to achieve further fitness improvements. The search then checks if a stopping criteria is met, if yes the best candidate, i.e., the candidate with the optimal fitness score, is output as the solution. If not, the search cycle continues.
Chapter 3

Overview of the Generalized Repair Approach, *F\text{ix}*

The four different repair techniques that I have developed as a part of this dissertation follow a common design of using search-based techniques for identifying a repair that can resolve the observed presentation failures while maintaining the aesthetic quality of the pages. To facilitate easy explanation of the repair techniques, I have abstracted their common structure into a generalized approach called *F\text{ix}*. I provide details of *F\text{ix}* in Section 3.1 and discuss its specializations in Section 3.2.

### 3.1 Design of *F\text{ix}*

The presentation of a web page, i.e., the placement, size, and appearance of a page's UI elements, is controlled by the page's HTML elements and CSS properties. Therefore, the presentation failures in a page can be fixed by changing the values of CSS properties that can make the faulty appearance of HTML elements in the page match the correct, intended appearance as closely as possible. With this in mind, *F\text{ix}* formally defines a fix for resolving presentation
failures as a tuple \(<e, p, v, v'\rangle\), where \(e\) is an HTML element in the page, \(p\) is a CSS property of \(e\), \(v\) is the value of \(p\), and \(v'\) is the suggested new value for \(p\).

Figure 3.1 shows an overview of *Fix. It takes four inputs. The first input is the page under test (\(PUT\)) exhibiting presentation failures. The \(PUT\) is obtained via a URL that points to a location on the file system or network that provides access to all of the necessary HTML, CSS, Javascript, and media files for rendering the \(PUT\). The second input is the oracle that specifies the correct or intended rendering of the \(PUT\). The third and fourth inputs are the detection (\(D\)) and localization (\(L\)) functions, respectively, that can identify the presentation failures and locate the potentially faulty HTML elements in the \(PUT\) responsible for the observed presentation failures. Existing detection and localization techniques, such as X-PERT [137] for Cross Browser Issues (XBIs) and GWALI [49] for Internationalization Presentation Failures (IPFs), can be leveraged to provide \(D\) and \(L\). The output of *Fix is a page, \(PUT'\), a repaired version of the \(PUT\).

I have abstracted the common components of my repair techniques into four abstraction points in *Fix. An abstraction point is a processing hook or plug-in point that allows developers to add specialized code to provide specific functionality. The first abstraction point, Initialize, identifies a set of initial candidate solutions from the solution space. Then, the Search abstraction point takes as input the initial set of candidate solutions and explores the solution space to find an optimal solution by using the Evaluate abstraction point as a guide. The Evaluate abstraction point (i.e., the fitness function) guides the search to the optimal solution by considering two objectives, minimize the number of presentation failures in the \(PUT\) and maximize its aesthetic similarity to the original version. Finally, the Terminate abstraction point determines whether the search should terminate or proceed to another iteration of the search cycle. I now discuss the purpose of each of the four abstraction points in more detail.

3.1.1 AP1: Initialize

Proper initialization of the search space has been repeatedly cited in literature as an important step for speeding up the convergence of a search-based algorithm [79, 114, 90, 65, 93]. With this in mind, the goal of the Initialize abstraction point is to identify a set of initial candidate solutions from the solution space that are potentially close to the optimal solution. To do this, domain specific knowledge of the presentation failure being targeted can be used. For identifying \(e\) in the fix tuple, the potentially faulty HTML elements reported by the input function \(L\) can be used. For example, for IPFs, \(e\) can be selected from the potentially faulty HTML elements reported by the GWALI tool [49]. For \(p\), relevant CSS properties that can influence the observed symptom of a presentation failure can be used. For example, the relevant CSS properties for IPFs are
font-size, width, and height, as they can be adjusted to make the layout of the page adapt to the changes from text translation. For identifying \( v' \), the visual manifestation of the presentation failure can be analyzed. For example, for IPFs, the text expansion that occurred in the PUT can be used to estimate a candidate \( v' \). The design of the Initialize abstraction point will vary for other types of presentation failures.

### 3.1.2 AP2 and AP3: Search and Evaluate

The goal of the Search abstraction point is to explore the solution space to find an optimal solution by using the Evaluate abstraction point (i.e., the fitness function) to guide it. In the context of presentation failures, an optimal solution is defined as a repair that when applied to the PUT causes it to conform with the oracle.

The Search abstraction point takes as input the initial set of candidate solutions produced by the Initialize abstraction point and searches the solution space to find new CSS values for the potentially faulty HTML elements and CSS properties that can repair the PUT. The Search abstraction point can be designed using different search-based algorithms, such as local search (e.g., Alternating Variable Method (AVM) search and simulated annealing) and global search (e.g., genetic algorithm). An appropriate search-based algorithm must be selected based on the following two criteria. First, the search technique should generate a repair for a presentation failure in a reasonable amount of time. In a search-based technique, the fitness function needs to be invoked multiple times to evaluate the quality of the generated candidate solutions. In the context of presentation failures, the fitness function is an expensive call since it requires applying the candidate repair to the PUT, capturing the new layout of the page, and comparing the new layout with the oracle to get the new fitness score. Therefore, a quicker convergence of the search-based algorithm to generate a repair can be achieved by minimizing the calls to the fitness function. Second, the search technique should resolve the PUT’s presentation failures without introducing new failures, while also ensuring an aesthetically pleasing and usable layout. This can be done by encoding these considerations in the design of the fitness function used to guide the search. A good fitness function can be built to balance both objectives: minimize the number of presentation failures in the PUT and maximize the PUT’s aesthetic similarity to the original version. The first objective can be designed by leveraging a measurement of the number of presentation failures detected in the PUT, by using the function \( D \). For the second objective, different User Interface (UI) change metrics can be designed to measure the deviation in the appearance of the changed PUT with respect to its original version. The design of the fitness
function and the selection of the search technique is different for different types of presentation failures.

3.1.3 AP4: Terminate

At the end of each search cycle, the Terminate abstraction point determines whether the search should terminate or proceed to another iteration of the search cycle. The search termination criteria can be different for different problem domains. In the context of presentation failures, common terminating conditions can be: all of the failures in the PUT are resolved or the amount of allocated resources, such as time, is exhausted.

3.2 Specializations of *FIX

<table>
<thead>
<tr>
<th>Detection (D)</th>
<th>XBI</th>
<th>MFP</th>
<th>IPF</th>
<th>MDDP</th>
<th>RDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>*FIX specialization</td>
<td>*FIX (§4)</td>
<td>*FIX (§5)</td>
<td>*FIX (§6)</td>
<td>*FIX (§7)</td>
<td>*FIX (§7)</td>
</tr>
</tbody>
</table>

Table 3.1: Overview of the specializations of *FIX

In this section, I give an overview of the four specializations of *FIX that I have designed for the repair of different types of presentation failures discussed in Chapter 2. The four specializations are explained in detail in Chapters 4, 5, 6 and 7. In each of these chapters, I summarize the input functions D and L for completeness and discuss the repair approach based on *FIX in detail. Table 3.1 shows a summary of the specializations and the techniques used for providing the D and L inputs. The cells highlighted in gray represent the techniques developed by me.

3.2.1 *FIX: Repair of Layout Cross Browser Issues (XBIs)

I designed an approach, *FIX, for the automated repair of layout XBIs in web pages. *FIX utilizes two phases of guided search to find the best repair. The first search finds one or more candidate fixes for each XBI by quantitatively comparing the layout similarity of the page rendered in different browsers via a fitness function. The second search then seeks to find an optimal combination of candidate fixes identified in the first phase to produce an overall best repair by leveraging a measurement of the number of layout XBIs detected in a page. The empirical evaluation of *FIX on 15 real world web pages showed that it was able to resolve 86% of the
layout XBIs reported by X-PERT, and 99% of the layout XBIs observable by humans. The results demonstrate that my approach is potentially of high use to developers by providing automated fixes for layout XBIs. I provide more details about \textit{XFix} in Chapter 4.

### 3.2.2 \textit{MFix}: Repair of Mobile Friendly Problems (MFPs)

I designed an approach, \textit{MFix}, to automatically generate CSS patches that can repair MFPs in a web page. A unique challenge relevant to the MFP problem domain is that a repair must maintain the page’s original layout as faithfully as possible. This requires fixing MFPs while maintaining, where possible, the relative proportions and positioning of elements that are related to one another on the page. To do this, \textit{MFix} first segments the page, i.e., identifies elements that form natural visual groupings on the page, and then builds a property dependence graph for each segment to adjust elements within a segment in synchronization with each other. The approach then builds graph-based models of the layout of a web page and uses constraints encoded by these graphs to compute patches that can improve mobile friendliness while minimizing layout disruption. To identify the best patch efficiently, \textit{MFix} leverages unique aspects of the problem domain to quantify metrics related to layout distortion and parallelize the computation of the solution. The empirical evaluation of \textit{MFix} on 38 popular websites listed in the Alexa Top 50 most visited websites showed that it could effectively resolve mobile friendly problems for 95% of the subjects. I also evaluated the results with a user study, in which participants overwhelmingly preferred the repaired version of the website for use on mobile devices and also considered the repaired page to be more readable than the original. Chapter 5 discusses the \textit{MFix} approach in more detail.

### 3.2.3 \textit{IFix}: Repair of Internationalization Presentation Failures (IPFs)

I designed an approach, \textit{IFix}, for automatically repairing IPFs in web pages. Repairing IPFs is challenging as any kind of style change to one element must be mirrored in stylistically related elements to maintain the aesthetic consistency of the page. To address this challenge, I designed a novel clustering technique that identifies groupings of elements that are stylistically similar and adjusts them together in order to maintain the visual consistency of the page. To find repairs, I devised a guided search-based technique that quantifies the amount of distortion in a page by leveraging an existing IPF detection technique, GWALI [49], and UI change metrics. The empirical evaluation showed that my approach was able to successfully resolve 98% of the reported IPFs for 23 real-world web pages. In a user study of the repaired web pages, I found that the repairs met
with high user approval, with over 70% of the user responses rating the repaired pages as better than the faulty versions. I explain the $\text{IFix}$ approach in more detail in Chapter 6.

### 3.2.4 $\text{GFix}$: Repair of Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs)

I designed a novel automated approach, $\text{GFix}$, for repairing MDDPs and RDPs in web pages. $\text{GFix}$ uses guided search-based techniques to automatically find repairs for the MDDPs and RDPs detected in web pages. As its fitness function, $\text{GFix}$ uses computer-vision techniques to quantify the amount of human perceptible differences between the actual appearance of a web page and its intended appearance, with the aim of minimizing the visual differences. In the evaluation of the approach on a set of real-world web applications, I found that the approach was able to accurately and quickly identify repairs for the failures. I provide more details of the $\text{GFix}$ approach in Chapter 7.
Chapter 4

XF\textsuperscript{i}x: Repair of Layout Cross Browser Issues (XBIs)

The constantly increasing number of web browsers with which users can access a website has introduced new challenges in preventing appearance related issues. Differences in how various browsers interpret HTML and CSS standards can result in Cross Browser Issues (XBIs) — inconsistencies in the appearance or behavior of a website across different browsers. Although XBIs can impact the appearance or functionality of a website, the vast majority — over 90% — result in appearance related problems [137]. This makes XBIs a significant challenge in ensuring the correct and consistent appearance of a website’s User Interface (UI).

Despite the importance of XBIs, their detection and repair poses numerous challenges for developers. First, the sheer number of browsers available to end users is large — an informal listing reports that there are over 115 actively maintained and currently available [33]. Developers must verify that their websites render and function consistently across as many of these different browsers and platforms as possible. Second, the complex layouts and styles of modern web applications make it difficult to identify the UI elements responsible for the observed XBI. Third, developers lack a standardized way to address XBIs and generally have to resolve XBIs on a case by case basis. Fourth, for a repair, developers must modify the problematic UI elements without introducing new XBIs. Predictably, these challenges have made XBIs an ongoing topic of concern for developers. A simple search on StackOverflow — a popular technical forum — with the search term “cross browser” results in over 23,000 posts discussing ways to resolve XBIs, of which approximately 7,000 are currently active questions [43].

Tool support to help developers repair XBIs is limited in terms of capabilities. Although tools such as Firebug [22] can provide useful information, developers still require expertise to manually analyze the XBIs (which involves determining which HTML elements to inspect, and understanding the effects of the various CSS properties defined for them), and then repair them by performing the necessary modifications so that the page renders correctly. XBI-oriented techniques
from the research community (e.g., X-PERT [137, 59, 141] and Browserbite [144]) are only able to
\textit{detect and localize} XBIs (i.e., they address the first two of the four previously listed challenges),
but are incapable of \textit{repairing} XBIs so that a web page can be “fixed” to provide a consistent
appearance across different browsers.

To address these limitations, I propose a novel search-based approach that enables automatic
generation of fixes for a significant class of appearance related XBIs. The XBIs targeted by
my approach are known as \textit{layout XBIs} (also referred to as “structure XBIs” by Choudhary et
al. [137]), which collectively refer to any XBI that relates to an inconsistent layout of HTML
elements in a web page when viewed in different browsers. Layout XBIs appear in over 56% of
the websites manifesting XBIs [137]. My key insight is that the impact of layout XBIs can
be quantified by a fitness function capable of guiding a search to a repair that minimizes the
number of XBIs present in a page. I implemented this search-based approach as a tool, XFix,
and evaluated it on 15 real world web pages containing layout XBIs. XFix was able to resolve 86% of
the XBIs reported by X-PERT [137], and 99% of the XBIs observed by humans. The results
therefore demonstrate that my approach is potentially of high use to developers by providing
automated fixes for problematic web pages involving layout XBIs.

4.1 Background and Example

In this section I provide background information that details why layout XBIs occur, what the
common practices are to repair them, and introduces an illustrative example.

\textbf{Layout XBIs}

Inconsistencies in the way browsers interpret the semantics of the Document Object Model (DOM)
and Cascading Style Sheets (CSS) can cause layout XBIs\footnote{Hereafter, layout XBIs is simply referred as XBIs.} — differences in the rendering of an
HTML page between two or more browsers. These inconsistencies tend to arise from different
interpretations of the HTML and CSS specifications, and are not per se, faults in the browsers
themselves [5]. Additionally, some browsers may implement new CSS properties or existing prop-
erties differently in an attempt to gain an advantage over competing browsers [113].

\textbf{Fixing Layout XBIs}

When a layout XBI has been detected, developers may employ several strategies to adjust its
appearance. For example, changing the HTML structure, replacing unsupported HTML tags, or
adjusting the page’s CSS. My approach targets XBIs that can be resolved by finding alternate
values for a page’s CSS properties. There are two significant challenges to carrying out this type of repair. First, the appearance (e.g., size, color, font style) of any given set of HTML elements in a browser is controlled by a series of complex interactions between the page’s HTML elements and CSS properties, which means that identifying the HTML elements responsible for the XBI is challenging. Second, assuming that the right set of elements can be identified, each element may have dozens of CSS properties that control its appearance, position, and layout. Each of these properties may range over a large domain. This makes the process of identifying the correct CSS properties to modify and the correct alternate values for those properties a labor intensive task.

Once the right alternate values are identified, developers can use browser-specific CSS qualifiers to ensure that they are used at runtime. These qualifiers direct the layout engine to use the provided alternate values for a CSS property when it is rendered on a specific browser [7, 6]. This approach is widely employed by developers. In my analysis of the top 480 websites (see Section 4.3), I found that 79% employed browser-specific CSS to ensure a consistent cross browser appearance. In fact, web developers typically maintain an extensive list of browser specific styling conditions [6] to address the most common XBI.

Example XBI and Repair

Figure 4.1 shows screenshots of the menu bar of one of the evaluation subjects, IncredibleIndia, as rendered in Internet Explorer (IE) (Figure 4.1a) and Firefox (Figure 4.1b). As can be seen, an XBI is present in the menu bar, where the text of the navigational links is unreadable in the Firefox browser (Figure 4.1b).

An excerpt of the HTML and CSS code that defines the navigation bar is shown in Listing 4.1. To resolve the XBI, an appropriate value for the margin-top or padding-top CSS property needs to be found for the HTML element corresponding to the navigation bar to push it down and into view. In this instance, the fix is to add “margin-top: 1.7%” to the CSS for the Firefox version. The inserted browser-specific code is shown in the red box in Listing 4.1. The “-moz” prefixed
selector declaration directs the layout engine to only use the included value if the browser type is Firefox (i.e., Mozilla), and other browsers’ layout engines will ignore this code.

Listing 4.1: HTML and CSS excerpt of the IncredibleIndia example shown in Figure 4.1. The highlighted section (lines 6–10) represents the fix added to the CSS to address the XBI.

This particular example was chosen because the fix is straightforward and easy to explain. However, most XBIs are much more difficult to resolve. Typically multiple elements may need to be adjusted, and for each one multiple CSS properties may also need to be modified. A fix itself may introduce new XBIs, meaning that several alternate fixes may need to be considered.

4.2 Specialization of the Generalized Approach, *Fix*

In this section, I provide details of my approach for repairing layout XBIs in web pages that is based on the generalized repair approach, *Fix, explained in Chapter 3. The two prerequisites for repairing presentation failures, Detection and Localization, can be instantiated using existing Cross Browser Testing (XBT) techniques, such as X-PERT [137]. For completeness, I summarize X-PERT’s detection and localization algorithm in Section 4.2.1. My contribution is in developing a repair approach for finding suitable fixes for layout XBIs in web pages using search-based techniques. Section 4.2.2 discusses this approach in more detail.
4.2.1 Detection and Localization of Layout XBIs

For implementing the Detection and Localization phases of the debugging process, I use the X-PERT tool [137], which is a well-known XBI oriented technique. X-PERT uses DOM differencing techniques to detect and localize relative-layout XBIs in web applications. For completeness, I provide a synopsis of X-PERT’s detection and localization algorithm and its evaluation results.

X-PERT first captures the layout of a page rendered in a browser as a directed graph, where the vertices are HTML elements in the page and an edge exists between two vertices only if they represent a parent-child or sibling relationship with each other. The edges are further qualified with attributes specifying the relative position of the two vertices with respect to each other, such as left-align, center-align, and above. These attributes are computed by comparing the Minimum Bounding Rectangles (MBRs) of the vertices.

After building the layout models for the page in the two browsers, X-PERT performs a heuristic-based matching to find corresponding vertices in the two layout models. It then systematically compares each edge in the matched set of vertices to find discrepancies in the edge attributes. If a discrepancy is found, X-PERT reports this as a layout XBI, represented by a tuple of the form \( \langle \text{label}, \langle e_1, e_2 \rangle \rangle \). Here \( e_1 \) and \( e_2 \) are the XPaths of the two HTML elements of the page that are rendered differently in the two browsers, and \( \text{label} \) is the discrepant edge attribute that denotes the layout position of \( e_1 \) and \( e_2 \) in one browser that was violated in the other browser. For example, \( \langle \text{top-align}, e_1, e_2 \rangle \) indicates that \( e_1 \) is pinned to the top edge of \( e_2 \) in one browser, but not in the other browser.

In the evaluation on 14 real-world subject web applications, X-PERT was found to be highly accurate — 76% precision and 95% recall, on average. X-PERT also provided significantly better results when compared with the state-of-the-art XBI detection tool, CrossCheck [59], which had an average precision of 18% and recall of 83%.

4.2.2 Repair of Layout XBIs

For repairing layout XBIs in web pages, I propose a novel automated approach using search-based techniques based on *Fix. The placement of a web page’s UI elements is controlled by the page’s HTML elements and CSS properties. Therefore to resolve the XBIs, my approach attempts to find new values for CSS properties that can make the faulty appearance match the correct appearance as closely as possible. In the remainder of this section, I discuss the overall algorithm of my approach followed by details about each of the different stages in the approach.
Formally, XBIs are due to one or more HTML-based root causes. A root cause is a tuple \(\langle e, p, v \rangle\), where \(e\) is an HTML element in the page, \(p\) is a CSS property of \(e\), and \(v\) is the value of \(p\). Given a set of XBIs \(X\) for a page PUT and a set of potential root causes, my approach seeks to find a set of fixes that resolve the XBIs in \(X\). I define a fix as a tuple \(\langle r, v' \rangle\), where \(r\) is a root cause and \(v'\) is the suggested new value for \(p\) in the root cause \(r\). A set of XBI-resolving fixes is referred as a repair.

The approach generates repairs using guided search-based techniques [80, 60]. Two aspects of the XBI repair problem motivate this choice of technique. The first is that the number of possible repairs is very large, since there can be multiple XBIs present in a page, each of which may have several root causes, and for which the relevant CSS properties range over a large set of possible values. Second, fixes made for one particular XBI may interfere with those for another, or, a fix for any individual XBI may itself cause additional XBIs, requiring a tradeoff to be made among possible fixes. Search-based techniques are ideal for this type of problem because they can explore large solution spaces intelligently and efficiently, while also identifying solutions that effectively balance a number of competing constraints. Furthermore, the visual manifestation of XBIs also lends itself to quantification via a fitness function, which is a necessary element for a search-based technique. A fitness function computes a numeric assessment of the “closeness” of candidate solutions found during the search to the solution ultimately required. My insight is that a good fitness function can be built that leverages a measurement of the number of XBIs detected in a PUT, by using well-known XBI detection techniques, and the similarity of the layout of the PUT when rendered in the reference and test browsers, by comparing the size and positions of the bounding boxes of the HTML elements involved in each XBI identified.

The approach works by first detecting XBIs in a page and identifying a set of possible root causes for those XBIs. Then the approach utilizes two phases of guided search to find the best repair. The first search takes the CSS property of each root cause and tries to find a new value for it that is most optimal with respect to the fitness function. This optimized property value is referred to as a candidate fix. The second search then seeks to find an optimal combination of candidate fixes identified in the first phase. This additional search is necessary since not all candidate fixes may be required, as the CSS properties involved may have duplicate or competing effects. For instance, the CSS properties margin-top and padding-top may both be identified as root causes for an XBI, but can be used to achieve similar outcomes — meaning that only one may actually need to be included in the repair. Conversely, other candidate fixes may be required to be used in combination with one another to fully resolve an XBI. For example, an HTML element may need to be adjusted for both its width and height. Furthermore, candidate fixes
produced for one XBI may have knock-on effects on the results of candidate fixes for other XBIs, or even introduce additional and unwanted XBIs. By searching through different combinations of candidate fixes, the second search aims to produce a suitable subset — a repair — that resolves as many XBIs as possible for a page when applied together.

I now introduce the steps of this approach in more detail, beginning with an overview of the complete algorithm.

### 4.2.2.1 Overall Algorithm

The top level algorithm of the approach is shown by Algorithm 1. Four inputs are required: the page under test, $PUT$, which exhibits XBIs. The $PUT$ is obtained via a URL that points to a location on the file system or network that provides access to all of the necessary HTML, CSS, Javascript, and media files for rendering $PUT$. The second input is the reference browser, $R$, that shows the correct rendering of $PUT$. The third input is the test browser, $T$, in which the rendering of $PUT$ shows XBIs with respect to $R$. The fourth input is a detection ($D$) and localization ($L$) function that can identify and report XBIs. For this input, we use the X-PERT tool [137] (summarized in Section 4.2.1). The output of the approach is a page, $PUT'$, a repaired version of $PUT$.

The overall algorithm, shown as Algorithm 1, comprises five stages, as shown by the overview diagram in Figure 4.2. The diagram also shows the instantiations of the different abstraction points (AP1–4) of the $\lambda$Fix approach.

**Stage 1 — Initial XBI Detection**

The initial part of the algorithm (lines 1–4) involves obtaining the set of XBIs $X$ when $PUT$ is rendered in $R$ and $T$. To identify XBIs, we use the input $D$ and $L$ function X-PERT tool [137], which is represented by the “$DL$” function called on line 2. X-PERT returns a set of identified XBIs, $X$, in which each XBI is represented by a tuple of the form $(label, \langle e_1, e_2 \rangle)$, where $e_1$ and
Algorithm 1 Overall Algorithm

Input: \( PUT \): Web page under test
\[ \begin{align*}
R & : \text{Reference browser} \\
T & : \text{Test browser}
\end{align*} \]
\( DL \): Detection and Localization function for XBIs

Output: \( PUT' \): Modified \( PUT \) with repair applied

1: /* Stage 1 — Initial XBI Detection */
2: \( X \leftarrow DL(PUT, R, T) \)
3: \( DOM_R \leftarrow \) buildDOMTree\((PUT, R)\)
4: \( DOM_T \leftarrow \) buildDOMTree\((PUT, T)\)
5: while true do

6: /* Stage 2 — Extract root causes */
7: \( rootCauses \leftarrow \{\} \)
8: for each \( \langle label, \langle e_1, e_2 \rangle \rangle \in X \) do
9: \( props \leftarrow \) getCSSProperties\((label)\)
10: for each \( p \in props \) do
11: \( v_1 \leftarrow \) getValue\((e_1, p, DOM_T)\)
12: \( rootCauses \leftarrow rootCauses \cup \langle e_1, p, v_1 \rangle \)
13: \( v_2 \leftarrow \) getValue\((e_2, p, DOM_T)\)
14: \( rootCauses \leftarrow rootCauses \cup \langle e_2, p, v_2 \rangle \)
15: end for
16: end for

17: /* Stage 3 — Search for Candidate Fixes */
18: \( candidateFixes \leftarrow \{\} \)
19: for each \( \langle e, p, v \rangle \in rootCauses \) do
20: \( candidateFix \leftarrow \) searchForCandidateFix\((\langle e, p, v \rangle, PUT, DOM_R, T)\)
21: \( candidateFixes \leftarrow candidateFixes \cup candidateFix \)
22: end for

23: /* Stage 4 — Search for Best Combination of Candidate Fixes */
24: \( repair \leftarrow \) searchForBestRepair\((candidateFixes, PUT, R, T)\)

25: /* Stage 5 — Check Termination Criteria */
26: \( PUT' \leftarrow applyRepair(PUT, repair) \)
27: \( X' \leftarrow DL(PUT', R, T) \)
28: if \( X' = \emptyset \) or \( |X'| = X \) then
29: return \( PUT' \)
30: else if \( |X'| > |X| \) then
31: return \( PUT \)
32: else
33: \( X \leftarrow X' \)
34: \( PUT \leftarrow PUT' \)
35: \( DOM_T \leftarrow \) buildDOMTree\((PUT', T)\)
36: end if
37: end while
$e_2$ are the XPaths of the two HTML elements of the PUT that are rendered differently in $T$ versus $R$, and label is a descriptor that denotes the original (correct) layout position of $e_1$ that was violated in $T$. For example, \langle top-align, e_1, e_2 \rangle indicates that $e_1$ is pinned to the top edge of $e_2$ in $R$, but not in $T$. After identifying the XBIs, the algorithm then enters its main loop, which comprises Stages 2–5.

**Stage 2 — Extract Root Causes**

The second stage of the algorithm (lines 7–16) extracts the root causes relevant to each XBI. The key step in this stage identifies CSS properties relevant to the XBI’s label (shown as “getCSSProperties” at line 9). For example, for the top-align label, the CSS properties margin-top and top can alter the top alignment of an element with respect to another and would therefore be identified in this stage. I identified this mapping through analysis of the CSS properties and it holds true for all web applications without requiring developer intervention. Each relevant CSS property forms the basis of two root causes, one for $e_1$, and one for $e_2$. These are added to the running set rootCauses, with the values of the CSS properties extracted for each element ($v_1$ and $v_2$ respectively) extracted from the DOM of the PUT when it is rendered in $T$ (lines 9 and 11).

**Stage 3 — Search for Candidate Fixes**

Comprising the first phase search, this stage produces individual candidate fixes for each root cause (lines 18–22). The fix is a new value for the CSS property that is optimized according to a fitness function, with the aim of producing a value that resolves, or is as close as possible to resolving the layout deviation. This optimization process occurs in the “searchForCandidateFix” procedure, which is described in detail in Section 4.2.2.2.

**Stage 4 — Search for the Best Combination of Candidate Fixes**

Comprising the second phase search, the algorithm makes a call to the “searchForBestRepair” procedure (line 24) that takes the set of candidate fixes in order to find a subset, repair, representing the best overall repair. This procedure is described in Section 4.2.2.3.

**Stage 5 — Check Termination Criteria**

The final stage of the algorithm (lines 26–36) determines whether the algorithm should terminate or proceed to another iteration of the loop and two-phase search. Initially, the fixes in the set repair are applied to a copy of PUT by adding test browser ($T$) specific CSS code to produce a modified version of the page PUT’ (line 26). The approach identifies the set of XBIs, $X'$
for $PUT'$, using the technique explained in Section 4.2.1, which is represented by the “getXBIs” function called on line 27.

Ideally, all of the XBIs in $PUT$ will have been resolved by this point, and $X'$ will be empty. If this is the case, the algorithm returns the repaired page $PUT'$. If the set $X'$ is identical to the original set of XBIs $X$ (originally determined on line 2), the algorithm has made no improvement in this iteration of the algorithm, and so the $PUT'$ is returned, having potentially only been partially fixed as a result of the algorithm rectifying a subset of XBIs in a previous iteration of the loop.

If the number of XBIs has increased, the current repair introduces further layout deviations. In this situation, $PUT$ is returned (which may reflect partial fixes from a previous iteration of the loop, if there were any). However, if the number of XBIs has been reduced, the current repair represents an improvement that may be improved further in another iteration of the algorithm.

Broadly, there are two scenarios under which the approach could fail: (1) X-PERT does not initially include the faulty HTML element in $X$; or (2) the search does does not identify an acceptable fix, which could happen due to the non-determinism of the search.

### 4.2.2.2 Search for Candidate Fixes

The first search phase (represented as the procedure “searchForCandidateFix”) focuses on each potential root cause $⟨e, p, v⟩$ in isolation of the other root causes, and attempts to find a new value $v'$ for the root cause that improves the similarity of the page when rendered in the reference browser $R$ and the test browser $T$. Guidance to this new value is provided by a fitness function that quantitatively compares the relative layout discrepancies between $e$ and the elements that surround it when $PUT$ is rendered in $R$ and $T$. I begin by giving an overview of the search algorithm used, and then explain the fitness function employed.

**Search Algorithm**

The inputs to the search for a candidate fix are the page under test, $PUT$, the test browser, $T$, the DOM tree from the reference browser, $DOM_R$, and the root cause tuple, $⟨e, p, v⟩$. The search attempts to find a new value, $v'$, for $p$ in the root cause. The search process used to do this is inspired by the variable search component of the Alternating Variable Method (AVM) [82, 84], and specifically the use of “exploratory” and “pattern” moves to optimize variable values. The aim of exploratory moves is to probe values neighboring the current value of $v$ to find one that improves fitness when evaluated with the fitness function. Exploratory moves involve adding small delta values (i.e., $[-1,1]$) to $v$ and observing the impact on the fitness score. If the fitness
Algorithm 2 Fitness Function for Candidate Fixes

Input: e: XPath of HTML element under analysis
p: CSS property of HTML element, e
v: Value of CSS property, p
PUT: Web page under test
DOMR: DOM tree of PUT rendered in R
T: Test browser

Output: fitness: Fitness value of the hypothesized fix ⟨e, p, v⟩

1: PUT ← applyValue(e, p, v, PUT)
2: DOMT ← buildDOMTree(PUT, T)
3: /* Component 1 — Difference in location of e with respect to R and T */
4: ⟨x1, y1, x2, y2⟩ ← getBoundingBox(DOMT, e)
5: ⟨x1, y1, x2, y2⟩ ← getBoundingBox(DOMR, e)
6: DTL ← √((x2 - x1)2 + (y2 - y1)2)
7: DBR ← √((x2 - x1)2 + (y2 - y1)2)
8: ∆pos ← DTL + DBR
9: /* Component 2 — Difference in size of e with respect to R and T */
10: widthR ← x2 - x1
11: widthT ← x2 - x1
12: heightR ← y2 - y1
13: heightT ← y2 - y1
14: ∆size ← |widthR - widthT| + |heightR - heightT|
15: /* Component 3 — Differences in locations of neighboring elements of e */
16: neighborsT ← getNeighbors(e, DOMT, Nr)
17: ∆npos ← 0
18: for each n ∈ neighborsT do
19: n' ← getMatchingElement(n, DOMR)
20: ⟨x1, y1, x2, y2⟩ ← getBoundingBox(DOMT, n)
21: ⟨x1, y1, x2, y2⟩ ← getBoundingBox(DOMR, n')
22: DTL ← √((x2 - x1)2 + (y2 - y1)2)
23: DBR ← √((x2 - x1)2 + (y2 - y1)2)
24: ∆pos ← DTL + DBR
25: ∆npos ← ∆npos + ∆pos
26: end for
27: /* Compute final fitness value */
28: fitness ← (w1 * ∆pos) + (w2 * ∆size) + (w3 * ∆npos)
29: return fitness
is observed to be improved, pattern moves are made in the same “direction” as the exploratory
move to accelerate further fitness improvements through step sizes that increase exponentially. If
a pattern move fails to improve fitness, the method establishes a new direction from the current
point in the search space through further exploratory moves. If exploratory moves fail to yield a
new direction (i.e., a local optima had been found), this value is returned as the best candidate fix
value. The fix tuple, \( \langle e, p, v, v' \rangle \), is then returned to the main algorithm (line 20 of Algorithm 1).

Fitness Function

The fitness function for producing a candidate fix is shown by Algorithm 2. The goal of the fitness
function is to quantify the relative layout deviation for \textit{PUT} when rendered in \textit{R} and \textit{T} following
the change to the value of a CSS property for an HTML element. Given the element \( e \) in \textit{PUT},
the fitness function considers three components of layout deviation between the two browsers: (1)
the difference in the location of \( e \); (2) the difference in the size of \( e \); and (3) any differences in the
location of \( e \)'s neighbors. Figure 4.3 shows a diagrammatic representation of these components.
Rectangles with a solid background correspond to the bounding boxes of elements rendered in \textit{R}
and the rectangles with diagonal lines correspond to the bounding boxes of elements rendered in
\textit{T}. Intuitively, all three components should be minimized as the evaluated fixes make progress
towards resolving an XBI without introducing any new differences or introducing further XBIs
for \( e \)'s neighbors. The fitness function for an evaluated fix is the weighted sum of these three
components.

The first component, location difference of \( e \), is computed by lines 3–8 of Algorithm 2, and
assigned to the variable \( \Delta_{pos} \). This value is calculated as the sum of the Euclidean distance
between the top-left (TL) and bottom-right (BR) corners of the bounding box of \( e \) when it is
rendered in \textit{R} and \textit{T}. The bounding box is obtained from the DOM tree of the page for each
browser.

The second component, difference in size of \( e \), is calculated by lines 10–14 of the algorithm,
and is assigned to the variable \( \Delta_{size} \). The value is calculated as the sum of the differences of \( e \)'s
width and height when rendered in \textit{R} and \textit{T}. The size information is obtained from the bounding
box of \( e \) obtained from the DOM tree of the page in each browser.

The third and final component of the fitness function, finding the location difference of \( e \)'s
neighbors occurs on lines 16–26 of the algorithm, and is assigned to the variable \( \Delta_{npos} \). The
neighbors of \( e \) are the set of HTML elements that are within \( N_r \) hops from \( e \) in \textit{PUT}’s DOM tree
as rendered in \textit{T}. For example, if \( N_r = 1 \), then the neighbors of \( e \) are its parents and children. If
\( N_r = 2 \), then the neighbors are its parent, children, siblings, grandparent, and grandchildren. For
(a) Component 1: $\Delta_{pos} = D_{TL} + D_{BR}$, where $D_{TL}$ and $D_{BR}$ is the Euclidean distance between the top left (TL) and bottom right (BR) corners, respectively, of $e$ rendered in $R$ and $T$. $\Delta_{pos}$ decreases as the boxes move closer.

(b) Component 2: $\Delta_{size} = |w_R - w_T| + |h_R - h_T|$, where $w_R$ and $h_R$ are the respective width and height of $e$ rendered in $R$, and $w_T$ and $h_T$ are the respective width and height of $e$ rendered in $T$. $\Delta_{size}$ decreases as the boxes become similar in size.

(c) Component 3: $\Delta_{npos} = D_{TL} + D_{BR}$, where $D_{TL}$ and $D_{BR}$ is the Euclidean distance between the top left (TL) and bottom right (BR) corners, respectively, of $e$’s neighbor $n$ rendered in $R$ and $T$. $\Delta_{npos}$ decreases as $e$’s boxes move closer, which causes $n$’s boxes to also move closer.

Figure 4.3: Fitness function components of the XBI repair approach
each neighbor, the approach finds its corresponding element in the DOM tree of PUT rendered in $R$ and calculates $\Delta_{pos}$ for each pair of elements.

The final fitness value is then formed from the weighted sum of the three components $\Delta_{pos}$, $\Delta_{size}$, and $\Delta_{npos}$ (line 28).

4.2.2.3 Search for the Best Combination of Candidate Fixes

The goal of the second search phase (represented by a call to “searchForBestRepair” at line 24 of Algorithm 1) is to identify a subset of $candidateFixes$ that together minimize the number of XBIs reported for the PUT. This step is included in the approach for two reasons. Firstly, a fix involving one particular CSS property may only be capable of partially resolving an XBI and may need to be combined with another fix to fully address the XBI. Furthermore, the interaction of certain fixes may have emergent effects that result in further unwanted layout problems. For example, suppose a submit button element appears below, rather than to the right of a text box. Candidate fixes will address the layout problem for each HTML element individually, attempting to move the textbox down and to the left, and the button up and to the right. Taking these fixes together will result in the button appearing to the top right corner of the text box, rather than next to it. Identifying a selection of fixes, a candidate repair, that avoids these issues is the goal of this phase. To guide this search, I use the number of XBIs that appear in the PUT after the candidate repair has been applied.

The search begins by evaluating a candidate repair with a single fix — the candidate fix that in the first search phase produced the largest fitness improvement. Assuming this does not eradicate all XBIs, the search continues by generating new candidate repairs in a biased random fashion. Candidate repairs are produced by iterating through the set of fixes. A fix is included in the repair with a probability $imp_{fix}/imp_{max}$, where $imp_{fix}$ is the improvement observed in the fitness score when the fix was evaluated in the first search phase divided by the maximum improvement observed over all of the fixes in $candidateFixes$. Each candidate repair is evaluated for fitness in terms of the number of resulting XBIs, with the best repair retained. A history of evaluated repairs is maintained, so that any repeat solutions produced by the biased random generation algorithm are not re-evaluated.

The random search terminates when (a) a candidate repair is found that fixes all XBIs, (b) a maximum threshold of candidate repairs to be tried has been reached, or (c) the algorithm has produced a sequence of candidate repairs with no improvement in fitness.
4.3 Evaluation

I conducted empirical experiments to assess the effectiveness and efficiency of my repair approach for resolving XBIs, with the aim of answering the following four research questions:

**RQ1:** How effective is the approach at reducing layout XBIs?

**RQ2:** What is the impact on the cross-browser consistency of the page when the suggested repairs are applied?

**RQ3:** How long does the approach take to find repairs?

**RQ4:** How similar in size are the approach generated repair patches to the browser-specific code present in real-world websites?

4.3.1 Implementation

I implemented the approach in a prototype tool in Java, named “XFix” [69]. XFix is a standalone tool that exposes a simple API to specify inputs and run the repair technique. To facilitate easy management of third-party dependencies, XFix is packaged as a Maven project. XFix can be run on any platform, such as Windows, Linux, and macOS. Since XFix analyzes the client side code of the page under test, it is agnostic to the server side technology used.

![High-level overview of the XFix tool](image)

**Figure 4.4:** High-level overview of the XFix tool

Figure 4.4 shows a high-level overview of XFix with the different stages explained in Section 4.2.2 shown in italics. XFix accepts the page under test (PUT) input in the form of a URL pointing to the location of the HTML page on the file system where all the CSS, Javascript, and media necessary for rendering the PUT can be accessed. For R and T, XFix provides an option to select a browser from XFix’s supported set of browsers. XFix currently supports all the versions of the three most widely used browsers, Firefox, Chrome, and IE. More browsers can be easily added to XFix by including the browser’s standalone server implementation of the Selenium WebDriver’s wire protocol. Note that R and T need to be pre-installed on the user’s computer.
I leveraged Javascript and the Selenium WebDriver library for making dynamic changes to web pages, such as applying candidate fix values. For identifying the set of layout XBIs, I used the latest publicly available version [58] of the well-known XBI detection tool, X-PERT [137, 139]. I made minor changes to the publicly available version to fix bugs and add accessor methods for data structures. I used this modified version throughout the rest of the evaluation. Details of the changes made to X-PERT can be found on the XFix project page (https://github.com/sonalmahajan/xfix). The fitness function parameters for the search of candidate fixes discussed in Section 4.2.2.2 are set as: $N_r = 2$, and $w_1 = 1$, $w_2 = 2$, and $w_3 = 0.5$ for the weights for $\Delta_{pos}$, $\Delta_{size}$, and $\Delta_{npos}$ respectively. (The weights assigned prioritize $\Delta_{size}$, $\Delta_{pos}$ and $\Delta_{npos}$ in that order. I deemed size of an element as most important, because of its likely impact on all three components, followed by location, which is likely to impact its neighbors.) For the termination conditions (b) and (c) of the search for the best combination of candidate fixes (Section 4.2.2.3), the maximum threshold value is set to 50 and the sequence value is set to 10.

Figure 4.5: Example of repair.css generated by XFix

Upon termination, XFix generates a repair.css file (shown in Figure 4.5 for the IncredibleIndia example of Figure 4.1) containing the repair patch and modifies the PUT file to include repair.css in the ⟨head⟩ section of the HTML of the page. Note that a new repair.css is created with a timestamp appended to the name for every run of XFix to effectively resolve XBIs across different test browsers for the PUT. XFix generates the repair.css as follows. First, XFix adds a browser specific qualifier corresponding to the test browser (e.g., -moz for Firefox) as shown in line 1 of Figure 4.5. Such qualifiers direct the layout engine to use the provided alternate values for the CSS property when it is rendered on a specific browser. For example, the repair patch shown in Figure 4.5 is only applied if the browser type is Firefox, and is ignored by the layout engines of other browsers. Then, for each fix tuple $(e, p, v, v')$ in the repair set identified in stage 4, XFix converts the XPath of the element $e$ to a CSS selector and adds it to the browser specific qualifier block. For example, line 2 shows the CSS selector derived from the XPath, /html/body/div[3]/div/div. XFix then converts the fix value $v'$, which is an absolute
value (e.g., margin-top:20px), to a relative fix value with respect to an element’s parent’s dimensions, such as margin-top:1.7%. AFix then adds this relative fix value for the CSS property $p$ (line 3).

4.3.2 Subjects

For the evaluation 15 real-world subjects were used as listed by Table 4.1. The columns labeled “#HTML” and “#CSS” report the total number of HTML elements present in the DOM tree of a subject, and the total number of CSS properties defined for the HTML elements in the page respectively. These metrics of size give an estimate of a page’s complexity in debugging and finding potential fixes for the observed XBIs. The “Ref” column indicates the reference browser in which the subject displays the correct layout; while the column “Test” refers to the browser in which the subject shows a layout XBI. In these columns, “CH”, “FF”, and “IE” refer to the Chrome, Firefox, and Internet Explorer browsers respectively.

I collected the subjects from three sources: (1) websites used in the evaluation of X-PERT [137], (2) my prior interaction with websites exhibiting XBIs, and (3) the random URL generator, UROULETTE [39]. The “GrantaBooks” subject came from the first source. The other subjects from X-PERT’s evaluation could not be used because their GUI had been reskinned or the latest version of the IE browser now rendered the pages correctly. The “HotwireHotel” subject was chosen from the second source, and the remaining thirteen subjects were gathered from the third source.

The goal of the selection process was to select subjects that exhibited human perceptible layout XBIs. I did not use X-PERT for an initial selection of subjects because it was found that it reported many subjects with XBIs that were difficult to observe. For selecting the subjects, I used the following process: (1) render the page, PUT, in the three browser types; (2) visually inspect the rendered PUT in the three browsers to find layout XBIs; (3) if layout XBIs were found in the PUT, select the browser showing a layout problem, such as overlapping, wrapping, or distortion of content, as the test browser, and one of the other two browsers showing the correct rendering as the reference browser; (4) try to manually fix the PUT by using the developer tools in browsers, such as Firebug for Firefox, and record the HTML elements to which the fix was applied; (5) run X-PERT on the PUT with the selected reference and test browsers; and (6) use the PUT as a subject, if the manually recorded fixed HTML elements were present in the set of elements reported by X-PERT. I included steps 4–6 in the selection process to ensure that if X-PERT reported false negatives, they would not bias the evaluation results.
Table 4.1: Subjects used in the evaluation of XFix

<table>
<thead>
<tr>
<th>Name</th>
<th>URL</th>
<th>#HTML</th>
<th>#CSS</th>
<th>Ref</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenjaminLees</td>
<td><a href="http://www.benjaminlees.com">http://www.benjaminlees.com</a></td>
<td>317</td>
<td>1,525</td>
<td>CH</td>
<td>FF</td>
</tr>
<tr>
<td>Bitcoin</td>
<td><a href="https://bitcoin.org/en/">https://bitcoin.org/en/</a></td>
<td>207</td>
<td>1,957</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>Eboss</td>
<td><a href="http://www.e-boss.gr">http://www.e-boss.gr</a></td>
<td>439</td>
<td>789</td>
<td>IE</td>
<td>FF</td>
</tr>
<tr>
<td>EquilibriumFans</td>
<td><a href="http://www.equilibriumfans.com">http://www.equilibriumfans.com</a></td>
<td>340</td>
<td>868</td>
<td>CH</td>
<td>FF</td>
</tr>
<tr>
<td>GrantaBooks</td>
<td><a href="http://grantabooks.com">http://grantabooks.com</a></td>
<td>325</td>
<td>6,545</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>HenryCountyOhio</td>
<td><a href="http://www.henrycountyohio.com">http://www.henrycountyohio.com</a></td>
<td>300</td>
<td>983</td>
<td>IE</td>
<td>FF</td>
</tr>
<tr>
<td>HotwireHotel</td>
<td><a href="https://goo.gl/pH9d6d">https://goo.gl/pH9d6d</a></td>
<td>1,457</td>
<td>10,618</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>IncredibleIndia</td>
<td><a href="http://incredibleindia.org">http://incredibleindia.org</a></td>
<td>251</td>
<td>2,172</td>
<td>IE</td>
<td>FF</td>
</tr>
<tr>
<td>Leris</td>
<td><a href="http://clear.uconn.edu/leris/">http://clear.uconn.edu/leris/</a></td>
<td>195</td>
<td>1,262</td>
<td>CH</td>
<td>FF</td>
</tr>
<tr>
<td>Minix3</td>
<td><a href="http://www.minix3.org">http://www.minix3.org</a></td>
<td>118</td>
<td>821</td>
<td>IE</td>
<td>CH</td>
</tr>
<tr>
<td>Newark</td>
<td><a href="http://www.ci.newark.ca.us">http://www.ci.newark.ca.us</a></td>
<td>598</td>
<td>17,426</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>Ofa</td>
<td><a href="http://www.ofa.org">http://www.ofa.org</a></td>
<td>578</td>
<td>5,381</td>
<td>IE</td>
<td>CH</td>
</tr>
<tr>
<td>PMA</td>
<td><a href="http://www.pilatesmethodalliance.org">http://www.pilatesmethodalliance.org</a></td>
<td>456</td>
<td>10,159</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>StephenHunt</td>
<td><a href="http://stephenhunt.net">http://stephenhunt.net</a></td>
<td>497</td>
<td>13,743</td>
<td>FF</td>
<td>IE</td>
</tr>
<tr>
<td>WIT</td>
<td><a href="http://www.wit.edu">http://www.wit.edu</a></td>
<td>300</td>
<td>3,249</td>
<td>FF</td>
<td>IE</td>
</tr>
</tbody>
</table>

4.3.3 Methodology

For the experiments, the latest stable versions of the browsers, Mozilla Firefox 46.0.1, Internet Explorer 11.0.33, and Google Chrome 51.0, were used. These browsers were selected for the evaluation as they represent the top three most widely used desktop browsers [15, 35]. The experiments were run on a 64-bit Windows 10 machine with 32GB memory and a 3rd Generation Intel Core i7-3770 processor. Since the set of XBIs reported by X-PERT can vary based on screen resolution, I also report the test monitor setup, which had a resolution of 1920 × 1080 and size of 23 inches. The subjects were rendered in the browsers with the browser viewport size set to the screen size.

Each subject was downloaded using the Scrapbook-X Firefox plugin and the wget utility, which download an HTML page along with all of the files (e.g., CSS, JavaScript, images, etc.) it needs to display. I then commented out portions of the JavaScript files and HTML code that made active connections with the server, such as Google Analytics, so that the subjects could be run locally in an offline mode. The downloaded subjects were then hosted on a local Apache web server.

X-PERT was run on each of the subjects to collect the set of initial XBIs present in the page. XFix was then run 30 times on each of the subjects to mitigate non-determinism in the search, and measured the run time in seconds. After each run of XFix on a subject, X-PERT was run on the repaired subject and recorded the remaining number of XBIs reported, if any.
I also conducted a human study with the aim of judging \xft with respect to the human-perceptible XBIs, and to gauge the change in the cross-browser consistency of the repaired page. The study involved 11 participants consisting of PhD and post-doctoral researchers whose field of study was Software Engineering. For the study, I first captured three screenshots of each subject page: (1) rendered in the reference browser, (2) rendered in the test browser before applying \xft's suggested repair, and (3) rendered in the test browser after applying the suggested fixes. I embedded these screenshots in HTML pages provided to the participants. I varied the order in which the before (pre-\xft) and after (post-\xft) versions were presented to participants, to minimize the influence of learning on the results and referred to them in the study as version\textsubscript{1} and version\textsubscript{2} based on the order of their presentation.

Each participant received a link to an online questionnaire and a set of printouts of the renderings of the page. I instructed the participants to individually (i.e., without consultation) answer four questions per subject: The first question asked the users to compare the reference and version\textsubscript{1} by opening them in different tabs of the same browser and circle the areas of observed visual differences on the corresponding printout. The second question asked the participants to rate the similarity of version\textsubscript{1} and reference on a scale of 0–10, where 0 represents no similarity and 10 means identical. Note that the similarity rating includes the participants reaction to intrinsic browser differences as well since they were not asked to exclude these. The third and fourth questions in the questionnaire were the same, but for version\textsubscript{2}.

For RQ1, I used X-PERT to determine the initial number of XBIs in a subject and the average number of XBIs remaining after each of the 30 runs of \xft. From these numbers I calculated the reduction of XBIs as a percentage.

For RQ2, I classified the similarity rating results from the human study into three categories for each subject: (1) improved: the after similarity rating was higher than that of the before version, (2) same: the after and before similarity ratings were exactly the same, and (3) decreased: the after similarity rating was lower than that of the before version. The human study data can be found at the project website [69].

For RQ3, I collected the average total running times of \xft and for Stages 3 and 4, the search phases, of the algorithm.

For RQ4, I compared the size, measured by the number of CSS properties, of browser specific code found in real-world websites to that of the automatically generated repairs. I used size for comparing similarity because CSS has a simple structure and does not contain any branching or looping constructs. I used wget to download the homepages of 480 websites in the Alexa Top 500 Global Sites [10] and analyzed their CSS to find the number of websites containing browser
specific code. Twenty sites could not be downloaded as they pointed to URLs without UIs — for instance the googleadservices.com and twimg.com web services. To find whether a website has browser specific CSS, I parsed its CSS files using the CSS Parser tool [18] and searched for browser specific CSS selectors, such as the one shown in Listing 4.1, based on well-known prefix declarations: -moz for Firefox, -ms for IE, and -webkit for Chrome. To calculate the size, I summed the numbers of CSS properties declared in each browser specific selector. To establish a comparable size metric for each subject web page used with $\mathcal{X}$Fix, I added the size of each subject’s previously existing browser specific code for $T$, the test browser, to the average size of the repair generated for $T$.

4.3.4 Threats to Validity

External Validity: The first potential threat is that I used a manual selection of the subjects. To minimize this threat, I only performed a manual filtering of the subjects to ensure that the subjects showed human perceptible XBIs and that X-PERT did not miss the observed XBIs (i.e., have a false negative). I also selected subjects from three different sources, including a random URL generator, to make the selection process generalizable across a wide variety of subjects. All the subjects had multiple XBIs reported by X-PERT (Table 4.2), and a mix of single (e.g., Bitcoin and IncredibleIndia) and multiple (e.g., HotwireHotel and Grantabooks) human-observable XBIs. A second potential threat is the use of only three browsers. To mitigate this threat, I selected the three most widely used browsers, as reported by different commercial agencies studying browser statistics [35, 15]. Furthermore, my approach is not dependent on the choice of browsers, so the results should generalize to other browsers.

Internal Validity: One potential threat is the use of X-PERT. However, there are no other publicly available tools for detecting XBIs that report the level of detail required by $\mathcal{X}$Fix to produce repairs. A further threat is represented by the changes I made to X-PERT favored my approach. However, the changes made were to provide access to existing information (and so do not change XBI-identifying behavior) or to address specific bugs. An example of one of the defects I found was a mismatch in the data type of a DomNode object being checked to see if it is contained in an array of String specifying the HTML tags to be ignored. I corrected this defect by adding a call to the getTagName() method of the DomNode object that returns the String HTML tag name of the node. I have made my patched version of X-PERT publicly available [69], with the download containing a README.txt file detailing the defects that were corrected.

The fact that my judgment was used to determine which browser rendering was the reference is not a threat to validity. This is because the metrics used were relative comparisons (e.g.,
consistency) and flipping the choice of reference rendering would have produced the same difference. Human participant understanding as to what constituted an XBI was not a threat to the correctness of the protocol either since I only asked them to spot differences between the renderings.

A potential threat is the number of real-world (Alexa) websites found to be using browser-specific styling. There exist numerous other ways to declare browser specific styling [6, 7] than the simple prefix selector declarations I used, and therefore the number of Alexa websites I found to be using browser-specific styling and the browser-specific code sizes calculated for each only represents a lower bound.

**Construct Validity:** A potential threat is that the similarity metric used in the human study is subjective. To mitigate this threat I used the relative similarity ratings given by the users, as opposed to the absolute value, to understand the participants’ relative notion of consistency quality. A second potential threat to validity is that screenshots of the subjects were used in the human study instead of actual HTML pages. I opted for this mechanism as not all of the users had the required environment (OS and browsers). Also, to mitigate this threat I designed the HTML pages containing the screenshots to scale based on the width of the user’s screen. Another potential threat is that the browser-specific code found in real-world (Alexa) websites might not necessarily be repair code for XBIs, so it might not be fair to compare that with XFix generated repair patches. However, to the best of my knowledge the primary purpose of browser-specific code is to target a particular browser and ensure cross-browser consistency.

### 4.3.5 Discussion of Results

#### 4.3.5.1 RQ1: Reduction of XBIs

Table 4.2 shows the results of RQ1. The results show that XFix reported an average 86% reduction in XBIs, with a median of 93%. This shows that XFix was effective in finding XBI fixes. Of the 15 subjects, XFix was able to resolve all of the reported XBIs for 33% of the subjects and was able to resolve more than 90% of the XBIs for 67% of the subjects.

I investigated the results to understand why XFix was not able to find suitable fixes for all of the XBIs. I found that the dominant reason for this was that there were pixel-level differences between the HTML elements in the test and reference browsers that were reported as XBIs. In many cases, perfect matching at the pixel level was not feasible due to the complex interaction among the HTML elements and CSS properties of a web page. Also, the different implementations
Table 4.2: Effectiveness of XFix in reducing XBIs

<table>
<thead>
<tr>
<th>Subject</th>
<th>#Before XBIs</th>
<th>Avg. #After XBIs</th>
<th>Reduction (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenjaminLees</td>
<td>25</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>37</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Eboss</td>
<td>49</td>
<td>29</td>
<td>41</td>
</tr>
<tr>
<td>EquilibriumFans</td>
<td>117</td>
<td>6</td>
<td>95</td>
</tr>
<tr>
<td>GrantaBooks</td>
<td>16</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HenryCountyOhio</td>
<td>11</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>HotwireHotel</td>
<td>40</td>
<td>4</td>
<td>90</td>
</tr>
<tr>
<td>IncredibleIndia</td>
<td>20</td>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>Leris</td>
<td>13</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Minix3</td>
<td>11</td>
<td>0.73</td>
<td>93</td>
</tr>
<tr>
<td>Newark</td>
<td>42</td>
<td>2</td>
<td>95</td>
</tr>
<tr>
<td>Ofa</td>
<td>16</td>
<td>3</td>
<td>83</td>
</tr>
<tr>
<td>PMA</td>
<td>39</td>
<td>10</td>
<td>75</td>
</tr>
<tr>
<td>StephenHunt</td>
<td>159</td>
<td>33</td>
<td>79</td>
</tr>
<tr>
<td>WIT</td>
<td>40</td>
<td>3</td>
<td>92</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td><strong>42</strong></td>
<td><strong>7</strong></td>
<td><strong>86</strong></td>
</tr>
<tr>
<td><strong>Median</strong></td>
<td><strong>37</strong></td>
<td><strong>3</strong></td>
<td><strong>93</strong></td>
</tr>
</tbody>
</table>

of the layout engines of the browser meant that a few pixel-level differences were unavoidable. After examining these cases, I hypothesized that these differences would not be human perceptible.

To investigate this hypothesis, I inspected the user-marked printouts of the before and after versions from the human study. I filtered out the areas of visual differences that represented inherent browser-level differences, such as font styling, font face, and native button appearance, leaving only the areas corresponding to XBIs.

I found that, for all but one subject, the majority of participants had correctly identified the areas containing layout XBIs in the before version of the page but had not marked the corresponding areas again in the after version. This indicated that the after version did not show the layout XBIs after they had been resolved by XFix. Overall, this analysis showed an average 99% reduction in the human observable XBIs (median 100%), confirming my hypothesis that almost all of the remaining XBIs reported by X-PERT were not actually human observable.

4.3.5.2 RQ2: Impact on Cross-browser Consistency

I calculated the impact of XFix on the cross-browser consistency of a subject based on the user ratings classifications, improved, same, or decreased. I found that 78% of the user ratings reported an improved similarity of the after version, implying that the consistency of the subject pages had improved with XFix’s suggested fixes. 14% of the user ratings reported the consistency quality as same, and only 8% of the user ratings reported a decreased consistency. Figure 4.6 shows the
distribution of the participant ratings for each of the subjects. As can be seen, all of the subjects, except two (Eboss and Leris), show a majority agreement among the participants in giving the verdict of improved cross-browser consistency. The improved ratings without considering Eboss and Leris rise to 85%, with the ratings for same and decrease dropping to 10% and 4%, respectively.

I investigated the two outliers, Eboss and Leris, to understand the reason for high discordance among the participants. I found that the reason for this disagreement was the significant number of inherent browser-level differences related to font styling and font face in the pages. Both of the subject pages are text intensive and contain specific fonts that were rendered very differently by the respective reference and test browsers. In fact, I found that the browser-level differences were so dominant in these two subjects that some of the participants did not even mark the areas of layout XBIs in the before version. Since the approach does not suggest fixes for resolving inherent browser-level differences, the judgment of consistency was likely heavily influenced by these differences, thereby causing high disagreement among the users. To further quantify the impact of the intrinsic browser differences on participant ratings, I controlled for intrinsic differences, as discussed in Section 4.3.5.1. This controlled analysis showed a mean of 99% reduction in XBIs, a value consistent with the results in Table 4.2.
Table 4.3: XFix’s average run time in seconds

<table>
<thead>
<tr>
<th>Subject</th>
<th>Search for Candidate Fixes</th>
<th>Search for Best Combination</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenjaminLees</td>
<td>159</td>
<td>14</td>
<td>204</td>
</tr>
<tr>
<td>Bitcoin</td>
<td>144</td>
<td>42</td>
<td>358</td>
</tr>
<tr>
<td>Eboss</td>
<td>1,729</td>
<td>780</td>
<td>2,685</td>
</tr>
<tr>
<td>EquilibriumFans</td>
<td>822</td>
<td>225</td>
<td>1,208</td>
</tr>
<tr>
<td>GrantaBooks</td>
<td>41</td>
<td>7</td>
<td>86</td>
</tr>
<tr>
<td>HenryCountyOhio</td>
<td>219</td>
<td>41</td>
<td>291</td>
</tr>
<tr>
<td>HotwireHotel</td>
<td>3,281</td>
<td>2,036</td>
<td>5,582</td>
</tr>
<tr>
<td>IncredibleIndia</td>
<td>599</td>
<td>247</td>
<td>908</td>
</tr>
<tr>
<td>Leris</td>
<td>105</td>
<td>46</td>
<td>169</td>
</tr>
<tr>
<td>Minix3</td>
<td>18</td>
<td>6</td>
<td>43</td>
</tr>
<tr>
<td>Newark</td>
<td>477</td>
<td>232</td>
<td>841</td>
</tr>
<tr>
<td>Ofa</td>
<td>122</td>
<td>113</td>
<td>257</td>
</tr>
<tr>
<td>PMA</td>
<td>3,050</td>
<td>1,384</td>
<td>4,488</td>
</tr>
<tr>
<td>StephenHunt</td>
<td>5,535</td>
<td>1,114</td>
<td>6,639</td>
</tr>
<tr>
<td>WIT</td>
<td>3,725</td>
<td>1,409</td>
<td>4,980</td>
</tr>
<tr>
<td>Mean</td>
<td>369</td>
<td>90</td>
<td>1916</td>
</tr>
<tr>
<td>Median</td>
<td>194</td>
<td>48</td>
<td>841</td>
</tr>
</tbody>
</table>

4.3.5.3 RQ3: Time Needed to Run XFix

Table 4.3 shows the average time results over the 30 runs for each subject. These results show that the total analysis time of XFix ranged from 43 seconds to 110 minutes, with a median of 14 minutes. The table also reports time spent in the two search routines. The “searchForCandidateFix” procedure was found to be the most time consuming, taking up 67% of the total runtime, with “searchForBestRepair” occupying 32%. (The remaining 1% was spent in other parts of the overall algorithm, for example the setup stage.) The time for the two search techniques was dependent on the size of the page and the number of XBIs reported by X-PERT. Although the runtime is lengthy for some subjects, it can be further improved via parallelization, as has been achieved in related work [85, 105].

4.3.5.4 RQ4: Similarity of Repair Patches to Real-world Websites’ Code

My analysis of the 480 Alexa websites revealed that browser specific code was present in almost 80% of the websites and therefore highly prevalent. This indicates that the patch structure of XFix’s repairs, which employs browser specific CSS code blocks, follows a widely adopted practice of writing browser specific code.

Figure 4.7 shows a box plot for browser specific code size observed in the Alexa websites and XFix subjects. The boxes represent the distribution of browser specific code size for the Alexa
websites for each browser (i.e., Firefox (FF), Internet Explorer (IE), and Chrome (CH)), while the circles show the data points for \(X\)\(\text{Fix}\) subjects. In each box, the horizontal line and the upper and lower edges show the median and the upper and lower quartiles for the distribution of browser specific code sizes, respectively. As the plot shows, the size of the browser specific code reported by Alexa websites and \(X\)\(\text{Fix}\) subjects are in a comparable range, with both reporting an average size of 9 CSS properties across all three browsers (Alexa: FF = 9, IE = 7, CH = 10 and \(X\)\(\text{Fix}\): FF = 9, IE = 13, CH = 6).

![Figure 4.7: Similarity of \(X\)\(\text{Fix}\) generated repair patches to real-world websites’ code](image)

### 4.4 Conclusion

To summarize, in this chapter I introduced an approach, \(X\)\(\text{Fix}\), that addresses layout XBIs in web applications. The prerequisites for repair, Detection and Localization, are instantiated using the existing XBT technique, X-PERT [137]. Designing an algorithm to resolve XBIs is my contribution. My repair approach implemented in \(X\)\(\text{Fix}\) uses two phases of guided search to find suitable fixes for the detected layout XBIs. The first phase of the search finds candidate fixes for each of the root causes identified for an XBI. The second phase then finds a subset of the candidate fixes that together minimizes the number of XBIs in the web page. In the evaluation performed on 15 real-world web pages, my repair approach was able to resolve, on average, 86%
of the X-PERT reported layout XBIs and 99% of the human observed XBIs. In a human study assessing the impact on the cross-browser consistency of the pages, 78% of the participant ratings reported improved consistency afterXFIX’s suggested fixes were applied. The repair patches generated by my approach were comparable in size to the browser-specific code present in real-world websites. Overall, these evaluation results are strong and support the hypothesis of my dissertation by showing that this approach using search-based techniques is highly effective in repairing layout XBIs in web pages.
Chapter 5

MFIX: Repair of Mobile Friendly Problems (MFPs)

Mobile devices have become one of the most common means of accessing the Internet. In fact, recent studies show that for a significant portion of web users, a mobile device is their primary means of accessing the Internet and interacting with other web-based services, such as online shopping, news, and communication [73, 35, 21, 29]. Unfortunately, many websites are not designed to gracefully handle users who are accessing their pages through a non-traditional sized device, such as a smartphone or tablet. These problematic sites may exhibit a range of usability issues, such as unreadable text, cluttered navigation, or content that overflows the device’s viewport and forces the user to pan and zoom the page in order to access content. Such usability issues are collectively referred as Mobile Friendly Problems (MFPs) [26, 13] and lead to a frustrating and poor user experience.

Despite the importance of MFPs, they are highly prevalent in modern websites — in a recent study over 75% of users reported problems in accessing websites from their mobile devices [73]. Over one third of users also said that they abandon mobile unfriendly websites and find other websites that work better on mobile devices. This underscores the importance for developers in ensuring the mobile friendliness of the web pages they design and maintain. Adding to this motivation is the fact that, as of April 2015, Google has incorporated mobile-friendliness as part of its ranking criteria when returning search results to mobile devices [28]. This means that unless a website is deemed to be mobile friendly, it is less likely to be highly ranked in the results returned to users.

Making websites mobile friendly is challenging even for a well motivated developer. These challenges arise from the difficulties in detecting and repairing MFPs. To detect these problems, developers must be able to verify a web page’s appearance on many different types and sizes of mobile devices. Since the scale of testing required for this is generally quite large, developers often use mobile testing services, such as BrowserStack [16] and SauceLabs [41], to determine if there
are problems in their sites. However, even with this information it is difficult for developers to improve or repair their pages. The reason for this is that the appearance of web pages is controlled by complex interactions between the HTML elements and CSS style properties that define a web page. This means that to fix a MFP, developers must typically adjust dozens of elements and properties while at the same time ensuring that these adjustments do not impact other parts of the page. For example, a seemingly simple solution, such as increasing the font size of text or the margins of clickable elements, can result in a distorted user interface that is unlikely to be acceptable to end users or developers.

Existing approaches are limited in helping developers to detect and repair MFPs. For example, the Mobile Friendly Test Tools produced by Google [26] and Bing [13], only focus on the detection of MFPs in a web page. While these tools may provide hints or suggestions as to how to repair the pages, the task of performing the repair is still a manual effort. Developers may also use frameworks, such as Bootstrap and Foundation, to help create pages that will be mobile friendly. However, the use of frameworks cannot guarantee the absence of mobile-friendly problems [3]. Some commercial websites attempt to automate this process (e.g., [14, 34, 20]), but are generally targeted for hobbyist pages as they require the transformed website to use one of their preset templates. This leaves developers with a lack of automated support for repairing MFPs.

To address this problem, I designed an approach to automatically generate CSS patches that can improve the mobile friendliness of a web page. To do this the approach builds graph-based models of the layout of a web page. It then uses constraints encoded by these graphs to find patches that can improve mobile friendliness while minimizing layout disruption. To efficiently identify the best patch, the approach leverages unique aspects of the problem domain to quantify metrics related to layout distortion and parallelize the computation of the solution. I implemented the approach in a prototype tool, \textit{MFix}, and evaluated its effectiveness on the home pages of 38 of the Alexa Top 50 most visited websites. The results showed that \textit{MFix} could effectively increase the mobile friendliness ratings of a page, typically by 33\%, while minimizing layout distortion. \textit{MFix} was also fast, needing less than 5 minutes, on average, to generate the CSS patch. I also evaluated the results with a user study, in which participants overwhelmingly preferred the repaired version of the website for use on mobile devices, and also considered the repaired page to be more readable than the original. Overall, these results are very positive and indicate that \textit{MFix} can help developers to improve the mobile friendliness of their web pages.


5.1 Background

In this section I discuss a variety of MFPs and current ways of addressing them in order to build a mobile friendly website.

5.1.1 Types of MFPs

Wide used mobile testing tools provided by Google [26] and Bing [13] report mobile friendly problems in five areas:

1. Font sizing: Font sizes optimized for viewing a web page on a desktop are often too small to be legible on a mobile device, forcing users to zoom in to read the text, and then out again to navigate around the page.

2. Tap target spacing: “Tap targets” are elements on a web page, such as a hyperlinks, buttons, or input boxes, that a user can tap or touch to perform actions, such as navigate to another page or fill and submit a form. If tap targets are located close to each other on a mobile screen, it can become difficult for a user to physically select the desired element without hitting a neighboring element accidentally. Targets may also be too small, requiring users to zoom into the page in order to tap them on their device.

3. Content sizing: When a web page extends beyond the width of a device’s viewport, the user is required to scroll horizontally or zoom out to access content. Horizontal scrolling is particularly considered problematic since users are typically used to scrolling vertically but not horizontally [74]. This can lead to important content being missed by users. Therefore attention to content sizing is particularly important on mobile devices, where a smaller screen means that space is limited, and the browser may not be resizable to fit the page.

4. Viewport configuration: Using the “meta viewport” HTML tag allows browsers to scale web pages based on the size of a user’s device. Web pages that do not specify or correctly use the tag may have content sizing issues, as the browser may simply scale or clip the content without adjusting for the layout of the page.

5. Flash usage: Flash content is not rendered by most mobile browsers. This makes content based on Flash, such as animations and navigation, inaccessible.

In the MFx approach, detailed in Section 5.2.2, I focus on addressing the first three of these problems. I regard the Flash usage as out of scope for the approach, since it requires a major content change in the page; while the viewport configuration problem is trivial to address, as it only requires insertion of a missing “meta viewport” tag into the page’s HTML head.
5.1.2 Current Methods for Addressing MFPs

I now discuss different approaches for addressing MFPs in websites to improve their mobile friendliness.

There are a number of ways in which a website can be adjusted to become more mobile friendly. In the early days of mobile web browsing, a common approach was to simply build an alternative mobile version of an existing desktop website. Such websites were typically hosted at a separate URL and delivered to a user when the web server detected the use of a mobile device. However, the cost and effort of building such a separate mobile website was high. To address this problem, commercial services, such as bMobilized [14] and Mobify [34], can automatically create a mobile website from a desktop version using a series of pre-designed templates. A drawback of these templated websites, however, is that they fail to capture the distinct design details of the original desktop version, making them look identical to every other organization using the service. Broadly speaking, although having a separate mobile website could address mobile friendly concerns, it introduces a heavy maintenance debt on the organization in ensuring that the mobile website renders and behaves consistently and as reliably as its regular desktop version, thereby doubling the cost of an organization’s online presence. Furthermore, having a separate mobile-only site would not help improve search-engine rankings of the organization’s main website, since the two versions reside at different URLs.

To avoid developing and maintaining separate mobile and desktop versions of a website, an organization may employ responsive design techniques. This kind of design makes use of CSS media queries to dynamically adjust the layout of a page to the screen size on which it will be displayed. The advantage of this technique over mobile dedicated websites is that the URL of the website remains the same. However, converting an existing website into a fully responsive website is an extremely labor intensive task, and is better suited for websites that are being built from scratch. As such, repairing an existing website may be a more cost effective solution than completely redeveloping the site. Furthermore, although a responsive design is likely to allow for a good mobile user experience, it does not necessarily preclude the possibility of MFPs, since additional styles may be used or certain provided styles may be incorrectly overridden [3].

MF1x introduces a novel technique for handling MFPs by adjusting specific CSS properties in the page and producing a repair patch. The repair patch uses CSS media queries to ensure that the modified CSS is only used for mobile viewing – that is, it does not affect the website when viewed on a desktop.
5.2 Specialization of the Generalized Approach, $\text{F}1\text{X}$

In this section, I provide details of my approach for repairing layout MFPs in web pages that is based on the generalized repair approach, $\text{F}1\text{X}$, explained in Chapter 3. For identifying MFPs in web pages, i.e., for the Detection phase of the debugging process, existing mobile testing tools, such as Google Mobile-Friendly Test Tool (GMFT) [26], can be used as an input to my approach. For completeness, I discuss the GMFT technique in Section 5.2.1. GMFT has limited support for Localization as the list of faulty HTML elements it supplies is generally incomplete. Therefore I developed a localization technique that is incorporated into the $\text{M}1\text{X}$ approach. I explain the localization technique in detail in Section 5.2.4. My contribution is in developing a localization and repair approach for finding suitable fixes for MFPs in web pages using search-based techniques to improve their mobile friendliness. Section 5.2.2 discusses this approach in more detail.

5.2.1 Detection of MFPs

For implementing the Detection phase of the debugging process, I use the mobile testing tool provided by Google, GMFT [26]. For completeness, I provide an overview of the GMFT; however, since the GMFT is a commercial product, its algorithmic details are not available. The GMFT reports mobile friendly problems in the five areas discussed in Section 5.1.1, namely, font sizing, tap target spacing, content sizing, viewport configuration, and flash usage. The GMFT web service takes as input a URL of a web page and returns a list of mobile friendly problems in the page that it finds and a screenshot of how the page appears on a mobile device. The GMFT also provides a list of suggested values for CSS properties for improving mobile friendliness of web pages [25].

5.2.2 Repair of MFPs

The goal of $\text{M}1\text{X}$ is to automatically generate a patch that can be applied to the CSS of a web page to improve its mobile friendliness. $\text{M}1\text{X}$ addresses the three specific problem types introduced in Section 5.1, namely font sizing, tap target spacing, and content sizing for the viewport – factors used by Google to rate the mobile friendliness of a page.

There is usually a straightforward fix for these problems – simply increase the font size used in the page and the margins of the elements within it. The result, however, is one that would likely be unacceptable to an end-user: such changes when taken to excess can significantly disrupt the layout of a page and require the user to perform excessive panning and scrolling. The challenge in
generating a successful repair, therefore, involves balancing two objectives – addressing a page’s mobile friendliness problems, while also ensuring an aesthetically pleasing and usable layout.

With this in mind, the goal of \( \mathcal{M} \text{Fix} \) is to generate a solution that is as faithful as possible to the page’s original layout. This requires fixing mobile friendliness problems while maintaining, where possible, the relative proportions and positioning of elements that are related to one another on the page (for example, links in the navigation bar, and the proportions of fonts for headings and body text in the main content pane).

My approach for generating a CSS patch can be roughly broken down into three distinct phases, segmentation, localization, and repair. These are shown in Figure 5.1. The diagram also shows the instantiations of the different abstraction points (AP1–4) of the \( \mathcal{M} \text{Fix} \) approach. \( \mathcal{M} \text{Fix} \) takes two inputs. The first input is the URL of a page under test (PUT). Typically, this would be a page that has been identified as failing a mobile friendly test (e.g., by using Google’s [26] or Bing’s [13] tool), but it may also be a page for which a developer would like to simply improve mobile friendliness. The second input is a detection function \( \mathcal{D} \), that identifies MFPs. I use GMFT as the function \( \mathcal{D} \). The segmentation phase identifies elements that form natural visual groupings on the page – referred to as segments. The localization phase then identifies the MFPs in the page, and relates these to the HTML elements and CSS properties in each segment. The last phase – repair – seeks to adjust the proportional sizing of elements within segments, along with the relative positions of each segment and the elements within them in order to generate a suitable patch. I now explain each of these three phases in more detail.

5.2.3 Phase 1: Segmentation

The first phase analyzes the structure of the page to identify segments – sets of HTML elements whose properties should be adjusted together to maintain the visual consistency of the repaired web page. An example of a segment is a series of text-based links in a menu bar where if the font size of any link in the segment is too small, then all of the links should be adjusted by the
same amount to maintain the links’ visual consistency. The reason MFix uses segments is that through manual experimentation with pages that contained MFPs, I found that once the optimal fix value for an element was identified, to maintain visual consistency, the same value would also need to be applied to closely related elements (i.e., those in the element’s segment). This insight motivated the use of segments, and it allowed the approach to treat many HTML elements as an equivalence class, which also reduced the complexity of the patch generation process.

To identify the segments in a page, the approach analyzes the Document Object Model (DOM) tree of the PUT. In informal experiments, I evaluated several well-known page segmentation analyses, such as VIPS [55], Block-o-matic [143], and correlation clustering [56]. I chose to use an automated clustering-based partitioning algorithm proposed by Romero et al. [136], as its segmentation results more readily conformed to my definition of a segment. I summarize the algorithm here for completeness. The approach starts by assigning each leaf element of the DOM tree to its own segment. Then, to cluster the elements, the approach iterates over the segments and uses a cost function to determine when it can merge adjacent segments. The cost function is based on the number of hops in the DOM tree between the lowest common ancestors of the two segments under consideration. If the number of hops is below a threshold based on the average depth of leaves in the DOM tree, then the approach will cluster the adjacent segments. The value
of this threshold is determined empirically. The approach continues to iterate over the segments until no further merges are possible (i.e., the segment set has reached a fixed point). The output is a set of segments, $\mathcal{Segs}$, where each segment contains a set of XPath IDs denoting the HTML elements that have been grouped together in the segment.

Figure 5.2a shows a simplified version of the segments that were identified for one of the web pages, Wiley, used in the evaluation. The red overlay rectangles show the visible elements that were grouped together as segments. These include the header content, a left-aligned navigation menu, the content pane, and the page’s footer.

5.2.4 Phase 2: Localization

The second phase identifies the parts of the PUT that must be targeted to address its MFPs. The second phase consists of two steps. In the first step, the approach analyzes the PUT to identify which segments contain MFPs. Then, based on the structure and problem types identified for each segment, the second step identifies the CSS properties that will most likely need to be adjusted to resolve each problem. The output of the localization phase is a mapping of the potentially problematic segments to these properties.

5.2.4.1 Identifying Problematic Segments

In the first step of the localization phase, the approach identifies MFP types in the PUT and the subset of segments that will likely need to be adjusted to address them.

In $\mathcal{MFix}$, MFPs in the PUT are detected by an Mobile Friendly Oracle (MFO). An MFO is a function that takes a web page as input and returns a list of MFP types it contains. The MFO can identify the presence of MFPs but cannot identify the faulty HTML elements and CSS properties responsible for the observed problems. In the implementation of $\mathcal{MFix}$, I use the GMFT [26] as $\mathcal{MFix}$’s MFO. However, any detector or testing tool may also be used as an MFO. The basic requirement for an MFO is that it can accurately report whether there are any types of MFPs present in the page. Ideally, the MFO should also detail what types of problems are present, along with a mapping of each problem to the corresponding HTML elements. However, these are not strict requirements: $\mathcal{MFix}$ can correctly function with the assumption that all segments have all problem types. Though this over-approximation can increase the amount of time needed to compute the best solution in the second phase.

Since I leverage the GMFT in the implementation, I discuss how the output of this particular tool is used by $\mathcal{MFix}$. I expect that other MFOs, such as Bing, could be adapted in a similar way. Given a PUT, the GMFT returns, for each problem type it detects, a reference to the HTML...
elements that contain that problem. However, through experimentation with the GMFT, I learned that the list of HTML elements it supplies is generally incomplete. Therefore, given a reported problem type, MFix applies a conservative filtering to the segments to identify which ones may be problematic with respect to that problem type. For example, if the GMFT reports that there is a problem with font sizing in the PUT, then MFix identifies any segment that contains a visible text element as potentially problematic. As mentioned above, this over-approximation may increase the time needed to compute the best solution, but does not introduce unsoundness into MFix.

The output of this step is a set of tuples of the form \( s, T \) where \( s \in Segs \) is a potentially problematic segment and \( T \) is the set of problem types associated with \( s \) (i.e., in the domain of \( \{tap._targets, font._size, content._size\} \)). Referring back to the example in Figure 5.2a, GMFT identified S3 as having two problem types, the tap targets were too close and the font size was too small, so the approach would generate a tuple for S3 where \( T \) includes these two problem types.

### 5.2.4.2 Identifying Problematic CSS Properties

After identifying the subset of problematic segments, the approach needs to identify the CSS properties that may need to be adjusted in each segment to make the page mobile friendly. The general intuition of this step is that each of a segment’s identified problem types generally map to a set of CSS properties within the segment. However, this step is complicated by the fact that HTML elements may not explicitly define a CSS property (i.e., they may inherit a style from a parent element) and that MFix adjusts CSS properties at the segment level instead of the individual element level.

To address these issues, I introduce the concept of a Property Dependence Graph (PDG), which for a given segment and problem type, models the relevant style relationships among its HTML elements based on CSS inheritance and style dependencies. Formally, I define a PDG as a directed graph of the form \( \langle E, R, M \rangle \). Here \( e \in E \) is a node in the graph that corresponds to an HTML element in the PUT that has an explicitly defined CSS property, \( p \in P \), where \( P \) is the set of CSS properties relevant for a problem type (e.g., \texttt{font-size} for font sizing problems, \texttt{margin} for tap target issues, etc.). \( R \subseteq E \times E \) is a set of directed edges, such that for each pair of elements \( (e_1, e_2) \in R \), there exists a dependency relationship between \( e_1 \) and \( e_2 \). \( M \) is a function \( M : R \rightarrow 2^C \) that maps each edge to a set of tuples of the form \( C : \langle p, \varphi \rangle \), where \( p \in P \) and \( \varphi \) is a ratio between the values of \( p \) for \( e_1 \) and \( e_2 \). This function is used in the following repair phase (Section 5.2.5) to ensure that style changes made to a segment remain consistent across pairs of elements in a dependency relationship.
\textsc{M}Fix defines a variant of PDG for each of the three problem types: the Font PDG (FPDG), the Content Size PDG (CPDG), and the Tap Target PDG (TPDG). Each of these three graphs has a specific set of relevant CSS properties \((P)\), a dependency relationship, and a mapping function \((M)\). Note that I only present the formal definition of the FPDG, as the other two graphs are defined in a similar manner.

The FPDG is constructed for any segment for which a font sizing problem type has been identified. For this problem type, the most relevant CSS property is clearly \texttt{font-size}, but the \texttt{line-height}, \texttt{width}, and \texttt{height} properties of certain elements may also need to be adjusted if font sizes are changed. Therefore \(P = \{\texttt{font-size}, \texttt{line-height}, \texttt{width}, \texttt{height}\}\). A dependency relationship exists between any \(e_1, e_2 \in E\), if and only if \(e_1\) is an ancestor of \(e_2\) in the DOM tree and \(e_2\) has an explicitly defined CSS property, \(p \in P\), i.e., the value of the property is not inherited from \(e_1\). The general intuition of using this dependency relationship is that only nodes that explicitly define a relevant property may need to be adjusted and the remainder of the nodes in between \(e_1, e_2\) will simply inherit the style from \(e_1\). The ratio, \(\varphi\), associated with each edge is the value of \(p\) defined for \(e_1\) divided by the value of \(p\) defined for \(e_2\). To illustrate consider two HTML elements in S3 of Figure 5.2a. The first, \(e_1\), is a \texttt{⟨div⟩} tag wrapping all of the elements in S3 with \texttt{font-size = 13px} and the second, \(e_2\), is the \texttt{⟨h2⟩} element containing the text “Resources” with \texttt{font-size = 18px}. A dependency relationship exists from \(e_1\) to \(e_2\) with \(p\) as \texttt{font-size} and the ratio \(\varphi = 0.72\).

The output of this final step is the set, \(I\), of tuples where each tuple is of the form \(<s, g, a>\) where \(s\) identifies the segment to which the tuple corresponds, \(g\) identifies a corresponding PDG, and \(a\) is an adjustment factor for the PDG that is initially set to 1. The adjustment factor is used in the repair phase and serves as a multiplier to the ratios defined for the edges of each PDG. A tuple is added to \(I\) for each problem type that was identified as applicable to a segment. Referring back to the example in Figure 5.2a, the approach would generate two tuples for S3, one containing an FPDG and the other containing an TPDG.

### 5.2.5 Phase 3: Repair

The goal of the third phase is to compute a repair for the PUT. The best repair has to balance two objectives. The first objective is to identify the set of changes – a \textit{patch} – that will most improve the PUT’s mobile friendliness. The second objective is to identify the set of changes that does not significantly change the layout of the PUT.
5.2.5.1 Metrics

A key insight for $\mathcal{MF}i\mathcal{x}$ is that both of the aforementioned objectives – mobile friendliness and layout distortion – can be quantified. For the first objective, it is typical for mobile friendly test tools to assign a numeric score to a page, where this score represents the page’s mobile friendliness. For example, the Google PageSpeed Insights Tool (PSIT) assigns pages a score in the range of 0 to 100, with 100 being a perfectly mobile friendly page. By treating this score as a function, $F$, that operates on a page, it is possible to establish an ordering on solutions and use that ordering to identify a best solution among a group of solutions. The second objective can also be quantified as a function, $L$, that compares the amount of change between the layout of a page containing a candidate patch versus the layout of the original page. The amount of change in a layout can be determined by building models that express the relative visual positioning among and within the segments of a page. I refer to these models as the Segment Model (SM) and Intra-Segment Model (ISM), respectively. Given these two models, $\mathcal{MF}i\mathcal{x}$ uses graph comparison techniques to quantify the difference between the models for the original page and a page with an applied candidate solution.

I now provide a more formal definition of the SM and ISM. A Segment Model (SM) is defined as a directed complete graph where the nodes are the segments identified in the first phase (Section 5.2.3) and the edge labels represent layout relationships between segments. To determine the edge labels, the approach first computes the Minimum Bounding Rectangles (MBRs) of each segment. This done by finding the maximum and minimum X and Y coordinates of all of the elements included in the segment, which can be found by querying the DOM of the page. Based on the coordinates of each pair of MBRs, the approach determines which of the following relationships apply: (1) intersection, (2) containment, or (3) directional (i.e., above, below, left, right). Each edge in an SM is labeled in this manner. Referring to Figure 5.2a, one of the relationships identified would be that S1 is above S3 and S4. An ISM is the same, but is built for each segment and the nodes are the HTML elements within the segment.

To quantify the layout differences between the original page and a transformed page to which a candidate patch has been applied, the approach computes two metrics. The first metric is at the segment level. The approach sums the size of the symmetric difference between each edge’s labels in the SM of the original page and the SM of the transformed page. Recall that both models are complete graphs, so a counterpart for each edge exists in the other model. To illustrate, consider the examples shown in Figures 5.2a and 5.2b. The change to the page has caused segments S3 and S4 to overlap. This change in the relationship between the two segments would be counted as a difference between the two SMs and increase the amount of layout difference. The second metric
is similar to the first but compares the ISM for each segment in the original and transformed page. The one difference in the computation of the metric is that the symmetric difference is only computed for the intersection relationship. The intuition behind this difference in counting is that I consider movement of elements within a segment, except for intersection, to be an acceptable change to accommodate the goal of increasing mobile friendliness. Referring back to the example shown in Figure 5.2b, nine intra-segment intersections are counted among the elements in segment S4 as shown by dashed red ovals. The difference sums calculated at the segment and intra-segment level are returned as the amount of layout difference.

5.2.5.2 Computing Candidate Mobile Friendly Patches

To identify the best CSS patch, the approach must find new values for the potentially problematic properties, identified in the first phase, that make the PUT mobile friendly while also maintaining its layout. To state this more formally, given \( I \), the approach must identify a set of new values for each of the adjustment factors (i.e., \( a \)) in each tuple of \( I \) so that the value of \( F \) is 100 (i.e., the maximum mobile friendliness score) and the value of \( L \) is zero (i.e., there are no layout differences).

A direct computation of this solution faces two challenges. The first of these challenges is that an optimal solution that satisfies both of the above conditions may not exist. This can happen due to constraints in the layout of the PUT. The second challenge is that, even if such a solution were to exist, it exists in a solution space that grows exponentially based on the number of elements and properties that must be considered. Since many of the CSS properties have a large range of potential values, a direct computation of the solution would be too expensive to be practical. Both of these challenges motivate the use of an approximation algorithm to identify a repair. Therefore, the approach must find a set of values that minimizes the layout score while maximizing the mobile friendliness score.

The design of my approximation algorithm in \( M\text{F}1\text{X} \) takes into account several unique aspects of the problem domain to generate a high quality patch in a reasonable amount of time. The first of these aspects is that, through manual experimentation, I learned that good or optimal solutions typically involve a large number of small changes to many segments. This motivates targeting a solution space comprised of candidate solutions that differ from the original page in many places but by only small amounts. The second of these aspects is that computing the values of the \( L \) and \( F \) functions is expensive. The reason for this is that \( F \) requires accessing an API on the web and \( L \) requires rendering the page and computing layout information for the two versions of the PUT. This motivates us to avoid algorithms that require sequential processing of \( L \) and \( F \) (e.g., simulated annealing or genetic algorithms).
To incorporate these insights, the approximation algorithm first generates a set of size $n$ of candidate patches. To generate each candidate patch, the approach creates a copy of $I$, called $I'$, then iterates over each tuple in $I'$ and with probability $x$, randomly perturbs the value of the adjustment factor (i.e., $a$) using a process I describe in more detail in the next paragraph. Then $I'$ is converted into a patch, $R$, using the process described in the next section (Section 5.2.5.3), and added to the set of candidate patches. This process is repeated until the approach has generated $n$ candidate patches. The approach then computes, in parallel, the values of $F$ and $L$ for a version of the PUT with an applied candidate patch. (The implementation of $\mathcal{MF}ix$ uses Amazon Web Services (AWS) to parallelize this computation.) The objective score for the candidate patch is then computed as a weighted sum of $F$ and $L$. The candidate patch with the maximum score, i.e., with the highest value of $F$ and the lowest value of $L$, is selected as the final solution, $R_{max}$. Figure 5.2c shows $R_{max}$ applied to the example page.

$\mathcal{MF}ix$ perturbs adjustment factors in such a way as to take advantage of my insight that the optimal solutions differ from the original page in many places but by only small amounts. To represent this insight, I based the perturbation on a Gaussian distribution around the original value in a property. Through experimentation, I found that it was most effective to have the mean ($\mu$) and standard deviation ($\sigma$) values used for the Gaussian distribution vary based on the specific MFP type being addressed. For each problem type, the goal was to identify a $\mu$ and $\sigma$ that provided a large enough range to allow sufficient diversity in the generation of candidate patches. For identifying $\mu$ values, I found through experimentation that $\mu$ set at the values suggested by the GMFT [25] was not effective in generating candidate patches that could improve the mobile friendliness of the PUT. Therefore I added an amendment factor to the values suggested by the GMFT to allow the approach to select a value considered mobile friendly with a high probability. The specific amendment factors I found the most effective were: +14 for font size, -20 for content sizing, and 0 for tap target sizing problems. For example, if the GMFT suggested value for font size problems was 16px, I set $\mu$ at 30px. For each problem type, I then identified a $\sigma$ value. The specific values I determined to be most effective were: $\sigma = 16$ for content size problems, $\sigma = 5$ for font size problems, and $\sigma = 2$ for tap target spacing problems.

5.2.5.3 Generating the Mobile Friendly Patch

Given a set $I$, the approach generates a repair patch, $R$, and modifies the PUT so that $R$ will be applied at runtime. The general form of $R$ is a set of CSS style declarations that apply to the HTML elements of each segment in $I$. To generate $R$, the approach iterates over all tuples in $I$. For each tuple, the approach iterates over each node of its PDG, starting with the root node, and
computes a new value that will be assigned to the CSS property represented by the node. The new value for a node is computed by multiplying the new value assigned to its predecessor by the ratio, \(\phi\), defined on the edge with the predecessor. Once new property values have been computed for all nodes in the PDG, the approach generates a set of fixes, where each fix is represented as a tuple \((i, p, v)\), where \(i\) is the XPath for each node in the PDG that had a property change, \(p\) is the changed CSS property, and \(v\) is the newly computed value. These tuples are made into CSS style declarations by converting \(i\) into a CSS selector and then adding the declarations of \(p\) and \(v\) within the selector. All of the generated CSS style declarations are then wrapped in a CSS media query that will cause it to be loaded when accessed by a mobile device. In practice I found that the size range specified in the \(M\)Fix generated patch’s media query is applicable to a wide range of mobile devices. However, to allow developers to generate patches for specific device sizes, I provide configurable size parameters in the media query.

Referring back to the example, the ratio \((\phi)\) between \(e_1\) (\(<\text{div}\rangle\) containing all elements in S3) and \(e_2\) (\(<\text{h2}\rangle\) containing text “Resources”) is 0.72. Consider a tuple \(\langle\text{S3, font-size, 2}\rangle\) in \(I\). Thus, a value \(v\) of 26px is calculated for the predecessor node \(e_1\) based on the adjustment factor 2. Accordingly \(v = 26\text{px} \times 1/0.72 = 36\text{px}\) is calculated for \(e_2\). Thus, the approach generates two fix tuples: \(\langle\text{div, font-size, 26px}\rangle\) and \(\langle\text{h2, font-size, 36px}\rangle\).

### 5.3 Evaluation

To evaluate my approach for repairing MFPs, I designed experiments to determine its effectiveness, running time, and the visual appeal of its solutions. The specific research questions I considered were:

**RQ1:** How effective is \(M\)Fix in repairing mobile friendly problems in web pages?

**RQ2:** How long does it take for \(M\)Fix to generate patches for the mobile friendly problems in web pages?

**RQ3:** How does \(M\)Fix impact the visual appeal of web pages after applying the suggested CSS repair patches?

#### 5.3.1 Implementation

I implemented the approach in Java as a prototype tool named \(M\)Fix [94]. For identifying the mobile friendly problems in a web page, I used the GMFT [26] and PSIT [27] APIs. I also used the PSIT for obtaining the mobile friendliness score (labeled as “usability” in the PSIT report).
For identifying segments in a web page and building the SM and ISM, MFix first builds the DOM tree by rendering the page in an emulated mobile Chrome browser v60.0 and extracting rendering information, such as element MBRs and XPath, using Javascript and Selenium WebDriver. The segmentation threshold value determined by the average depth of leaves in a DOM tree was capped at four to avoid the situation where all of the visible elements in a page were wrapped in one large segment. This constant value was determined empirically, and was implemented as a configurable parameter in MFix. I used jStyleParser for identifying explicitly defined CSS properties for HTML elements in a page for building the PDG. I parallelized the evaluation of candidate solutions using a cloud of 100 Amazon EC2 t2.xlarge instances pre-installed with Ubuntu 16.04.

5.3.2 Subjects

For the experiments I used 38 real-world subjects collected from the top 50 most visited websites across all seventeen categories tracked by Alexa [9]. The subjects are listed in Table 5.1. The columns “Category” and “Rank” refer to the source Alexa category and rank of the subject within that category, respectively. The column “#HTML” refers to the total number of HTML elements in a subject, which I counted by parsing the subject’s DOM for node type “element”. This value gives an approximation for the size and complexity of the subject.

I used Alexa as the source of the subjects as the websites represent popular widely used sites and a mix of different layouts. From the 651 unique URLs that were identified across the 17 categories, I excluded the websites that passed the GMFT or had adult content. Each of the remaining 38 subjects was downloaded using the Scrapbook-X Firefox plugin, which downloads an HTML page and its supporting files, such as images, CSS, and Javascript. I then removed the portions of the subject pages that made active internet connections, such as for advertisements, to enable running of the subjects in an offline mode.

5.3.3 Experiment One

To address RQ1 and RQ2, I ran MFix ten times on each of the 38 subjects to mitigate the non-determinism inherent in the approximation algorithm used to find a repair solution.

For RQ1, I considered two metrics to gauge the effectiveness of MFix. For the first metric, I used the GMFT to measure how many of the subjects were considered mobile friendly after the patch was applied. For the second metric, I compared the before and after scores for mobile friendliness and layout distortion for each subject. For comparing mobile friendliness score, I
Table 5.1: Subjects used in the evaluation of MFix

<table>
<thead>
<tr>
<th>ID</th>
<th>URL</th>
<th>Category</th>
<th>Rank</th>
<th>#HTML</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://aamc.org">http://aamc.org</a></td>
<td>Health</td>
<td>23</td>
<td>598</td>
</tr>
<tr>
<td>2</td>
<td><a href="https://arxiv.org">https://arxiv.org</a></td>
<td>Science</td>
<td>21</td>
<td>381</td>
</tr>
<tr>
<td>3</td>
<td><a href="http://us.battle.net">http://us.battle.net</a></td>
<td>Kids and teens</td>
<td>2</td>
<td>615</td>
</tr>
<tr>
<td>4</td>
<td><a href="https://bitcointalk.org">https://bitcointalk.org</a></td>
<td>Science</td>
<td>25</td>
<td>1302</td>
</tr>
<tr>
<td>5</td>
<td><a href="http://blizzard.com">http://blizzard.com</a></td>
<td>Kids and teens</td>
<td>33</td>
<td>313</td>
</tr>
<tr>
<td>6</td>
<td><a href="https://boardgamegeek.com">https://boardgamegeek.com</a></td>
<td>Games</td>
<td>31</td>
<td>4474</td>
</tr>
<tr>
<td>7</td>
<td><a href="https://bulbagarden.net">https://bulbagarden.net</a></td>
<td>Kids and teens</td>
<td>26</td>
<td>151</td>
</tr>
<tr>
<td>8</td>
<td><a href="http://coinmarketcap.com">http://coinmarketcap.com</a></td>
<td>Science</td>
<td>8</td>
<td>1964</td>
</tr>
<tr>
<td>9</td>
<td><a href="http://correios.com.br/para-voce">http://correios.com.br/para-voce</a></td>
<td>Society</td>
<td>14</td>
<td>769</td>
</tr>
<tr>
<td>10</td>
<td><a href="http://dict.cc">http://dict.cc</a></td>
<td>Reference</td>
<td>20</td>
<td>633</td>
</tr>
<tr>
<td>11</td>
<td><a href="https://www.discogs.com">https://www.discogs.com</a></td>
<td>Arts</td>
<td>26</td>
<td>5738</td>
</tr>
<tr>
<td>12</td>
<td><a href="http://drudgereport.com">http://drudgereport.com</a></td>
<td>News</td>
<td>23</td>
<td>719</td>
</tr>
<tr>
<td>13</td>
<td><a href="http://www.finalfantasyxiv.com">http://www.finalfantasyxiv.com</a></td>
<td>Games</td>
<td>37</td>
<td>61</td>
</tr>
<tr>
<td>14</td>
<td><a href="http://www.flashscore.com">http://www.flashscore.com</a></td>
<td>Sports</td>
<td>16</td>
<td>6621</td>
</tr>
<tr>
<td>15</td>
<td><a href="https://www.fragrantica.com">https://www.fragrantica.com</a></td>
<td>Health</td>
<td>35</td>
<td>1091</td>
</tr>
<tr>
<td>16</td>
<td><a href="http://forum.gsmhosting.com/vbb">http://forum.gsmhosting.com/vbb</a></td>
<td>Home</td>
<td>39</td>
<td>2618</td>
</tr>
<tr>
<td>17</td>
<td><a href="http://www.intellicast.com">http://www.intellicast.com</a></td>
<td>Science</td>
<td>38</td>
<td>1393</td>
</tr>
<tr>
<td>18</td>
<td><a href="https://www.irctc.co.in">https://www.irctc.co.in</a></td>
<td>Regional</td>
<td>34</td>
<td>1031</td>
</tr>
<tr>
<td>20</td>
<td><a href="https://www.leo.org">https://www.leo.org</a></td>
<td>Reference</td>
<td>31</td>
<td>990</td>
</tr>
<tr>
<td>21</td>
<td><a href="http://letour.fr">http://letour.fr</a></td>
<td>Sports</td>
<td>3</td>
<td>1260</td>
</tr>
<tr>
<td>22</td>
<td><a href="http://lolcounter.com">http://lolcounter.com</a></td>
<td>Kids and teens</td>
<td>30</td>
<td>1257</td>
</tr>
<tr>
<td>23</td>
<td><a href="http://www.mmo-champion.com">http://www.mmo-champion.com</a></td>
<td>Games</td>
<td>29</td>
<td>1903</td>
</tr>
<tr>
<td>24</td>
<td><a href="http://myway.com">http://myway.com</a></td>
<td>Computers</td>
<td>42</td>
<td>135</td>
</tr>
<tr>
<td>26</td>
<td><a href="http://www.nasa.gov">http://www.nasa.gov</a></td>
<td>Science</td>
<td>28</td>
<td>2108</td>
</tr>
<tr>
<td>27</td>
<td><a href="http://nvidia.com">http://nvidia.com</a></td>
<td>Games</td>
<td>20</td>
<td>719</td>
</tr>
<tr>
<td>28</td>
<td><a href="http://rotoworld.com">http://rotoworld.com</a></td>
<td>Sports</td>
<td>41</td>
<td>2523</td>
</tr>
<tr>
<td>29</td>
<td><a href="http://sigmaaldrich.com">http://sigmaaldrich.com</a></td>
<td>Science</td>
<td>37</td>
<td>141</td>
</tr>
<tr>
<td>30</td>
<td><a href="http://us.soccerway.com">http://us.soccerway.com</a></td>
<td>Sports</td>
<td>30</td>
<td>2708</td>
</tr>
<tr>
<td>31</td>
<td><a href="http://www.square-enix.com">http://www.square-enix.com</a></td>
<td>Games</td>
<td>30</td>
<td>198</td>
</tr>
<tr>
<td>32</td>
<td><a href="https://travel.state.gov">https://travel.state.gov</a></td>
<td>Home</td>
<td>26</td>
<td>440</td>
</tr>
<tr>
<td>33</td>
<td><a href="http://www.weather.gov">http://www.weather.gov</a></td>
<td>Science</td>
<td>18</td>
<td>1101</td>
</tr>
<tr>
<td>34</td>
<td><a href="http://www.bom.gov.au">http://www.bom.gov.au</a></td>
<td>Kids and teens</td>
<td>48</td>
<td>685</td>
</tr>
<tr>
<td>35</td>
<td><a href="http://www.wiley.com">http://www.wiley.com</a></td>
<td>Shopping</td>
<td>14</td>
<td>460</td>
</tr>
<tr>
<td>36</td>
<td><a href="http://onlinelibrary.wiley.com">http://onlinelibrary.wiley.com</a></td>
<td>Business</td>
<td>33</td>
<td>824</td>
</tr>
<tr>
<td>37</td>
<td><a href="https://www.wowprogress.com">https://www.wowprogress.com</a></td>
<td>Games</td>
<td>46</td>
<td>2828</td>
</tr>
<tr>
<td>38</td>
<td><a href="https://xkcd.com">https://xkcd.com</a></td>
<td>Arts</td>
<td>48</td>
<td>121</td>
</tr>
</tbody>
</table>
selected, for each subject over the ten runs, the repair that represented a median score. For layout distortion, I selected, for each subject over the ten runs, the best and worst repair, in terms of layout distortion, that passed the mobile friendly test. Essentially, for each subject, these were the two patched pages that passed the mobile friendly test and had the lowest (best) and highest (worst) amount of distortion. For the subjects that did not pass the mobile friendly test, I considered the patched pages with the highest mobile friendly scores to be the “passing” pages.

For RQ2, I measured the average total running time of $M$Fix for each of the ten runs for each of the subjects, and also measured the time spent in the different stages of the approach.

5.3.3.1 Discussion of results

The results for effectiveness (RQ1) were that 95% (36 out of 38) of the subjects passed the GMFT after applying $M$Fix’s suggested CSS repair patch. This shows that the patches generated by $M$Fix were effective in making the pages pass the mobile friendly test.

Figure 5.3 shows the results of comparing the before and after median mobile friendliness scores for each subject. For each subject, the dark gray portion shows the score reported by the PSIT for the patched page and the light gray portion shows the score for the original version. The black horizontal line drawn at 80 indicates the value above which the GMFT considers a page to have passed the test and to be mobile friendly. On average, $M$Fix improved the mobile friendliness score of a subject by 33%. Overall, these results show that $M$Fix was able to consistently improve a subject’s mobile friendliness score.
I also compared the layout distortion score for the best and worst repairs of each subject. On average, the best repair had a layout distortion score 55% lower than the worst repair. These results show that \( \mathcal{MF} \text{ix} \) was effective in identifying patches that could reduce the amount of distortion in a solution that was able to pass the mobile friendly test. (For RQ3, I examined, via a user study, if this reduction in distortion translates into a more attractive page.)

I investigated the results to understand why two subjects did not pass the GMFT. The patched version of the first subject, gsmhosting, contained a content sizing problem. The original version of the page did not contain this problem, which indicates that the increased font size introduced by the patch caused content in this page to overflow the viewport width. For the second subject, aamc, \( \mathcal{MF} \text{ix} \) was not able to fully resolve its content sizing problem as the required value was extremely large compared to the range explored by the Gaussian perturbation of the adjustment factor. Both of these issues suggest further refinements to \( \mathcal{MF} \text{ix} \) that could be explored in future work, such as making the process iterative and expanding the initial search space.

The total running time (RQ2) required by \( \mathcal{MF} \text{ix} \) for the different subjects ranged from 2 minutes to 10 minutes, averaging a little less than 5 minutes. As of August 2017, an Amazon EC2 t2.xlarge instance was priced at $0.188 per hour. Thus, with an average time of 5 minutes the cost of running \( \mathcal{MF} \text{ix} \) on 100 instances was $1.50 per subject. Figure 5.4 shows a breakdown of the average time for the different stages of the approach. As can be seen from the chart, finding the repair for the mobile friendly problems (phase 3) was the most time consuming, taking up almost 60% of the total time. A major portion of this time was spent in evaluating the candidate solutions by invoking the PSIT API. The remainder of the time was spent in calculating layout distortion, which is dependent on the size of the page. The overhead caused by network delay in communicating with the Amazon cloud instances was negligible. For the API invocation, I
implemented a random wait time of 30 to 60 seconds between consecutive calls to avoid retrieving stale or cached results. Identifying problematic segments was the next most time consuming step as it required invoking the GMFT API.

5.3.4 Experiment Two

To address RQ3, I conducted a user-based survey to evaluate the aesthetics and visual appeal of the repaired page. The main intent of the study was to evaluate the effectiveness of the layout distortion metric, $L$ (Section 5.2.5), in minimizing layout disruptions and producing attractive pages. The general format of the survey was to ask participants to compare the original and repaired versions of a subset of the subjects. To make the survey length manageable, I divided the 38 subjects into six different surveys, each with six or seven subjects. For each subject, the survey presented, in random order, a screenshot of the original and repaired pages when displayed in a frame of the mobile device. The screenshots were obtained from the output of the GMFT. An example of one such screenshot is shown in Figure 5.2c. I asked each human subject to (1) select which of the two versions (original or repaired) they would prefer to use on their mobile device; (2) rate the readability of each version of the page on a scale of 1–10, where 1 means low and 10 means high; and (3) rate the attractiveness of the page on a scale of 1–10, where 1 means low and 10 means high. I had two variants of the survey, one that used the best repair as the screenshot of the repaired page and the other one that used the worst repair as the screenshot of the repaired page. Here, the best and worst repairs were as defined in Experiment 1.

I used Amazon Mechanical Turk (AMT) service to conduct the surveys. AMT allows users (requesters) to anonymously post jobs which it then matches them to anonymous users (workers) who are willing to complete those tasks to earn money. To avoid workers who had a track record of haphazardly completing tasks, I only allowed workers that had high approval ratings for their previously completed tasks (over 95%) and had completed more than 5,000 approved tasks to complete the survey. In general, this is considered a fairly selective criteria for participant selection on AMT. For each survey, I had 20 anonymous participants, giving us a total of 240 completed surveys across both variants of the survey. Each participants was paid $0.65 for completing a survey.

5.3.4.1 Discussion of results

Based on the analysis of the results of the first variant of the survey, I found that the users preferred to use the repaired version in 26 out of 38 subjects, three subjects received equal preference for the original and repaired versions, and only nine subjects received a preference for using the original
version. Interestingly, users preferred to use the repaired version even for the two subjects that did not pass the GMFT. For readability, all but four subjects were rated as having an improved readability over the original versions. On average, the readability rating of the repaired pages showed a 17% improvement over original versions (original = 5.97, repaired = 6.98). This result was also confirmed as statistically significant using the Wilcoxon signed-rank test with a p-value = 1.53 × 10^{-14} < 0.05. Using the effect size metric based on Vargha-Delaney A measure, readability of the repaired version was observed to be 62% of the time better than the original version.

With regards to attractiveness, no statistical significance was observed, implying that MFix did not deteriorate the aesthetics of the pages in the process of automatically repairing the reported mobile friendly problems. In fact, overall, the repaired versions produced by MFix were rated slightly higher than original versions for attractiveness (avg. original = 6.50, avg. repaired = 6.67 and median original = 6.02, median repaired = 7.12).

I investigated the nine subjects where the repaired version was not preferred by the participants. Based on my analysis, I found two dominant reasons that applied to all of the nine subjects. First, these subjects all had a fixed sized layout, meaning that the section and container elements in the pages were assigned absolute size and location values. This caused a cascading effect with any change introduced in the page, such as increasing font sizes or decreasing width to fit the viewport. The second reason was linked to the first as the pages were text intensive, thereby requiring MFix to increase font sizes. These results motivate future work in techniques that can better handle these types of pages.

Overall, these results indicate that MFix was very effective in generating repaired pages that (1) users preferred over the original version, (2) considered to be more readable, and (3) that did not suffer in terms of visual aesthetics.

The results for the second variant of the survey underscored the importance of the layout distortion objective and the impact visual distortions can have on end users’ perception of a page’s attractiveness. The results showed that the users preferred to use the original, non-mobile friendly version, in 22 out of 38 subjects and preferred to use the repaired version for only 16 subjects. Readability showed similar results as the first survey variant. On average, an improvement of 11% in readability was observed for the repaired pages compared to the original versions, and was still found to demonstrate statistical significance (p-value = 7.54 × 10^{-6} < 0.05). This is expected as the enlarged font sizes can make the text very readable in the repaired versions despite layout distortions. However, in this survey a statistical significance (p-value = 2.20 × 10^{-16} < 0.05) was observed for the attractiveness of the original version being rated higher than the repaired version. On average, the original version was rated 6.82 (median 7.00) and the repaired version
was rated 5.64 (median 5.63). In terms of the effect size metric, the repaired version was rated to have a better attractiveness only 38% of the time. These results strongly indicate that the layout distortion objective plays an important role in generating patches that make the pages more attractive to end users.

5.3.5 Threats to Validity

**External Validity**: The first potential threat is bias in the selection of participants for the user-based study in Experiment two. To address this threat, I used AMT that provided us with a large pool of anonymous participants. I only specified qualification requirements for the participants in the user study (i.e., high numbers of previously completed tasks and high approval ratings, as outlined in Section 5.3.4) to ensure authentic results. Another potential threat is in the selection of subject web pages for the evaluation of $M$Fix. To mitigate any bias, I used the home pages of websites drawn from Alexa’s 50 top ranked websites from different categories.

**Internal Validity**: One potential threat is that the survey used in the user-based study may not render in full resolution when viewed on small screen devices, potentially impacting the results. To mitigate this threat, I asked participants to enter the device they used for answering the survey, so that I could isolate those results. However, only a small minority of the participants did not use a desktop or laptop, and the results from the few who used a mobile phone or tablet to answer the survey did not indicate any anomalous responses. Another potential threat is the use of GMFT and PSIT in $M$Fix to determine mobile friendly problems and mobile friendliness score. However, the PSIT is the only publicly available tool that reports a mobile friendliness score. Bing only offers a web interface for detecting mobile friendly problems, unlike GMFT, which provides an API. Furthermore, GMFT and PSIT are stable tools that are used by Google to rank pages in their own search results.

**Construct Validity**: A potential threat is that the layout distortion objective used in $M$Fix quantifies the aesthetic value of a page, which is a subjective aspect of a web page. To address this threat, I conducted two user-based studies (i.e., Experiment Two) to qualitatively understand the impact of layout distortion on the visual appeal of a page. A second potential threat is that the numbers supplied by participants in response to the readability and attractiveness ratings that I asked them to provide are also subjective. To mitigate this threat, I used relative values given by the participants for the before and after repair versions for the subjects, as opposed to using their absolute values. That is, although two participants may supply different numbers for the ratings for the same pair of web pages, they will supply higher values for one of the pages if they believe that readability/attractiveness is better for that page. A third potential threat is that I
used screenshots of the subject pages in the user-based study as opposed to allowing the users to interact with the pages on mobile devices. I selected this mechanism as I wanted the users to visualize the before and after repair versions of the pages next to each other to allow for an easy comparison. Also, I did not have control over participants’ mobile devices and wanted to avoid variations in results that this could cause. A fourth potential threat is that the participants have a bias in selecting the repaired version based on the order in which the original and repaired versions are presented in the survey. To mitigate this threat, I randomized the order of the two versions for each question of the survey. A final potential threat is that the definition of correct repair used in MFix may be different from developer intent. However, this threat is mitigated by the user study results discussed in RQ3, which show that the repairs generated by MFix were considered visually appealing and preferred by the participants.

5.4 Conclusion

To summarize, in this chapter I introduced an approach, MFix, for the automated repair of MFPs in web pages. MFix performs a search for finding the repair for MFPs using an approximation algorithm. Candidate repairs in the search are selected using Gaussian perturbation around the value suggested by the GMFT. The fitness function is comprised of two objectives. First, maximizing the mobile friendliness score given by the PSIT. Second, minimizing the amount of change between the layout of a page containing a candidate repair versus the layout of the original page. The amount of change in the layout of a page is calculated using constraints encoded by the graph-based models of the page’s layout. For building the graph-based models, MFix first segments the page into areas that form natural visual groupings. It then builds graph-based models of the HTML elements within each segment and also among the different segments in a page. In the evaluation, I found that MFix was effective in resolving mobile friendly problems for 95% of the subjects and required only an average of five minutes per subject. In a user study, the participants overwhelmingly preferred the repaired version of the website for use on mobile devices and also considered the repaired page to be significantly more readable than the original. Overall, these results are strong and support the hypothesis of my dissertation by showing that this approach using search-based techniques can help developers to improve the mobile friendliness of their web pages, while maintaining a usable layout.
Chapter 6

TFix: Repair of Internationalization Presentation Failures (IPFs)

Web applications enable companies to easily establish a global presence. To more effectively communicate with this global audience, companies often employ internationalization (i18n) frameworks for their websites, which allow the websites to provide translated text or localized media content. However, because the length of translated text differs in size from text written in the original language of the page, the page’s appearance can become distorted. HTML elements that are fixed in size may clip text or look too large, while those that are not fixed can expand, contract, and move around the page in ways that are inconsistent with the rest of the page’s layout. Such distortions, called Internationalization Presentation Failures (IPFs), reduce the aesthetics or usability of a website and occur frequently — a recent study reports their occurrence in over 75% of internationalized web pages [47]. Avoiding presentation problems, such as these, is important. Studies show that the design and visual attractiveness of a website affects users’ impressions of its credibility and trustworthiness, ultimately impacting their decision to spend money on the products or services that it offers [67, 70, 71].

Repairing IPFs poses several challenges for web developers. First, modern web pages may contain hundreds, if not thousands, of HTML elements, each with several CSS properties controlling their appearance. This makes it challenging for developers to accurately determine which elements and properties need to be adjusted in order to resolve an IPF. Assuming that the relevant elements and properties can be identified, the developers must still carefully construct the repair. Due to complex and cascading interactions between styling rules, a change in one part of a web page User Interface (UI) can easily introduce further issues in another part of the page. This means that any potential repair must be evaluated in the context of not only how well it resolves the targeted IPF, but also its impact on the rest of the page’s layout as a whole. This
task is complicated because it is possible that more than one element will have to be adjusted together to repair an IPF. For example, if the faulty element is part of a series of menu items, then all of the menu items may have to be adjusted to ensure their new styling matches that of the repaired element.

Existing techniques targeting internationalization problems, such as GWALI [49], are only able to detect IPFs, and cannot generate repairs. Meanwhile other web page repair approaches target fundamentally different UI problems and are not capable of repairing IPFs. These include XFix [97], which repairs cross-browser issues; and PhpRepair [142] and PhpSync [126], which repair malformed HTML.

To address these limitations, I present an approach, I-Fix, for automatically repairing IPFs in web pages. I-Fix is designed to handle the practical and conceptual challenges particular to the IPF domain: To identify elements whose styling must be adjusted together, I designed a novel style-based clustering approach that groups elements based on their visual appearance and DOM characteristics. To find repairs, I designed a guided search-based technique that efficiently explores the large solution space defined by the HTML elements and CSS properties. This technique is capable of finding a repair solution that best fixes an IPF while avoiding the introduction of new layout problems. To guide the search, I designed a fitness function that leverages existing IPF detection techniques and UI change metrics. In an evaluation of the implementation of I-Fix, I found that it was effective at repairing IPFs, resolving over 98% of the detected IPFs; and also fast, requiring about four minutes on average to generate the repair. In a user study of the repaired web pages, I found that the repairs met with high user approval — over 70% of user responses rated the repaired pages as better than the faulty versions. Overall, these results are positive and indicate that I-Fix can help developers automatically resolve IPFs in web pages.

6.1 Background

Developers internationalize web applications by isolating language-specific content, such as text, icons, and media, into resource files. Different sets of resource files can then be utilized depending on the user’s language — a piece of information supplied by their browser — and inserted into placeholders in the requested page. This isolation of language-specific content allows a developer to design a universal layout for a web page, easing its management and maintenance, while also modularizing language specific processing.

However, the internationalization of web pages can distort their intended layout because the length of different text segments in a page can vary depending on their language. An increase in
the length of a text segment can cause it to overflow the HTML element in which it is contained, be clipped, or spill over into surrounding areas of the page. Alternatively, the containing element may expand to fit the text, which can, in turn, cause a cascading effect that disrupts the layout of other parts of the page. IPFs can affect both the usability and the aesthetics of a web page. An example is shown in Figure 6.1b. Here, the text of the page in Figure 6.1a has been translated, but the increased number of characters required by the translated text pushes the final link of the navigation bar under an icon, making it difficult to read and click. Internationalization can also cause non-layout failures in web pages, such as corrupted text, inconsistent keyboard shortcuts, and incorrect/missing translations. ZFdx does not target these non-layout related failures as I see the solutions as primarily requiring developer intervention to provide correct translations.

The complete process of debugging an IPF requires developers to (1) detect when an IPF occur in a page, (2) localize the faulty HTML elements that are causing the IPF to appear, and (3) repair the web page by modifying CSS properties of the faulty elements to ensure that the failure no longer occurs. An existing technique, GWALI [49], has been shown to be an accurate detection and localization technique for IPFs, i.e., it addresses the first and second part of the debugging process described above. The inputs to GWALI are a baseline (untranslated) page, which represents a correct rendering of the page, and a translated version (page under test (PUT)), which is analyzed for IPFs. To detect IPFs, GWALI builds a model called a Layout Graph (LG), which captures the position of each HTML element in a web page relative to the other elements. Each node of the graph represents a visible HTML element, while an edge between two nodes is annotated with a type of visual layout relationship (e.g., “East of”, “intersects”, “aligns with”, “contains” etc.) that exists between the two elements. After building the LGs for the two versions of a page, GWALI compares them and identifies edges whose annotations are different in the PUT. A difference in annotations indicates that the relative positions of the two elements are different, signaling a potential IPF. If an IPF is detected, GWALI outputs a list of HTML elements that are most likely to have caused it. ZFdx leverages the output of GWALI to initialize the repair process.

Assuming that an IPF has been detected and localized, there are several strategies developers can use to repair the faulty HTML elements. One of these is to change the translation of the original text, so that the length of the translated text closely matches the original. However, this solution is not normally applicable for two reasons. Firstly, the translation of the text is not always under the control of developers, having typically been outsourced to professional translators or to an automatic translation service. Secondly, a translation that matches the original text length may not be available. Therefore a more typical repair strategy is to adapt the layout of the
internationalized page to accommodate the translation. To do this, developers need to identify the right sets of HTML elements and CSS properties among the potentially faulty elements, and then search for new, appropriate values for their CSS properties. Together, these new values represent a language specific CSS patch for the web page. To ensure that the patch is employed at runtime, developers use the CSS \texttt{:lang()} selector. This selector allows developers to specify alternative values for CSS properties based on the language in which the page is viewed. Although this repair strategy is relatively straightforward to understand, complex interactions among HTML elements, CSS properties, and styling rules make it challenging to find a patch that resolves all IPFs without introducing new layout problems or significantly distorting the appearance of a web UI. This challenge motivates my approach, which I present in the next section.

6.2 Specialization of the Generalized Approach, *Fix

In this section, I provide details of my approach for repairing IPFs in web pages that is based on the generalized repair approach, *Fix, explained in Chapter 3. The two prerequisites for repairing presentation failures, Detection and Localization, can be instantiated using existing techniques, such as GWALI [49]. For completeness, I summarize GWALI’s detection and localization algorithm in Section 6.2.1. My contribution is in developing a repair approach for finding suitable fixes for IPFs in web pages using search-based techniques. Section 6.2.2 discusses this approach in more detail.

6.2.1 Detection and Localization of IPFs

For implementing the Detection and Localization phases of the debugging process, I our prior work, GWALI [49]. For completeness, I provide a summary of GWALI's detection and localization algorithm and its evaluation results.

GWALI takes as input a PUT and a baseline version (untranslated) of the page that shows its correct layout. To detect and localize IPFs, GWALI first builds graph-based models called layout graphs of the layout of each of these pages. GWALI models the layout of a page as a complete graph, where the nodes are HTML and text elements in the page and edges represent the visual relationships, such as alignment, direction and containment, between the elements. After building the layout graphs for the PUT and the baseline, GWALI performs a heuristic-based matching to find corresponding nodes in the two layout graphs. GWALI then compares the two layout graphs to identify differences between them. These differences represent potentially faulty elements. Rather than performing a comprehensive pair-wise comparison of the edges in
the graph, GWALI compares subgraphs of nodes and edges that are spatially close in the layout graph. Finally, GWALI analyzes and filters the elements identified by the graph comparison to produce a ranked list of elements for the developer.

In the evaluation on 54 real-world subject web pages, GWALI was accurate – detecting IPFs with 91% precision and 100% recall, and identifying the faulty element with a median rank of three. GWALI was also fast – performing detection and localization for a given web page in 9.75 seconds.

6.2.2 Repair of IPFs

The goal of IFFix is to automatically repair IPFs that have been detected in a translated version of a web page. As described in Section 6.1, a translation can cause the text in a web page to expand or contract, which leads to text overflow, element movement, incorrect text wrapping, and misalignment. The placement and the size of elements in a web page is controlled by their CSS properties. Therefore, these failures can be fixed by changing the value of the CSS properties of elements in a page to allow them to accommodate the new size of the text after translation.

Finding these new values for the CSS properties is complicated by several challenges. The first challenge is that any kind of style change to one element must also be mirrored in stylistically related elements. This is illustrated in Figure 6.1 that shows an IPF on the DMV homepage (https://www.dmv.ca.gov), when translated from English to Spanish. To correct the overlap shown in Figure 6.1b, the text size of the word “Informacion” can be decreased, resulting in the layout shown in Figure 6.1c. However, this change is unlikely to be visually appealing to an end user since the consistency of the header appearance has been changed. Ideally, the change in Figure 6.1d is preferred, which subtly decreases the font size of all of the stylistically related elements in the header. This challenge requires that my solution identify groupings of elements that are stylistically similar and adjust them together in order to maintain the aesthetics of a web page. The second challenge is that a change for any particular IPF may introduce new layout problems into other parts of the page. This can happen when the elements surrounding the area of the IPF move to accommodate the changed size of the repaired element. This challenge is compounded when there are multiple IPFs in a page or there are many elements that must be adjusted together, since multiple changes to the page increase the likelihood that the final layout will be distorted. This challenge requires that the solution find new values for the CSS properties that fix IPFs while avoiding the introduction of new layout problems.

Two insights into these challenges guide the design of IFFix. The first insight is that it is possible to automatically identify elements that are stylistically similar through an approach that
uses traditional density based clustering techniques. I designed a clustering technique that is based on a combination of visual aspects (e.g., elements’ alignment) and DOM-based metrics (e.g., XPath similarity). This allows I\textsuperscript{F}IX to accurately group stylistically similar elements that need to be changed together to maintain the aesthetic consistency of a web page’s style. The second insight is that it is possible to quantify the amount of distortion introduced into a page by IPFs and use this value as a fitness function to guide a search for a set of new CSS values. I designed I\textsuperscript{F}IX’s fitness function using existing detectors for IPFs (i.e., GWALI [49]) and other metrics for measuring the amount of difference between two UI layouts. Therefore, the goal of I\textsuperscript{F}IX’s search-based approach is to find a solution (i.e., new CSS values) that minimizes this fitness function.

Figure 6.2 shows an overview of I\textsuperscript{F}IX and shows the instantiations of the different abstraction points (AP1–4) of the I\textsuperscript{F}IX approach. The inputs to the approach are: a version of the web page (baseline) that shows its correct layout, a translated version (PUT) that exhibits IPFs, and a list of HTML elements of the PUT that are likely to be faulty. The last input can be
provided either by a detection technique, such as GWALI, or manually by developers. Developers could simply provide a conservative list of possibly faulty HTML elements, but the use of an automated detection technique allows the debugging process to be fully automated. I-Fix begins by analyzing the PUT and identifying the stylistically similar clusters that include the potentially faulty elements. Then, the approach performs a guided search to find the best CSS values for each of the identified clusters. When the search terminates, the best CSS values obtained from all of the clusters are converted to a web page CSS repair patch and provided as the output of the approach. I now explain the parts of the approach in more detail in the following subsections.

6.2.3 Identifying Stylistically Similar Clusters

The goal of this step is to group HTML elements in the page that are visually similar into Sets of Stylistically Similar Elements (SimSets). To group a page’s elements into SimSets, I-Fix computes visual similarity and DOM information similarity between each pair of elements in the page. I designed a distance function that quantifies the similarity between each pair of elements $e_1$ and $e_2$ in the page. Then I-Fix uses a density-based clustering technique to determine which elements are in the same SimSet. After computing these SimSets, I-Fix identifies the SimSet associated with each faulty element reported by GWALI. This subset of the SimSets serves as an input to the search.

Different techniques can be used to group HTML elements in a web page. A naive mechanism is to put elements having the same style class attribute into the same SimSet. In practice I found that the class attribute is not always used by developers to set the style of similar elements, or in some cases, it is not matching for elements in the same SimSet. There are several more sophisticated techniques that may be applied to group related elements in a web page, such as Vision-based Page Segmentation (VIPS) [55], Block-o-Matic [143], and R-Trees [103]. These techniques rely on elements’ location in the web page and use different metrics to divide the web page into multiple segments. However, these techniques do not produce sets of visually similar
elements as needed by I\textsc{Fix}. Instead, they produce sets of web page segments that group elements that are located closely to each other and are not necessarily similar in appearance. The clustering in I\textsc{Fix} uses multiple visual aspects to group the elements, while the aforementioned techniques rely solely on the location the elements, which makes them unsuitable for I\textsc{Fix}.

To identify stylistically similar elements in the page, I\textsc{Fix} uses a density-based clustering technique, DBSCAN [68]. A density-based clustering technique finds sets of elements that are close to each other, according to a predefined distance function, and groups them into clusters. Density-based clustering is well suited for I\textsc{Fix} for several reasons. First, the distance function can be customized for the problem domain, which allows I\textsc{Fix} to use style metrics instead of location. Second, this type of clustering does not require prior knowledge of the number of clusters, which is ideal for I\textsc{Fix} since each stylistically similar group may have a different number of elements, making the total number of clusters unknown beforehand. Third, the clustering technique puts each element into only one cluster (i.e., hard clustering). This is important because if an element is placed into multiple SimSets, the search could define multiple change values for it, which may prevent the search from converging if the changes are conflicting.

I\textsc{Fix}'s distance function uses several metrics to compute the similarity between pairs of elements in a page. At a high-level, these metrics can be divided into two types of similarity: (1) similarity in the visual appearance of the elements, including width, height, alignment, and CSS property values and (2) similarity in the DOM information, including XPath, HTML class attribute, and HTML tag name. I include DOM related metrics in the distance function because only using visual similarity metrics may produce inaccurate clusters in cases where the elements belonging to a cluster are intentionally made to appear different. For example, to highlight the link of the currently rendered page from a list of navigational menu links. Since the different metrics have vastly different value ranges, I\textsc{Fix} normalizes the value of each metric to a range [0,1], with zero representing a match for the metric and 1 being the maximum difference. The overall distance computed by the function is the weighted sum of each of the normalized metric values. The metrics’ weights were determined based on experimentation on a set of web pages and are the same for all subjects. Next, I provide a detailed description of each of the metrics I\textsc{Fix} uses in the distance function.

### 6.2.3.1 Visual Similarity Metrics

These metrics are based on the similarity of the visual appearance of the elements. I\textsc{Fix} uses three types of visual metrics to compute the distance between two elements $e_1$ and $e_2$. These are:
Elements’ width and height match: Elements that are stylistically similar are more likely to have matching width and/or height. I Fix defines width and height matching as a binary metric. If the widths of the two elements $e_1$ and $e_2$ match, then the width metric value is set to 0, otherwise it is set to 1. The height metric value is computed similarly.

Elements’ alignment match: Elements that are similar are more likely to be aligned with each other. This is because browsers render a web page using a grid layout, which aligns elements belonging to the same group either horizontally or vertically. Alignment includes left edge alignment, right edge alignment, top edge alignment, and bottom edge alignment. These four alignment metrics are binary metrics, so they are computed in a way similar to the width and height metrics.

Elements’ CSS properties similarity: Aspects of the appearance of the elements in a web page, such as their color, font, and layout, are defined in the CSS properties of these elements. For this reason, elements that are stylistically similar typically have the same values for their CSS properties. I Fix computes the similarity of the CSS properties as the ratio of the matching CSS values over all CSS properties defined for both elements. For this metric, I Fix only considers explicitly defined CSS properties, so it does not take into account default CSS values and CSS values that are inherited from the body element in the web page. These values are matching for all elements and are not helpful in distinguishing elements of different SimSets.

6.2.3.2 DOM Information Similarity Metrics

These metrics are based on the similarity of features defined in the DOM of the web page. I Fix uses three types of DOM related metrics to compute the distance between two elements $e_1$ and $e_2$. These are:

Elements’ tag name match: Elements in the same SimSet have the same type, so the HTML tag names for them need to match. HTML tag names are used as a binary metric, i.e., if $e_1$ and $e_2$ are the same tag name, then the metric value is set to 0, otherwise it is set to 1.

Elements’ XPath similarity: Elements that are in the same SimSet are more likely to have similar XPaths. The XPath similarity between two elements quantifies the commonality in the ancestry of the two elements. In HTML, elements in the page inherit CSS properties from their parent elements and pass them on to their children. More ancestors in common between two elements means more inherited styling information is shared between them. To compute XPath distance, I Fix uses the Levenshtein distance between elements’ XPath. More formally, XPath distance is the minimum number of HTML tags edits (insertions, deletions or substitutions) required to change one XPath into the other.
Elements’ class attribute similarity: As mentioned earlier, an HTML element’s class attribute is often insufficient to group similarly styled elements. Nonetheless, it can be a useful signal; therefore I use class attribute similarity as one of the metrics for style similarity. An HTML element can have multiple class names for the class attribute. Our approach computes the similarity in class attribute as the ratio of class names that are matching over all class names that are set.

6.2.4 Candidate Solution Representation

A repair for the PUT is represented as a collection of changes for each of the SimSets identified by the clustering technique. More formally, I define a potential repair as a candidate solution, which is a set of change tuples. Each change tuple is of the form $\langle S, p, \Delta \rangle$ where $\Delta$ is the change value that $T\text{Fix}$ applies to a specific CSS property $p$ for a particular SimSet $S$. The change value can be positive or negative to represent an increase or decrease in the value of $p$. Note that a candidate solution can have multiple change tuples for the same SimSet as long as they target different CSS properties.

An example candidate solution is $(\langle S_1, \text{font-size}, -1 \rangle, \langle S_1, \text{width}, 0 \rangle, \langle S_1, \text{height}, 0 \rangle, \langle S_2, \text{font-size} \rangle, -1 \rangle, \langle S_2, \text{width}, 10 \rangle, \langle S_2, \text{height}, 0 \rangle)$. This candidate solution represents a repair to the PUT that decreases the font-size of the elements in $S_1$ by one pixel, decreases the font-size of the elements in $S_2$ by one pixel, and increases the width of the elements in $S_2$ by ten pixels. Note that the value “0” means that there is no change to the elements in the SimSet for the specified property.

6.2.5 Fitness Function

To evaluate each candidate solution, $T\text{Fix}$ first generates a PUT’ by adjusting the elements of the PUT based on the values in the candidate solution. The approach then calculates the fitness score of the PUT’ when it is rendered in a browser. I now describe both these steps in detail.

6.2.5.1 Generating the PUT’

To generate the PUT’, $T\text{Fix}$ modifies the PUT according to the values in the candidate solution that will subsequently be evaluated. The approach also modifies the width and the height of any ancestor element that has a fixed width or height that prevents the children elements from expanding freely. An example of such an ancestor element is shown in Figure 6.3. In the example,
increasing the width of the elements in SimSet $S$ requires modification to the fixed width value of the ancestor $div$ element in order to make space for the children elements’ expansion.

To modify the elements that need to be changed in the PUT, $\mathcal{IX}$ uses the following algorithm. $\mathcal{IX}$ iterates over each change tuple $\langle S, p, \Delta \rangle$ in the candidate solution and modifies the elements $e \in S$ by changing their CSS property values: $e.p = e.p + \Delta$. Then $\mathcal{IX}$ computes the cumulative increase in width and height for all the elements in $S$ and determines the new coordinates $\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle$ of the Minimum Bounding Rectangles (MBRs) of each element $e$. Then $\mathcal{IX}$ finds the new position of the right edge of the rightmost element $max(e_{x_2})$, and the new position of the bottom edge of the bottommost element $max(e_{y_2})$. After that, $\mathcal{IX}$ iterates over all the ancestors of the elements in $S$. For each ancestor $a$, if $a$ has a fixed value for the width CSS property and $max(e_{x_2})$ is larger than $a_{x_2}$, then $\mathcal{IX}$ increases the width of the ancestor $a.width = a.width + (max(e_{x_2}) - a_{x_2})$. A similar increase is applied to the height, if the ancestor has a fixed value for the height CSS property and $max(e_{y_2})$ is larger than $a_{y_2}$.

### 6.2.5.2 Fitness Function Components

As mentioned earlier, a challenge in fixing IPFs is that any change to fix a particular IPF may introduce layout problems into other parts of the page. In addition, larger changes that are applied to the page make it more likely that the final layout will be distorted. This motivates the goal of the fitness function, which is to minimize the differences between the layout of the PUT and the layout of the baseline while making minimal amount of changes to the page.

To address this goal, $\mathcal{IX}$’s fitness function involves two components. The first is the Amount of Layout Inconsistency component. This component measures the impact of IPFs by quantifying the dissimilarity between the PUT’s layout and the baseline layout. The second part of the fitness...
function is the *Amount of Change* component. This component quantifies the amount of change the candidate solution applies to the page in order to repair it. To combine the two components of the fitness function, *IFix* uses a prioritized fitness function model in which minimizing the amount of layout inconsistency has a higher priority than minimizing the amount of change. The amount of layout inconsistency is given higher priority because it is strongly tied with resolving the IPFs, which is the goal of *IFix*, while amount of change component is used after resolving the IPFs to make the changes as minimal as possible. The prioritization is done by using a sigmoid function to scale the amount of change to a fraction between 0 and 1 and adding it to the amount of layout inconsistency value. Using this, the overall fitness function is equal to \( \text{amount of layout inconsistency} + \text{sigmoid}(\text{amount of change}) \). I now describe the components of the fitness function in more detail.

**Amount of Layout Inconsistency:** This component represents a quantification of the dissimilarity between the baseline and the PUT’ LGs. To compute the value for this component, *IFix* computes the coordinates of the MBRs of each element and the inconsistencies in the PUT as reported by GWALI. Then *IFix* computes the distance (in pixels) required to make the relationships in the two LGs match. The number of pixels is computed for every inconsistent relationship reported by GWALI. For alignment inconsistencies, if two elements \( e_1 \) and \( e_2 \) are top-aligned in the baseline and not top-aligned in the PUT’, *IFix* computes the difference in the vertical position of the top side of the two elements \( |e_1y_1 - e_2y_1| \). A similar computation is performed for bottom-alignment, right-alignment, and left-alignment. For direction inconsistencies, if \( e_1 \) is situated to the “West” of \( e_2 \) in the baseline, and is no longer “West” in the PUT’, *IFix* computes the number of pixels by which \( e_2 \) needs to move to be to the West of \( e_1 \), which is \( e_1x_2 - e_2x_1 \). A similar computation is performed for East, North, and South relationships. For containment inconsistencies, if \( e_1 \) bounds (i.e., contains) \( e_2 \) in the baseline, and no longer bounds it in the PUT’, *IFix* computes the vertical and horizontal expansion needed for each side of \( e_1 \)’s MBR to make it bound \( e_2 \). The number of pixels computed for each of these inconsistent relationships (alignment, directional, and bounding) is added to get the total amount of layout inconsistency.

**Amount of Change:** This component represents the amount of change a candidate solution causes to the page. To compute this amount, *IFix* calculates the percentage of change that is applied to each CSS property for every modified element in the page. The total amount of change is the summation of the squared percentages of changes. The intuition behind squaring the percentages of change is to penalize solutions more heavily if they represent a large change.
6.2.6 Search

The goal of the search is to find values for the CSS properties of each SimSet that make the baseline page and the PUT have LGs that are matching with minimal changes to the page. \texttt{TFix} generates candidate solutions using the search operations I define in this section. Then \texttt{TFix} evaluates each candidate solution it generates using the fitness function to determine if the candidate solution produces a better version of the PUT.

The approach operates by going through multiple iterations of the search. In each iteration, the approach generates a population of candidate solutions. Then, the approach refines the population by keeping only the best candidate solutions and performing the search operations on them for another iteration. The search terminates when a termination condition is satisfied. After the search terminates, the approach returns the best candidate solution in the population. More formally, the iteration includes five main steps (1) initializing the population, (2) fine-tuning the best solution using local search, (3) performing mutation, (4) selecting the best set of candidate solutions, (5) and terminating the search if a termination condition is satisfied. The following is a description of each step in more detail:

**Initializing the population:** This step creates an initial population of candidate solutions that \texttt{TFix} performs the search on. The goal of this step is to create a diverse initial population that allows the search to explore different areas of the solution space. Figure 6.4 shows an overview of the process of initializing the population. In the figure, the first set of candidate solutions represents modifications to the elements that are computed based on text expansion that occurred to the PUT. To generate this set of candidate solutions, \texttt{TFix} computes the average percentage of text expansion in the elements of each SimSet that includes a faulty element. Then \texttt{TFix} generates three candidate solutions based on the expansion percentage. The first candidate solution increases the width of the elements in the SimSets by a percentage equal to the percentage of the text expansion. The second candidate solution increases the height by the same percentage. The third candidate solution decreases the font-size of the elements in the SimSets by the same percentage. The rest of the candidate solutions in the initial population (i.e., fourth candidate solution in the figure) are generated by creating copies of the current candidate solutions and mutating the copies using the mutation operation described in the mutation step below.

**Fine tuning using local search:** This step works by selecting the best candidate solution in the population and fine tuning the change values $\Delta$ in it in order to get the best possible fix. To do this, \texttt{TFix} uses the Alternating Variable Method (AVM) local search algorithm [82, 84]. \texttt{TFix} performs local search by iterating over all the change tuples in the candidate solution and for each change tuple it tries a new value in a specific direction (i.e., it either increases or decreases the
change value $\Delta$ for the CSS property), then evaluates the fitness of the new candidate solution to determine if it is an improvement. If there is an improvement, the search keeps trying larger values in the same direction. Otherwise, it tries the other direction. This process is repeated until the search finds the best possible change values $\Delta$ based on the fitness function. The newly generated candidate solution is added to the population.

**Mutation:** The goal of the mutation step is to diversify the population and explore change values that may not be reached during the AVM search. ZFix performs standard Gaussian mutation operations to the change values in the candidate solutions. It iterates over all the candidate solutions in the population and generates a new mutant for each one. ZFix creates a mutant by iterating over each tuple in the candidate solution and changing its value with a probability of $1 / (\text{number of change tuples})$. The new change value is picked randomly from a Gaussian distribution around the old value. The newly generated candidate solutions are added to the population to be evaluated in the selection step.

**Selection:** ZFix evaluates all of the candidate solutions in the current population and selects the best $n$ candidate solutions, where $n$ is the predefined size of the population. The best candidate solutions are identified based on the fitness function described in Section 6.2.5.2. The selected candidate solutions are used as the population for the next iteration of the search.

**Termination:** The algorithm terminates when either of two conditions are satisfied. The first condition is when a predefined maximum number of iterations is reached. This condition is used to bound the execution time of the search and prevents it from running for a long time without converging to a solution. The second condition is when the search reaches a saturation point (i.e., no improvement in the candidate solutions for multiple consecutive iterations). In this cases, the search most likely converged to the best candidate solution it could find, and further iterations will not introduce more improvement.
Table 6.1: Subjects used in the evaluation of \texttt{IFIX}

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>URL</th>
<th>#HTML</th>
<th>Baseline</th>
<th>Translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>akamai</td>
<td><a href="https://www.akamai.com">https://www.akamai.com</a></td>
<td>304</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>2</td>
<td>caLottery</td>
<td><a href="http://www.calottery.com">http://www.calottery.com</a></td>
<td>777</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>3</td>
<td>designSponge</td>
<td><a href="http://www.designspunge.com">http://www.designspunge.com</a></td>
<td>1,184</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>4</td>
<td>dmv</td>
<td><a href="https://www.dmv.ca.gov">https://www.dmv.ca.gov</a></td>
<td>638</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>5</td>
<td>doctor</td>
<td><a href="https://sfplasticsurgeon.com">https://sfplasticsurgeon.com</a></td>
<td>689</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>6</td>
<td>els</td>
<td><a href="https://www.els.edu">https://www.els.edu</a></td>
<td>483</td>
<td>English</td>
<td>Portuguese</td>
</tr>
<tr>
<td>7</td>
<td>facebookLogin</td>
<td><a href="https://www.facebook.com">https://www.facebook.com</a></td>
<td>478</td>
<td>English</td>
<td>Bulgarian</td>
</tr>
<tr>
<td>8</td>
<td>flynas</td>
<td><a href="http://www.flynas.com">http://www.flynas.com</a></td>
<td>1,069</td>
<td>English</td>
<td>Turkish</td>
</tr>
<tr>
<td>9</td>
<td>googleEarth</td>
<td><a href="https://www.google.com/earth">https://www.google.com/earth</a></td>
<td>323</td>
<td>Italian</td>
<td>Russian</td>
</tr>
<tr>
<td>10</td>
<td>googleLogin</td>
<td><a href="https://accounts.google.com">https://accounts.google.com</a></td>
<td>175</td>
<td>English</td>
<td>Greek</td>
</tr>
<tr>
<td>11</td>
<td>hightail</td>
<td><a href="https://tinyurl.com/y9tpmr07">https://tinyurl.com/y9tpmr07</a></td>
<td>1,135</td>
<td>English</td>
<td>German</td>
</tr>
<tr>
<td>12</td>
<td>hotwire</td>
<td><a href="https://www.hotwire.com">https://www.hotwire.com</a></td>
<td>583</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>13</td>
<td>ixigo</td>
<td><a href="https://www.ixigo.com/flight">https://www.ixigo.com/flight</a></td>
<td>1,384</td>
<td>English</td>
<td>Italian</td>
</tr>
<tr>
<td>14</td>
<td>linkedin</td>
<td><a href="https://www.linkedin.com">https://www.linkedin.com</a></td>
<td>586</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>15</td>
<td>mplay</td>
<td><a href="http://www.myplay.com">http://www.myplay.com</a></td>
<td>3,223</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>16</td>
<td>museum</td>
<td><a href="https://www.amnh.org">https://www.amnh.org</a></td>
<td>585</td>
<td>English</td>
<td>French</td>
</tr>
<tr>
<td>17</td>
<td>qualitrol</td>
<td><a href="http://www.qualitrolcorp.com">http://www.qualitrolcorp.com</a></td>
<td>401</td>
<td>English</td>
<td>Russian</td>
</tr>
<tr>
<td>18</td>
<td>rentalCars</td>
<td><a href="http://www.rentalcars.com">http://www.rentalcars.com</a></td>
<td>1,011</td>
<td>English</td>
<td>German</td>
</tr>
<tr>
<td>19</td>
<td>skype</td>
<td><a href="https://tinyurl.com/yceuuxhso">https://tinyurl.com/yceuuxhso</a></td>
<td>495</td>
<td>English</td>
<td>French</td>
</tr>
<tr>
<td>20</td>
<td>skyScanner</td>
<td><a href="https://www.skyscanner.com">https://www.skyscanner.com</a></td>
<td>388</td>
<td>French</td>
<td>Malay</td>
</tr>
<tr>
<td>21</td>
<td>twitterHelp</td>
<td><a href="https://support.twitter.com">https://support.twitter.com</a></td>
<td>327</td>
<td>English</td>
<td>French</td>
</tr>
<tr>
<td>22</td>
<td>westin</td>
<td><a href="https://tinyurl.com/ycc4qo8ar">https://tinyurl.com/ycc4qo8ar</a></td>
<td>815</td>
<td>English</td>
<td>Spanish</td>
</tr>
<tr>
<td>23</td>
<td>worldsBest</td>
<td><a href="http://www.theworlds50best.com">http://www.theworlds50best.com</a></td>
<td>581</td>
<td>English</td>
<td>German</td>
</tr>
</tbody>
</table>

\texttt{IFIX} could fail to find an acceptable fix under two scenarios. The first scenario is when \texttt{GWALI} does not include the actual faulty HTML element in its reported list. \texttt{IFIX} assumes that the initial set of elements provided as the input contains the faulty elements. If this assumption is violated, \texttt{IFIX} will not be able to find a repair. The second scenario is when the search does not converge to an acceptable fix. This could occur due to the non-determinism of the search.

### Evaluation

To assess the effectiveness and performance of \texttt{IFIX}, I conducted an empirical evaluation on 23 real-world subject web pages and answered three research questions:

**RQ1:** How effective is \texttt{IFIX} in reducing IPFs?

**RQ2:** How long does it take for \texttt{IFIX} to generate repairs?

**RQ3:** What is the quality of the fixes generated by \texttt{IFIX}?
6.3.1 Implementation

I implemented the approach in Java as a prototype tool named $Fix$ [2]. I used the *Apache Commons Math3* library implementation of the DBSCAN algorithm to group similarly styled HTML elements. I used *Javascript* and *Selenium WebDriver* for dynamically applying candidate fix values to the pages and for extracting the rendered Document Object Model (DOM) information, such as element MBRs and XPath. I used the *jStyleParser* library for extracting explicitly defined CSS properties for HTML elements in a page. For obtaining the set of IPFs, I used the latest version of GWALI [49]. For the search technique described in Section 6.2.2, I used the following parameter values: population size = 100, mutation rate = 1.0, max number of iterations = 20, and saturation point = 2. For the Gaussian distribution, used by the mutation operator, I used a 50% decrease and increase as the min and max values, and $\sigma = (\text{max} - \text{min})/8.0$ as the standard deviation. For clustering, I used the following weights for the different metrics: 0.1 for width/height and alignment, 0.3 for CSS properties similarity, 0.4 for tag name, 0.3 for XPath similarity, and 0.2 for class attribute similarity.

6.3.2 Subjects

For the evaluation I used 23 real-world subject web pages as shown in Table 6.1. The column “#HTML” shows the total number of HTML elements in the subject page, which was counted by parsing the subject page’s DOM for node type “element”. This gives a rough estimate of the page’s size and complexity. The column “Baseline” shows the language of the subject used in the baseline version that shows the correct appearance of the page, and “Translated” shows the language that exhibits IPFs in the subject with respect to the baseline. I gathered these subjects from the web pages used in the evaluation of GWALI [49]. The main criteria behind selecting this source was the presence of known IPFs in the study of GWALI and the diversity in size, layouts, and translation languages that the GWALI subjects offered. The 54 subjects used in the evaluation of GWALI were collected from three different sources: (1) *builtwith.com*, (2) Alexa top 100 most visited websites, and (3) manual sampling of popular travel-related and telecom company websites. Out of the total 54 subject pages used in the evaluation of GWALI, I filtered and selected only those web pages for which at least one IPF was reported.

6.3.3 Experiment One

To answer RQ1 and RQ2, I ran $Fix$ on each subject and recorded the set of IPFs before and after each run, as reported by GWALI, and measured the total time taken. To minimize the variance
Table 6.2: Effectiveness of IFFix in reducing IPFs

<table>
<thead>
<tr>
<th>Name</th>
<th>#Before</th>
<th>#After (Average Reduction in %)</th>
<th>Rand (Variation 1)</th>
<th>NoClust (Variation 2)</th>
<th>Rand-NoClust (Variation 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>akamai</td>
<td>6</td>
<td>0 (100)</td>
<td>2 (74)</td>
<td>0 (100)</td>
<td>0.20 (97)</td>
</tr>
<tr>
<td>caLottery</td>
<td>4</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>1 (70)</td>
<td>0.73 (81)</td>
</tr>
<tr>
<td>designSponge</td>
<td>9</td>
<td>0.07 (99)</td>
<td>3 (63)</td>
<td>0.07 (99)</td>
<td>3 (71)</td>
</tr>
<tr>
<td>dmv</td>
<td>18</td>
<td>0 (100)</td>
<td>4 (78)</td>
<td>2 (85)</td>
<td>9 (41)</td>
</tr>
<tr>
<td>doctor</td>
<td>21</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>6 (72)</td>
<td>21 (0)</td>
</tr>
<tr>
<td>els</td>
<td>6</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>facebookLogin</td>
<td>16</td>
<td>0 (100)</td>
<td>6 (65)</td>
<td>12 (25)</td>
<td>16 (0)</td>
</tr>
<tr>
<td>flynas</td>
<td>9</td>
<td>0 (100)</td>
<td>0.07 (99)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>googleEarth</td>
<td>15</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>4 (72)</td>
<td>7 (55)</td>
</tr>
<tr>
<td>googleLogin</td>
<td>6</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>highTail</td>
<td>2</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>hotwire</td>
<td>30</td>
<td>0 (100)</td>
<td>0.47 (98)</td>
<td>4 (87)</td>
<td>4 (87)</td>
</tr>
<tr>
<td>ixigo</td>
<td>38</td>
<td>12 (68)</td>
<td>12 (68)</td>
<td>0 (100)</td>
<td>12 (68)</td>
</tr>
<tr>
<td>linkedin</td>
<td>22</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>12 (46)</td>
<td>19 (13)</td>
</tr>
<tr>
<td>mplay</td>
<td>76</td>
<td>0.40 (99)</td>
<td>3 (96)</td>
<td>3 (95)</td>
<td>51 (33)</td>
</tr>
<tr>
<td>museum</td>
<td>32</td>
<td>0.40 (99)</td>
<td>0 (100)</td>
<td>12 (63)</td>
<td>19 (40)</td>
</tr>
<tr>
<td>qualitrol</td>
<td>19</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>21 (-9)</td>
<td>22 (-16)</td>
</tr>
<tr>
<td>rentalCars</td>
<td>6</td>
<td>0 (100)</td>
<td>2 (74)</td>
<td>0 (100)</td>
<td>1 (99)</td>
</tr>
<tr>
<td>skype</td>
<td>3</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>skyScanner</td>
<td>4</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
</tr>
<tr>
<td>twitterHelp</td>
<td>5</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0 (100)</td>
<td>0.17 (97)</td>
</tr>
<tr>
<td>westin</td>
<td>11</td>
<td>1 (91)</td>
<td>1 (91)</td>
<td>1 (91)</td>
<td>1 (91)</td>
</tr>
<tr>
<td>worldsBest</td>
<td>24</td>
<td>0 (100)</td>
<td>7 (69)</td>
<td>0 (100)</td>
<td>17 (29)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>16</td>
<td>0.6 (98)</td>
<td>2 (90)</td>
<td>3 (82)</td>
<td>8 (65)</td>
</tr>
</tbody>
</table>

in the results that can be introduced from the non-deterministic aspects of the search, I ran IFFix on each subject 30 times and used the mean values across the runs in the results. To further assess and understand the effectiveness of the two main features of IFFix, guided search and style similarity clustering, I conducted more experiment runs with three variations to IFFix. The first variation replaced the guided search in the approach with a random search to evaluate the benefit of guided search with a fitness function. For every subject, I time bounded the random search by terminating it once the average time required by IFFix for that subject had been utilized. The second variation removed the clustering component from IFFix to evaluate the benefit of clustering stylistically similar elements in a page. The third variation combined the first and second variation. Similar to IFFix, I ran the three variations 30 times on each subject. All of the experiments were run on a 64-bit Ubuntu 14.04 machine with 32GB memory, Intel Core i7-4790 processor, and screen resolution of $1920 \times 1080$. For rendering the subject web pages, I used Mozilla Firefox v46.0.01 with the browser window maximized to the screen size.
For RQ1, I used GWALI to determine the initial number of IPFs in a subject and the number of IPFs remaining after each of the 30 runs. I calculated the reduction in IPFs as a percentage of the before and after values for each subject.

For RQ2, I computed the average total running time of $\mathcal{I}F\mathcal{i}x$ and variation 2 across 30 runs for each subject. I did not compare the performance of $\mathcal{I}F\mathcal{i}x$ with its first and third variations since I time bounded their random searches, as described above. I also measured the time required for the two main stages in $\mathcal{I}F\mathcal{i}x$: clustering stylistically similar elements (Section 6.2.3) and searching for a repair patch (Section 6.2.6).

6.3.3.1 Presentation of Results

Table 6.2 shows the results for RQ1. The initial number of IPFs are shown under the column “#Before”. The columns headed “#After” show the average number of IPFs remaining after each of the 30 runs of $\mathcal{I}F\mathcal{i}x$ for its three variations: “Rand”, “NoClust”, and “Rand-NoClust”. (Since it is an average, the results under “#After” columns may show decimal values.) The average percentage reduction is shown in parenthesis.

6.3.3.2 Discussion of Results

The results show that $\mathcal{I}F\mathcal{i}x$ was the most effective in reducing the number of IPFs, with an average 98% reduction, compared to its variations. This shows the effectiveness of $\mathcal{I}F\mathcal{i}x$ in resolving IPFs.

The results also strongly validate my two key insights of using guided search and clustering in the approach. The first key insight was validated as $\mathcal{I}F\mathcal{i}x$ was able to outperform a random search that had been given the same amount of time. $\mathcal{I}F\mathcal{i}x$ was substantially more successful in primarily two scenarios. First, pages (e.g., dmv and facebookLogin) containing multiple IPFs concentrated in the same area that require a careful resolution of the IPFs by balancing the layout constraints without introducing new IPFs. Second, pages (e.g., akamai) that have strict layout constraints, permitting only a very small range of CSS values to resolve the IPFs. I also found that, overall, the repairs generated by random search were not visually pleasing as they often involved a substantial reduction in the font-size of text, indicating that guidance was helpful for $\mathcal{I}F\mathcal{i}x$. This observation was also reflected in the total amount of change made to a page, captured by the fitness function, which reported that random search introduced 28% more changes, on average, compared to $\mathcal{I}F\mathcal{i}x$. The second key insight of using a style-based clustering technique was validated as $\mathcal{I}F\mathcal{i}x$ not only rendered the pages more visually consistent compared to its non-clustered variations, but also increased the effectiveness by resolving a relatively higher number of IPFs.
Out of the 23 subjects, ZF1x was able to completely resolve all of the reported IPFs in 18 subjects in each of the 30 runs and in 21 subjects in more than 90% of the runs. I investigated the two subjects, ixigo and westin, where ZF1x was not able to completely resolve all of the reported IPFs. I found that the dominant reason for the ixigo subject was false positive IPFs that were reported by GWALI. This occurred because the footer area of the page had significant differences in terms of layout and structure between the baseline and translated page. Therefore CSS changes made by ZF1x were not sufficient to resolve the IPFs in the footer area. For the westin subject, elements surrounding the unrepaired IPF were required to be modified in order to completely resolve it. However, these elements were not reported by GWALI, thereby precluding ZF1x from finding a suitable fix.

The total running time of ZF1x ranged from 73 seconds to 17 minutes, with an average of just over 4 minutes and a median of 2 minutes. ZF1x was also three times faster, on average, than its second variation (no clustering). This was primarily because clustering enabled a narrowing of the search space by grouping together potentially faulty elements reported by GWALI that were also stylistically similar. Thereby a single change to the cluster was capable of resolving multiple IPFs. Moreover, the clustering overhead in ZF1x was negligible, requiring less than a second, on average. The detailed timing results can be found at the project website [2].

6.3.4 Experiment Two

For addressing RQ3, I conducted a user study to understand the visual quality of ZF1x’s suggested fixes from a human perspective. The general format of the survey was to present, in random order, an IPF containing a UI snippet from a subject web page before and after repair. The participants
were then asked to compare the two UI snippets on a 5-point Likert scale with respect to their appearance similarity to the corresponding UI snippet from the baseline version. Each UI snippet showing an IPF was captured in context of its surrounding region to allow participants to view the IPF from a broader perspective. Examples of UI snippets are shown in Figure 6.1b and Figure 6.5.

To select the “after” version of a subject, I used the run with the best fitness score across the 30 runs of iFix in Experiment One.

To figure out the number of IPFs to be shown for each subject, I manually analyzed the IPFs reported by GWALI and identified groups of IPFs that shared a common visual pattern. I called these groups “equivalence classes”. Figure 6.5 shows an example of an equivalence class from the Hotwire subject, where the two IPFs caused by the price text overflowing the container are highly similar. One IPF from each equivalence class was presented in the survey.

To make the survey length manageable for the participants, I divided the 23 subjects over five different surveys, with each containing four or five subjects. The participants of the user study were 37 undergraduate level students. Each participant was assigned to one of the five surveys. The participants were instructed to use a desktop or laptop for answering the survey to be able to view the IPF UI snippets in full resolution.

### 6.3.4.1 Presentation of Results

The results for the appearance similarity ratings given by the participants for each of the IPFs in the 23 subjects are shown in Figure 6.6. On the x-axis, the ID and number of IPFs for a subject are shown. For example, 4a, 4b, and 4c represent the dmv subject with three IPFs. The blue colored bars above the x-axis indicate the number of ratings in favor of the after (repaired) version. The dark blue color shows participants’ response for the after version being much better than the before version, while the light blue color shows the response for the after version being somewhat better than the before version. Similarly, the red bars below the X-axis indicate the number of ratings in favor of the before repair version, with dark and light red showing the response for the before version being much and somewhat better than the after version, respectively. The gray bars show the number of ratings where the participants responded that the before and after versions had the same appearance similarity to the baseline. For example, IPF 23a had a total of 11 responses, six for the after version being much better, three for the after version being somewhat better, one reporting both the versions as the same, and one reporting the before version as being somewhat better. As can be seen from Figure 6.6, 64% of the participant responses favored the after repair versions, 21% favored the before repair versions, and 15% reported both versions as the same.
6.3.4.2 Discussion of Results

The results of the user study show that the participants largely rated the after (repaired) pages as better than the before (faulty) versions. This indicates that IFIX generates repairs that are high in visual quality. The IPFs presented in the user study, however, do not comprehensively represent all of the IPFs reported for the subjects as the surveys only contained one representative from each equivalence class. Therefore I weighted the survey responses by multiplying each response from an equivalence class with the size of the class. The results are shown in Figure 6.7. With the weighting, 70% responses show support for the after version. Also, interestingly, the results show the strength of support for the after version — 41% of responses rate the after version as much better, while only 5% responses rate the before version as much better.

Two of the IPFs, 3b and 23b, had no participant responses in favor of the after version. I inspected these subjects in more detail and found that the primary reason for this was that IFIX substantially reduced the font-size (e.g., from 13px to 5px for 3b) to resolve the IPFs. Although these changes were visually unappealing, I was able to confirm that these extreme changes were the only way to resolve the IPFs. I also found that IPFs, 7a, 19a, and 22b, had a majority of...
the participant responses reporting both versions as the same. ZFix was unable to resolve 22b, implying that the before and after versions were practically the same. The issue with 7a and 19a was slightly different. Both IPFs were caused by guidance text in an input box being clipped because the translated text exceeded the size of the input box. Unless the survey takers could understand the target language translation, there was no way to know that the guidance text was missing words.

6.3.5 Threats to Validity

The first potential threat is the use of only GWALI for detecting IPFs. However, there exist no other available automated tools that can detect IPFs and report potentially faulty HTML elements. Another potential threat is that I manually categorized IPFs into equivalence classes for the user study. However, this categorization was fairly straightforward, and in practice there was no ambiguity regarding membership in an equivalence class, for example, as shown in Figure 6.5. To further support this, I have made the surveys and subject pages publicly available [2] for verification. A potential threat to construct validity is that I presented UI snippets of the subject pages to the participants, rather than full-page screenshots, which might have an impact on their appearance similarity ratings. I opted for this mechanism as the full page screenshots of the subjects were large in size, making it difficult to view all three screenshots, baseline, before (faulty), and after (repaired), in one frame for comparison. The benefit of this mechanism was that it allowed the participants to focus only on the areas of the pages that contained IPFs and were thus modified by ZFix.
6.4 Conclusion

In summary, in this chapter I presented the design of my approach, \texttt{IFix}, for the automated repair of IPFs in web pages. IPFs are distortions in the intended appearance of a web page that are caused by the relative expansion or contraction of translated text. \texttt{IFix} uses guided search-based techniques for finding repairs for IPFs. \texttt{IFix} uses text expansion based heuristics to identify initial candidate solutions. It then uses the AVM search to fine tune the best candidate solution in the population and mutation to diversify the population. The fitness function used to guide the search is designed based on existing detectors for IPFs (i.e., GWALI) and other metrics for measuring the amount of difference between two UI layouts. In the evaluation, \texttt{IFix} was able to resolve 98\% of the reported IPFs. In a user study, 70\% of the participants rated the fixed versions as better than the unfixed versions. Overall, these results are positive and support the hypothesis of my dissertation by showing that this approach using search-based techniques can help developers automatically resolve IPFs in web pages while maintaining the pages’ aesthetic consistency.
Chapter 7

\textit{G}Fix: Repair of Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs)

An attractive and visually appealing appearance is important for the success of a website. A recent study by Google underscores this point by noting that the average visitor forms a first impression of a web page within the first 50 milliseconds of visiting a page [152] — an amount of time that is heavily influenced by a page’s aesthetics. Other studies report that the appearance of a web application’s User Interface (UI) plays an important role in shaping end users’ perception of the services and products it offers [151, 87]. Companies put significant effort into the look and feel of their websites, employing graphic designers and illustrators to carefully craft their layout and graphics. Presentation failures — a discrepancy between the actual appearance of a website and its intended appearance — can undermine this effort and negatively impact end users’ perception of the quality of the site and the services it delivers. These types of failures can also impact the usability of a web application’s UI or be symptomatic of underlying logic or data problems.

The UIs of modern web applications are highly complex and dynamic. Back-end server code dynamically generates content and client-side browsers render this content based on complex HTML, CSS, and JavaScript rules. A typical web page may contain hundreds or thousands of HTML elements. In turn, each of these elements may contain definitions of several dozen Cascading Style Sheets (CSS) properties ranging over a large set of possible values that control the element’s appearance. These attributes can also interact with each other via cascading style rules, and other modern layout mechanisms, such as floating elements, overlays, and dynamic sizing. This complexity makes it challenging for developers to debug presentation failures. The sheer amount of HTML elements and style attributes that could be faulty makes it a labor intensive task, and the complex interactions of the CSS rules and layout mechanisms make it difficult to
accurately identify the relationship between an observed failure and the underlying HTML code responsible for that failure. To assist in this effort, developers have tools, such as Firebug [22], that can compute useful HTML and CSS information for a web page. Although helpful, such a debugging process remains a developer-driven process, and its accuracy and speed are dependent on the developer’s expertise. For example, to use Firebug effectively, developers must be able to determine which HTML elements to investigate, understand the effects of the various style properties defined by those elements, and then repair them by performing the necessary modifications so that the page renders correctly.

Developers have several techniques available to them to help detect and localize presentation failures, but cannot generate repairs. Moreover, these techniques have limitations in even detecting presentation failures that either reduce their effectiveness or make them inappropriate for general usage. For example, many techniques are focused on one type of presentation failure, such as Cross Browser Issues (XBIs) (e.g., [137, 59, 141]), or a limited and predefined set of application-independent failure types (e.g., Fighting Layout Bugs [149]), and cannot detect other types of presentation failures. Other techniques can only support debugging efforts where there is a prior working version that can be compared against (e.g., [76, 147]). Finally, a group of techniques require testers to exhaustively specify all correctness properties to be checked (e.g., Selenium [42], Crawljax [119, 116, 120], Cucumber [19], and Sikuli [57, 165], which is labor-intensive and potentially error-prone.

To address these limitations, I propose a novel approach, GFIX, to assist developers in debugging presentation failures. The advantages of GFIX are that it is fully automated and more widely applicable than previous approaches in terms of the types of presentation failures it can be used with. I frame the repair approach as a search-based problem where the identified answer is a new suggested fix value for the faulty HTML element and CSS style property that causes a presentation failure. My key insight into transforming this problem into a search-based problem is that image processing techniques can be used to compare the amount of deviation between a rendered page and its intended appearance. This difference can then be used as a fitness function to guide the exploration of likely repairs, and when the two have no differences, a successful repair has been identified. In the evaluation, I found that the GFIX approach is accurate – it was able to resolve 94% of the presentation failures reported by WebSee [103], an automated tool for detecting presentation failures. In a user study of the repaired web pages, I found that the repairs met with high user approval – 79% of user responses rated the repaired pages as better than the faulty versions. Overall, these results are positive and indicate that GFIX can help developers automatically resolve presentation failures in web pages.
7.1 Background and Motivation

After detecting a presentation failure, developers must debug their web applications to identify the underlying fault. In modern web applications, the appearance of a web page is defined by HTML tags and CSS properties that specify how each HTML tag will be rendered by the browser. Therefore, when a developer is debugging a page under test (PUT), they are trying to identify a new value for a CSS property of an HTML element that is set incorrectly. This is represented by a repair tuple, $⟨e, p, v, v'⟩$, where $e$ is an HTML element in the PUT, $p$ is a CSS property of $e$, $v$ is a value of $p$, and $v'$ is suggested value for $p$ for fixing the presentation failure. In the remainder of this section I describe two scenarios in which developers need to debug web pages in order to identify repairs for an observed presentation failure.

The first scenario is regression debugging. Developers often perform maintenance on their web pages in order to introduce new features, correct a bug, or refactor the HTML structure. For example, a refactoring may convert a $⟨$table$⟩$ based layout to one based on $⟨$div$⟩$ tags or convert HTML 4 tags to HTML 5. During this transformation, a developer may introduce a fault that causes a type of presentation failure called the Regression Debugging Problem (RDP). Developers must debug the UI to repair such RDPs. Existing techniques, such as Cross Browser Testing (XBT) [141, 59, 137], GUI differencing [161, 76, 75], automated oracle comparators [147, 146], or tools based on diff may be of limited use in this scenario. The reason for this is that these techniques use a tree-based representation (e.g., Document Object Model (DOM)) to compare the versions of the faulty web page. If a faulty change is small and localized within the tree, it may be straightforward for these techniques to identify the fault. However, if the tree structure has changed significantly (as in the above refactoring example) then a comparison will most likely result in many spurious changes being identified as the fault. Furthermore, these techniques assume that any difference between the tree-based representation implies a failure. This is not always true as there can be multiple ways to implement the same visual appearance using HTML and CSS properties.

The second scenario is mockup-driven development, which occurs during the initial development of a web page’s templates and user interfaces. In some development shops, front-end developers are guided by design mockups, which are detailed renderings of the intended appearance of a web page [123, 92, 130]. Front-end developers are expected to create “pixel-perfect” matches of these mockups [24] using web development tools, such as Adobe Muse, Amaya, or Visual Studio, which back-end developers can then modify by adding dynamic content. During this process, both types of developers need to ensure that their changes have not caused the page to look different than the mockup. Discrepancies in the actual appearance of the page from its
mockup cause Mockup-driven Development Problems (MDDPs) to occur. Tree-based debugging techniques, such as those discussed earlier, would not be applicable in this scenario because there are no prior working versions against which to compare the DOMs. Other well-known techniques, such as Selenium [42], Cucumber [19], Crawljax [116, 119, 120], Sikuli [165, 57], graphical automated oracles [64], or Cornipickle [78], are not practical in this scenario because they require all correctness properties to be exhaustively specified, which is labor intensive. Furthermore, the correctness properties are expressed in terms of HTML syntax, not the visual appearance of an element. Therefore, these techniques may miss presentation failures, such as incorrect inheritance of an ancestor element’s CSS properties.

7.2 Specialization of the Generalized Approach, *GFix*

In this section, I provide details of my approach, *GFix*, for repairing MDDPs and RDPs (MRPFs)\(^1\) in web pages. *GFix* is a specialization of the *F\(\)ix approach explained in Chapter 3. To provide the input functions \(D\) and \(L\), I developed a technique WebSee [103, 104] that detects and localizes MRPFs in web pages automatically. For completeness, I summarize WebSee’s detection and localization algorithm in Section 7.2.1. I then explain the repair approach, *G\(\)Fix*, in detail in Section 7.2.2.

7.2.1 Detection and Localization of MDDPs and RDPs

Testers have several techniques available to them to help detect and localize MRPFs. However, these techniques have limitations that either reduce their effectiveness or make them inappropriate for general usage. For example, many techniques are focused on one type of presentation failure, such as XBIs (e.g., [137]), or a limited and predefined set of application-independent failure types (e.g., Fighting Layout Bugs [149]), and cannot detect MRPFs. Other techniques can only support debugging efforts where there is a prior working version that can be compared against (e.g., [161, 147]). Finally, a group of techniques require testers to exhaustively specify all correctness properties to be checked (e.g., Selenium [42], Crawljax [119], Cucumber [19], and Sikuli [57], which is labor-intensive and potentially error-prone. To address these limitations, I developed a novel approach, WebSee [103, 104] to assist developers in detecting and localizing MRPFs. WebSee applies techniques from the field of computer vision to analyze the visual representation of a web page, identify MRPFs, and then determine which elements in the HTML source of the page could be responsible for the observed failures.

\(^1\)Henceforth, in this chapter I use MRPFs to collectively refer to MDDPs and RDPs
From a high-level, WebSee takes two inputs, a PUT and an appearance oracle (O) that specifies the visual correctness properties of the PUT, and processes them through three phases to locate MRPFs. The first phase, detection, compares the visual representations of PUT and O to detect a set of differences in the PUT. The second phase, localization, analyzes a rendering map of PUT to identify the set of HTML elements that define the difference pixels. Finally, the third phase, result set processing, prioritizes the identified set of elements and provides this as an output to the developer.

To detect MRPFs, WebSee uses Perceptual Image Differencing (PID), a computer vision based technique for image comparison. PID uses computational models of the human visual system to compare two images [163, 164]. This allows the approach to compare the visual representations of the PUT and O based on an idea of “similarity” that corresponds to humans’ visual concept of similarity. To compare a given pair of images, PID models three features of the human visual system: (1) spatial sensitivity, (2) luminance sensitivity, and (3) color sensitivity. The PID algorithm also accepts a threshold value \(\Delta\) as a parameter, which is used to decide whether the images are below a threshold of perceptible difference, and a field of view value in degrees \(F\), which indicates how much of the observer’s view the screen occupies. The PID technique is particularly well-suited for our problem for two reasons. First, the three modeled features roughly account for the location (or size), contrast, and color of the HTML elements in the two pages, which together cover almost all possible visual rendering effects available via CSS or HTML. Second, the \(\Delta\) and \(F\) allow the difference detection to be scaled to reduce false positives (via \(\Delta\)) and account for visual representation sizes that are either very small (e.g., smartphone) or large (e.g., desktop web browser). WebSee uses the PID algorithm to compare the visual representations of O and PUT at a tolerance level specified by \(\Delta\) and \(F\). The result of this is a set \(DP\) that contains all pixels of the two images considered to be perceptually different.

Next, WebSee identifies a set of HTML elements in the PUT that may be responsible for the detected presentation failure. The general intuition is to identify the HTML elements whose rendering area includes the difference pixels \(DP\). To do this, WebSee builds a Rectangle-tree (R-tree) model of the rendered PUT. An R-tree is a height-balanced tree data structure that is widely used in the spatial database community to store multidimensional information [77, 53]. In this case, the multidimensional data is the bounding rectangle that corresponds to the rendering area of an HTML element. The leaves of the R-tree are the HTML elements of the page and the non-leaf nodes are bounding rectangles that contain groups of nearby elements. For each \((x, y)\) pixel in \(DP\), the containing HTML elements are found by traversing the R-tree’s edges and added to a set of potentially faulty HTML elements, \(E\). The elements in the set \(E\) are then ordered
based on their likelihood of being the faulty element, since not all of the elements in $E$ are equally likely to contain the fault. To perform this prioritization, WebSee utilizes heuristics based on the relationship of elements that contain difference pixels. For example, prioritizing elements with a higher percentage of their pixels identified as difference pixels.

I evaluated the accuracy of WebSee using real-world mockups provided by an industrial partner and on several hundred faults seeded into well-known web apps, including Gmail, Craigslist, Oracle, Virgin America, and USC CS Research. For the test cases with the real-world mockups, WebSee performed strongly, 45% of the faulty elements were listed in the top 5 and 70% were within the top 10. For the seeded faults, WebSee was able to detect all of the failures and 93% of the time was able to return a set of potentially faulty HTML elements that contained the original seeded fault. I also evaluated WebSee in the context of a user study, where users had to manually perform the detection and identification of the potentially faulty elements. The users were only able to visually detect 76% of the failures, while WebSee detected all of the failures. Furthermore, the users could generate a set containing the original seeded fault only 36% of the time, while WebSee was able to correctly identify the fault 93% of the time. WebSee was also much faster, needed an average of 87 seconds to perform this analysis versus an average of 7 minutes for the users.

### 7.2.2 Repair of MDDPs and RDPs

The goal of $G \text{Fix}$ is to find potential fixes that can repair the detected MRPFs. The $G \text{Fix}$ approach treats the repair as a search-based problem where the identified solutions — modifications to the web page’s CSS — are potential fixes for the observed MRPFs. When the search finds the correct CSS value, applying it to the failing page will cause the rendering of that page to match an oracle representing the intended appearance.

Finding these correct values for the CSS properties is complicated by several challenges. The first challenge is that a repair for any particular MRPF may introduce new layout problems into other parts of the page. This can happen when the elements surrounding the area of the MRPF move to accommodate the changed size of a repaired element. This challenge is compounded when there are multiple MRPFs in a page or there are many elements that must be adjusted together, since multiple changes to the page increase the likelihood that the final layout will be distorted. This challenge requires that $G \text{Fix}$ must be able to find new values for the CSS properties that balance between fixing MRPFs and avoiding the introduction of new layout problems. My key insight is that it is possible to quantify the amount of distortion introduced into a page by MRPFs and use this value as a fitness function to guide a search for a set of new CSS values. The idea is
to use image comparison techniques, such as the one provided by WebSee [103], to measure the amount of distortion via the size of the difference pixels set. When the number of difference pixels is zero, the rendering of the web page matches the oracle, implying that a fault has likely been identified and repaired. The second challenge is to find the repair in a reasonable amount of time. Exploring every possible CSS property is not practical as the CSS properties can range over a large set of possible values, thereby making this a time and resource intensive approach. To address this challenge I have two key insights. The first key insight is that the visual differences between the PUT and its appearance oracle can be analyzed to identify a set of relevant CSS properties that are likely to have caused the MRPF in the PUT. Since the relevant CSS properties are a subset of all possible properties, this can help decrease the run time of $\mathcal{G}$Fix. The second key insight is that groups of CSS properties differ in ways that can allow specialized search techniques to be designed to identify repairs in a short amount of time.

Formally, MRPFs are caused by one or more root causes. A root cause is a tuple, $\langle e, p, v \rangle$, where $e$ is a faulty HTML element in the page, $p$ is a CSS property of $e$, and $v$ is the value of $p$. Given a set of potential root Causes, $\mathcal{G}$Fix tries to find a set of fixes that resolve the observed MRPFs. A fix is defined by the tuple, $\langle e, p, v, v' \rangle$, where $\langle e, p, v \rangle \in$ root Causes and $v'$ is a suggested new value for $p$.

At a high level, $\mathcal{G}$Fix works by first identifying a set of possible root causes for the failures detected in the PUT. Then the approach utilizes two forms of guided search to find the best repair. The first search examines each root cause in isolation to find viable candidate fixes. The search takes the CSS property of the root cause and finds a new value for it that minimizes the number of difference pixels between the page and the oracle. The second search then seeks to find a combination of candidate fixes identified in the first phase that can minimize the number of MRPFs reported in the page. The second search is necessary since not all candidate fixes may be required, as the CSS properties involved may have duplicate or competing effects. For instance, the CSS properties margin-top and padding-top may both be identified as root causes for a
presentation failure, but can be used to achieve similar outcomes — meaning that only one may actually need to be included in the repair. Conversely, other candidate fixes may be required to be used in combination with one another to fully resolve an MRPF. For example, an HTML element may need to be adjusted for both its \textit{width} and \textit{height}. Furthermore, candidate fixes produced for one MRPF may interfere with those for another, or even introduce additional and unwanted MRPFs. By searching through different combinations of candidate fixes, the second search aims to produce a suitable subset — a repair — that overall minimizes the number of MRPFs in a page when applied together.

![Illustrative example](image)

Figure 7.2: Illustrative example

Figure 7.2 shows an example web application that I will use to illustrate the $\mathcal{G}$Fix approach. Figure 7.2a shows the intended rendering of a web page, which is used as an oracle in the approach. A screenshot of the appearance of the web page under development is shown in Figure 7.2b. By visual inspection, one can determine that there are three presentation failures: (1) the location of the ‘Sign in’ button has changed, (2) the color of text in bottom box is different, and (3) the style of text, ‘Cellphone advertisement’, has changed. The three presentation failures are shown in Figure 7.2c by areas marked A, B, and C, respectively.

7.2.2.1 Overall Algorithm

Algorithm 3 shows the overall algorithm of my approach. The approach takes four inputs. The first input is the web page under test, \textit{PUT}. The form of the \textit{PUT} is a URL that points to either a location on the network or filesystem where all of the HTML, CSS, JavaScript, and media files of the \textit{PUT} can be accessed. The second input is the oracle (\textit{O}) that specifies the visual correctness properties of the \textit{PUT}. The form of \textit{O} is an image. This oracle could be the design mockup used by the front-end developers or a screenshot of the previously correct version of the \textit{PUT}. The third input is a function, \textit{D}, that can detect MRPFs by comparing the visual representations
Algorithm 3 Overall Algorithm

**Input:** PUT: Web page under test

O: Oracle for PUT

D: Function to detect MRPFs in the PUT

L: Function to localize MRPFs to faulty HTML elements in the PUT

**Output:** PUT’: Modified PUT with repair applied

1: /* Stage 1 — Initialization */
2: DP ← D (PUT, O)
3: E ← L (PUT, DP)
4: while true do
5:   rootCauses ← {}
6:   for each e ∈ E do
7:     props ← getCSSProperties (e, DP)
8:       for each p ∈ props do
9:         v ← getValue (e, p, PUT)
10:        rootCauses ← rootCauses ∪ {(e, p, v)}
11:      end for
12:   end for
13: /* Stage 2 — Search for Candidate Fixes */
14: candidateFixes ← {}
15:   for each ⟨e, p, v⟩ ∈ rootCauses do
16:     if p ∈ SizeAndPositionProperties then
17:       v’ ← sizeAndPositionAnalysis ((e, p, v), PUT, O, D)
18:     else if p ∈ ColorProperties then
19:       v’ ← colorAnalysis ((e, p, v), PUT, O, D)
20:     else if p ∈ PredefinedValuesProperties then
21:       v’ ← predefinedValuesAnalysis ((e, p, v), PUT, O, D)
22:     end if
23:     candidateFixes ← candidateFixes ∪ {(e, p, v, v’)}
24:   end for
25: /* Stage 3 — Search for Best Combination of Candidate Fixes */
26: repair ← searchForBestCombination (candidateFixes, PUT, O, D)
27: /* Stage 4 — Check Termination Criteria */
28: PUT’ ← applyRepair(PUT, repair)
29: DP’ ← D (PUT’, O)
30: E’ ← L (PUT’, DP’)
31: if |DP’| = 0 or (|DP’| = |DP| and E’ = E) then
32:   return PUT’
33: else
34:   DP ← DP’
35:   E ← E’
36:   PUT ← PUT’
37: end if
38: end while
of the PUT and O and report a set of differences at the pixel-level. The function D can be provided in different ways, for example using pixel-to-pixel equivalence comparison techniques, perceptual image differencing techniques, or manually by a tester. In G\textsc{Fix}, I use the detection module of WebSee [103], which uses the perceptual image differencing technique, to define D. The last input is a function, L, that can localize the detected MRPFs and report a ranked list of HTML elements in the PUT ordered by their likelihood of being faulty. Strictly speaking, prioritization is not required for the approach, however, it can assist the approach to find a repair faster. The prioritization can be provided in many ways. For example, “suspiciousness” could be computed using statistical based fault localization techniques or a tester could rank elements based on debugging experience. In G\textsc{Fix}, I leverage WebSee [103] to implement L, which uses a set of heuristics to prioritize the faulty elements in the PUT.

The overall algorithm, shown by Algorithm 3, comprises four stages, as shown by the overview diagram in Figure 7.1. The figure also shows the instantiations of the different abstraction points (AP1–4) of the *\textsc{Fix} approach.

\textbf{Stage 1: Initialization}

The initial part of the algorithm (lines 2–12) involves extracting a list of root causes relevant to the detected MRPFs in PUT. To do this, the approach begins by identifying the areas of visual differences between O and the rendered appearance of PUT (line 2) by invoking the detection function D. The approach then obtains a ranked list of potentially faulty HTML elements, E, by invoking the localization function L (line 3). To detect and localize MRPFs, i.e., for D and L, the approach uses the WebSee tool [103, 104]. Then the approach builds a list of root causes by iterating over each element \( e \in E \) and identifying CSS properties relevant to the detected MRPF (shown as “getCSSProperties” at line 7). Identifying relevant CSS properties is challenging in the domain of MRPFs since the oracle is available in the form of an image, which lacks the details of the underlying HTML elements and CSS properties that define the page’s appearance. Moreover, the functions D and L do not provide any descriptors indicating the nature of the MRPFs observed.

To address these challenges, I have designed the “getCSSProperties” (line 7) function based on the insight that the \textit{visual symptoms} of MRPFs on a page can be analyzed to identify the relevant CSS properties. Determining the relevant CSS properties is analogous to diagnosing a sick patient. In this analogy, I can observe \textit{visual symptoms} — an observable and quantifiable feature of the PUT’s appearance, such as color usage frequency, or text size — that can guide G\textsc{Fix} in identifying the applicable CSS properties. I describe the process of analyzing visual
Symptoms to identify relevant CSS properties in detail in Section 7.2.2.2. Each relevant CSS property forms the basis of one root cause. It is added to the running set rootCauses, with the value, $v$, extracted from the rendered PUT (lines 10).

**Stage 2: Search for Candidate Fixes**

This stage produces individual candidate fixes for each root cause (lines 14–24), comprising the first phase search. This stage works on each root cause, $\langle e, p, v \rangle$, in isolation to find a new fix value for the root cause that is optimized according to a fitness function, with the aim of minimizing the visual difference between PUT and $O$. The optimal fix value, $v'$, is returned as the output of this stage.

The design of the search algorithm for this stage is based on the key insight that tailored search algorithms can be designed for groups of CSS properties that have a common visual impact on the appearance of an HTML element. This allows $\mathcal{G}$FIX to effectively and efficiently identify candidate fixes. Based on this insight, I have developed specialized search functions for the three categories by considering the unique aspects of the categories. The three search functions are described in Sections 7.2.2.3, 7.2.2.4 and 7.2.2.5, respectively.

Each of the three search functions take as input the root cause tuple, $\langle e, p, v \rangle$, the page under test, PUT, the oracle, $O$, and the detection function, $D$. The search function attempts to find a new candidate fix value, $v'$, for $p$ (lines 17, 19, and 21). The search functions use the number of difference pixels reported by $D$ as the fitness function to guide the search.

**Stage 3: Search for Best Combination of Candidate Fixes**

The goal of the second search phase (represented by a call to “searchForBestCombination” on line 26) is to identify a subset of candidateFixes that when applied together minimize the overall number of MRPFs reported for the PUT. This stage takes as input the candidateFixes set produced by stage 2, the PUT, $O$, and $D$ and produces a set, repair, a subset of candidateFixes.

This stage is designed using a biased random search. The search begins by creating a pool of candidate repairs. A candidate repair is generated by adapting the roulette wheel technique using stochastic acceptance [89]. A candidate fix is included in the repair with a probability $imp_{fix}/imp_{max}$. Here, $imp_{fix}$ is the improvement in fitness score when the candidate fix was evaluated in the first phase search (stage 2), and $imp_{max}$ is the maximum improvement in fitness score observed across all of the candidate fixes. All of the candidate repairs in the pool are evaluated using the function $D$ as the fitness function and the candidate repair resulting in the lowest fitness score is selected as the best repair, which is returned as the output of the search.

105
Prior to making the design choice of using a biased random search for finding the repair, I experimented with a Genetic Algorithm (GA). However, in my experience I found that the GA took longer to converge than the biased random search, and so I used the biased random search in order to decrease the run time of $\text{GF}ix$.

**Stage 4: Check Termination Criteria**

The fourth and last stage determines whether the algorithm should terminate or proceed to another iteration the search (lines 28–37). Before checking the termination criteria, the approach first applies the repair generated by stage 3 to a copy of the $PUT$ to produce a modified version of the page, $PUT'$ (line 28). Then the approach obtains a new set of difference pixels, $DP'$, and a new list of potentially faulty HTML elements, $E'$, by invoking the detection and localization functions, $D$ and $L$, for $PUT'$ and $O$ (lines 29 and 30). The algorithm terminates under two conditions: first, all MRPFs in the page have been resolved resulting in $|DP'| = 0$ or, second, no fitness improvement was observed in this iteration of the algorithm compared to the previous iteration indicating that the approach was able to potentially only partially repair the MRPFs in the $PUT$. The modified page, $PUT'$, is returned as the output of the algorithm upon termination (line 32). If the algorithm does not terminate in the iteration, then it implies that the current repair represents an improvement that may be improved further in another iteration of the algorithm.

### 7.2.2.2 CSS Properties from Visual Symptoms of MDDPs and RDPs

Recall that in stage 1 of the approach, the visual symptoms of an MRPF observed on the $PUT$ are used to identify the CSS properties relevant to the MRPF (“getCSSProperties” at line 7 of Algorithm 3). In this section, I discuss the details of this process.

The approach of identifying the relevant CSS properties works by analyzing visual symptoms of a web page using image processing techniques. An MRPF manifests itself as a difference between the appearance of the $PUT$ and $O$. My insight is that these differences, or *visual symptoms*, are indications or clues to the underlying faulty CSS properties of the observed failure. For example, consider a MRPF where a login button is expected to be red (as specified in the oracle), but is green in the $PUT$. In this case, a visual symptom exists as there is a difference between the set of colors present in the oracle and the set of colors present in the rendering of the $PUT$. This visual symptom would likely be due to the incorrect setting of the CSS property, `background-color`, which controls the color of the button. Formally, I define a *visual symptom* as a predicate that can be either true or false in describing a visual difference between the $PUT$ and $O$. 
There are two challenges in identifying symptoms for $\mathcal{G}$\textsubscript{Fix}. First, a symptom must be independent of any particular web page. Otherwise, the usefulness of the symptom will not generalize to other web pages. To deal with this challenge, I define symptoms that are not based on a web page’s structure or content, since these can vary significantly among web pages. Second, the set of symptoms must be comprehensive enough to cover all CSS properties and also provide distinguishing power among related properties. This is challenging because there are many CSS properties. Fortunately though, many of these properties have similar visual impacts, which allows definition of broad-level groupings of symptoms. Fine-grained symptoms within each broad-level grouping further allow accurate distinction between the properties. To identify these groupings, I analyzed the set of visual properties and classified them based on their visual impact on an HTML element. For example, the CSS properties \texttt{display} and \texttt{visibility} can both make an element appear or disappear, so they are grouped in the visual impact category “Visibility.” In Table 7.1 under the column labeled “Impact Area” I show the six areas of visual impact that were identified in this process. For each of the visual impact areas, I list the visual properties that were classified into that area (column “Visual Properties”). In the third column (“Symptoms”) I then list the set of

<table>
<thead>
<tr>
<th>Impact Area</th>
<th>CSS Properties</th>
<th>Symptoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>color, background-color, border-top-color, border-left-color, background-color, border-right-color, border-bottom-color, outline-color</td>
<td>ModifiedColor</td>
</tr>
<tr>
<td></td>
<td>height, width, padding, padding-bottom, padding-top, padding-right, padding-left, border-top-width, border-bottom-width, border-left-width, border-right-width, max-height, min-height, border-spacing</td>
<td>SamePageSize, AllDPIInLeftOfElement, AllDPIInRightOfElement, AllDPIInTopOfElement, AllDPIInBottomOfElement</td>
</tr>
<tr>
<td>Position</td>
<td>margin, margin-top, margin-bottom, margin-right, margin-left, position, line-height, vertical-align, bottom, top, right, left, float</td>
<td>ShiftLeftElement, ShiftTopElement, ShiftRightElement, ShiftBottomElement</td>
</tr>
<tr>
<td>Visibility</td>
<td>display, overflow, visibility</td>
<td>VisibleElement</td>
</tr>
<tr>
<td>Text appearance</td>
<td>font-size, direction, font-family, white-space, text-align, letter-spacing, text-decoration, word-spacing, text-indent, font-weight, text-transform, font-variant</td>
<td>TextElement</td>
</tr>
<tr>
<td>Decoration style</td>
<td>outline-style, border-style-left, border-top-style, border-right-style, border-bottom-style</td>
<td>DPIInBorder, DPOutsideBorder, DPInsideBorder</td>
</tr>
</tbody>
</table>
symptoms we defined that relate to the group. I will now discuss each of the visual impact areas and their corresponding symptoms in more detail.

1. **Color Symptoms:** There are eight CSS properties that can manipulate the color of HTML elements. Developers can use these properties to change the color of text, background, border, or outline. To differentiate these CSS properties from others, the approach analyzes the color differences between the screenshot of the PUT and O. To perform the color analysis, the approach first crops the screenshot of the PUT and O to the area of the HTML element under consideration, which I refer to as PUT_e and O_e.

   **ModifiedColor:** This symptom is true when one or more RGB colors appears in PUT_e that do not appear in O_e. For example, when developers set a wrong color for an HTML element, it is possible that the color is changed to some other color that did not appear in the original web page. This causes a new color to appear in the color set of the web page, which means the symptom will be true. This symptom is also true when there is a mismatch in the frequency of colors in PUT_e and O_e, i.e., the number of pixels with the color under consideration is different. This can happen, for example, if a developer sets the value of the faulty color property to a color already present in the element, the color set of the element will remain the same as O_e but frequency would be different. The ModifiedColor symptom is calculated as the symmetric difference between the color histograms of PUT_e and O_e.

2. **Size Symptoms:** There are 14 CSS properties that are able to change the size of an HTML element. They can either directly change the HTML element by modifying its width or height, or indirectly change the element’s size by size-related properties, such as border, margin, or padding. My approach defines the following symptoms for this group.

   **SamePageSize:** This symptom is false when the size of O is not equal to that of PUT. This indicates that the MRPF changed the size of an HTML element that then led to a change in the size of the page.

   **AllDPIInTopOfElement:** This symptom is true when all of the pixels that are different between PUT and O are in the top half of an HTML element. This indicates the MRPF may have been caused by the CSS properties that refer to the top part of the element. Examples of these are border-top-width and padding-top. These types of faults either directly affect the pixels near the top of the element or indirectly affect them by moving the top either down or up in relation to its intended position. Similarly, I have analogous symptoms. **AllDPIInRightOfElement, AllDPIInLeftOfElement, and AllDPIInBottomOfElement.**
3. **Position Symptoms:** In this group, there are 12 CSS properties that are able to change the position of the HTML element. The visual impact of these properties is moving the HTML element without changing its appearance. To differentiate this group, the idea is to try to match the appearance of an HTML element in the *PUT* with a location in *O*. To do this, the approach first retrieves the position and size of the HTML element in the *PUT*. Then, the approach generates a screenshot of the *PUT* and crops it to the area of the HTML element. Lastly, the approach determines if the same cropped part of *O* matches the cropped image of the HTML element from the screenshot of *PUT*. Based on this matching, the approach derives the following symptoms.

*ShiftBottomElement:* This symptom is true when the HTML element moves downwards in the screenshot of *PUT*. When the approach is able to match the element in *O*, the approach further checks if the element has moved downwards. This happens for properties, such as `padding-top` and `margin-top`, which modify the spacing to on the top of the element likely pushing it down. Similarly, the approach defines *ShiftLeftElement*, *ShiftRightElement*, and *ShiftTopElement*.

4. **Visibility Symptoms:** There are three CSS properties in this group, and they can make an HTML element invisible (i.e., not appear in the *PUT*). To identify MRPFs that are caused by these CSS properties, the approach checks if the HTML element is shown in the *PUT*.

*VisibleElement:* This symptom is true when the HTML element is visible in the *PUT*. The approach checks that the height and width of the element are greater than zero in the browser rendering.

5. **Text Appearance Symptoms** In this group, there are 12 CSS properties related to the appearance of text. They can change the appearance of text in terms of its size, style, etc. To distinguish this group from others, the idea is to extract the text of the HTML element, then further analyze the content of the text using text processing techniques as follows.

*TextElement:* This symptom is true when the HTML element in *PUT* contains text. Because the 12 CSS properties change the appearance of the text in the HTML element, if the element does not contain any text, none of the 12 CSS properties will have any visual affect on the element, let alone cause a MRPF. Hence, this symptom equal to true is highly correlated with any of the 12 CSS properties in this group having an effect.

6. **Decoration Style Symptoms** In this group, there are six CSS properties. They control the style of the border or outline of the HTML element. One pattern I found for this group is that the visual change is likely to affect a certain area of the HTML element, such as border and
Table 7.2: Categorization of CSS properties based on visual impact

<table>
<thead>
<tr>
<th>Category</th>
<th>CSS Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size and Position Properties</td>
<td>height, max-height, max-width, min-height, min-width, width, padding-bottom, padding-top, padding-right, padding-left, border-bottom-width, border-top-width, border-left-width, border-right-width, outline-width, font-size, letter-spacing, line-height, margin-bottom, margin-top, margin-right, margin-left, bottom, top, left, right, z-index, text-indent, word-spacing</td>
</tr>
<tr>
<td>Color Properties</td>
<td>color, background-color, border-top-color, border-left-color, background-color, border-right-color, border-bottom-color, outline-color</td>
</tr>
<tr>
<td>Predefined Values Properties</td>
<td>background-attachment, background-repeat, border-bottom-style, border-top-style, border-left-style, border-right-style, outline-style, font-family, font-style, font-variant, font-weight, list-style-position, list-style-type, clear, display, float, overflow, position, visibility, border-collapse, caption-side, table-layout, direction, text-align, text-decoration, text-transform, white-space, vertical-align</td>
</tr>
</tbody>
</table>

outline (a line drawn around the HTML element outside of the border). Hence, the approach first calculates the border of the HTML element by retrieving rendering information from the DOM of the PUT. Next, the approach calculates the location of the difference pixels. The approach then checks if the difference pixels are on the border, outside the border, or inside the border of the HTML element.

**DPOnBorder:** The symptom occurs when the difference pixels are located on the border of the HTML element. This indicates the root causes contains CSS properties related to `border-style`. Similarly, the approach defines analogous symptoms, `DPOutsideBorder` and `DPInsideBorder`.

### 7.2.2.3 Size and Position Analysis

As discussed in Section 7.2.2.1, I designed a specialized search technique for finding candidate fixes for a group of CSS properties, including `margin`, `padding`, `height`, and `width`, that affect the size and position of elements ("sizeAndPositionAnalysis" at line 17 of Algorithm 3). A complete list of the CSS properties in `SizeAndPositionProperties` can be found in Table 7.2. I now explain the search analysis technique for handling the size and position properties.

The goal of this analysis is to process the root cause, \((e, p, v)\), for \(p \in SizeAndPositionProperties\), to find the fix value \(v'\). My key insight that motivates the design of the search technique for this analysis is that changes in size and position of an HTML element are directly proportional to the number of difference pixels, which is used as a fitness score to guide the search. The number of
difference pixels are reduced as the chosen value gets closer to the correct value, since the HTML element in the rendering of the PUT starts to overlap with that in $O$. On the other hand, the number of difference pixels increases or stays the same as the overlap decreases when the chosen value moves away from the correct value.

The search technique for size and position analysis is inspired by the *Alternating Variable Method (AVM)* [82, 84]. The AVM is a local search strategy that starts at an initial point in the solution space and then tries to optimize each input variable, one at a time. In the context of $\mathcal{GF}$x, the input variable is $v$ of the root cause that needs to be optimized and the initial point is a value $v_i$, calculated based on a *translation heuristic* discussed in the next paragraph. The AVM technique first tries to establish a direction of search, and then rapidly explores the space in that direction to find the optimal value. To establish a direction of search, the AVM technique performs *exploratory* moves, which adds small delta values (e.g., $[-1,1]$) to $v$, and observes the impact on the fitness score. If the fitness is observed to be improved, indicating that a potential direction for further improving the input variable has been found, the AVM switches to performing *pattern* moves. A pattern move adds values to $v$ through step sizes that increase exponentially. The pattern move continues with this exploration until the newly computed fitness score shows improvement over the previous fitness score, with the improvement implying that the search is progressing in the correct direction. If a pattern move fails to improve fitness, the AVM switches to exploratory moves to explore a new direction from the current point in search. If exploratory moves fail to improve fitness, the current point in search is returned as the best candidate fix value, $v'$ to the main algorithm (line 17 of Algorithm 3).

**Translation Heuristic:** The translation heuristic is used to identify an initial value for the AVM search that is potentially close to the expected correct value to facilitate quick convergence. On a high level, the initial value is calculated based on the translation that the elements in the PUT have undergone from their expected locations in $O$. The translation heuristic is based on the observation that a change in position or size of the rendered HTML element can be seen as a translation/displacement of pixels in the two images, $O$ and screenshot of the PUT. The translation heuristic computes the amount of translation that the difference pixels have undergone by taking a relative complement of the two images. In set theory, the relative complement of a set $A$ with respect to a set $B$ is given by the elements in $B$, but not in $A$. $(B\setminus A = \{ x \in B \mid x \notin A \})$ Thus, taking a relative complement of pixels in the PUT screenshot with respect to the pixels in $O$, we get a set $DP_o$ of difference pixels in $O$, but not in the PUT. Similarly, taking a relative complement of pixels in $O$ with respect to the PUT gives a set $DP_t$ of difference pixels in the PUT, but not in $O$. The translation value, $T$, is calculated as an average over the difference
between corresponding pixels pairs’ (x, y) values in the two complement sets, \(DP_o\) and \(DP_t\). The initial value \(v_i\) for the AVM search is set as \(T + v\), where \(v\) is the original value of the element, \(e\).

To illustrate the working of the size and position analysis, let us consider the example presented earlier (Figure 7.2). The area A marked in Figure 7.2c shows the positional shift of the “Sign in” button from left (as can be seen in O, Figure 7.2a) to right (as can be seen in the PUT screenshot, Figure 7.2b). Consider the faulty element identified for this visual difference as \(\langle \text{div style="padding-left: 75px;"} \rangle\). The correct value for the \text{padding-left} property is 5px. Let us assume the number of difference pixels in area A is 5,000. Let us say that the position of the top left corner of the “Sign in” button in O with 5px padding is (100, 100). Now with a padding-left of 75px, the new position of the top left corner is (170, 100). These are the two difference pixels present in the relative complement sets, \(DP_o\) and \(DP_t\), respectively. Taking a difference between the two pixels’ values gives a translation value of 70px. Similarly, translation values of all the difference pixels of the button are computed, and the average translation value is obtained to be -68px. With this value, the AVM search is initialized to 7px \((v + T = 75 − 68)\), which gives a fitness score of 300. Now trying a new padding-left value of 8px increases the fitness score to 450, indicating that the search is moving away from the expected fix value as 450 is greater than 300. Then trying a value 6px reduces the fitness score to 240, indicating that the search is moving in the correct direction. Eventually, the search explores the value, 5px, which reduces the number of difference pixels in the area A to 0. Thus, the search terminates and returns 5px as the candidate fix value, \(v'\), for the area A.

### 7.2.2.4 Color Analysis

The color analysis proposes a specialized search strategy for finding candidate fixes for the color related CSS properties, such as text color, background-color and border-color (“colorAnalysis” at line 19 of Algorithm 3). A complete list of the CSS properties in the \textit{ColorProperties} category can be found in Table 7.2. In this section, I discuss the search technique I designed for handling these properties.

The color analysis processes the root cause, \(\langle e, p, v \rangle\), if \(p \in \text{ColorProperties}\) to find the candidate fix value \(v'\). It is challenging to identify the correct color value since the color related CSS properties can be one of 16 million colors ranging from \#000000 to \#FFFFFF. To address this challenge, my insight is that the expected color can be identified by extracting the color in the area of \(e\) from the oracle image, \(O\). However, identifying the expected color is compounded by the fact that in practice multiple colors are likely to be retrieved from \(O\) due to the process of anti-aliasing. Anti-aliasing is used to smooth out the jagged edges of curves in text and shapes by
adding gradient colors between the curve line and the background. An example of this is shown in Figure 7.3, where shades of pink are added to smooth out the red text to the white background.

To address this challenge and find the expected color value, my insight is that the color occurring with the highest frequency in the area of $e$ in $O$ is likely to be the expected correct color, with the remaining colors likely reported due to anti-aliasing. Based on this insight, $G\text{Fix}$ first builds a color histogram, $H_o$, for $O$ by extracting the color value for every pixel in the area of $e$. Similarly, a color histogram, $H_t$, is built for $e$ by extracting the color values from the screenshot of the $PUT$. Then $G\text{Fix}$ calculates the relative complement $H_t$ in $H_o$ and stores all of the unmatched colors, i.e., colors that are present in $O$ but not in the $PUT$, in a new histogram, $H$. The histogram $H$ is then sorted in descending order of frequency. This ranked list of colors is then searched systematically to find the fix value, $v'$, that reports a minimum fitness score and returned to the main algorithm (line 19 of Algorithm 3).

Consider the running example for understanding the working of the color analysis. The difference area $B$ in Figure 7.2c is processed for color. The oracle (Figure 7.2a) requires the text color to be red (#FF0000) in that region, but the $PUT$ (Figure 7.2b) has the text color as black (#000000). The approach first builds the color histograms, $H_o = \{(#FFAFFF, 1000), (#FF55AA, 500), ..., (#FF0000, 5500)\}$ and $H_t = \{(#566573, 1000), (#D6DBDF, 500), ..., (#000000, 5500)\}$. The relative complement histogram $H$ after sorting is obtained as \{(#FF0000, 5500), (#FFAFFF, 1000), ..., (#FF55AA, 500)\}. The first value from the ranked list chosen by the approach, #FF0000, reduces the number of difference pixels in the area $B$ to 0, indicating that we have found the correct candidate fix value.
7.2.2.5 Predefined Values Analysis

The predefined values analysis finds candidate fixes for the CSS properties in the category PredefinedValuesProperties (Table 7.2), which includes properties such as font-style, display, and font-family ("predefinedValuesAnalysis" at line 21 of Algorithm 3). The CSS properties in this category can take a value from a set of discrete predefined values. For example, font-style can take a value from the predefined set {italic, oblique, normal}. A guided search cannot be used for selecting a candidate value in this analysis as there is no clear relationship between the different discrete values in the set. Therefore, I process this category using an exhaustive exploration of all of the predefined values for a given CSS property. In general, exploring all possible values can be expensive, however, that expense is mitigated by the fact that W3C defines only small sets of possible values for the CSS properties in this category. The average size of these sets is 5, with the largest set containing only 21 elements. The predefined value with the best fitness score from the exhaustive search is returned as the candidate fix value, v' (line 21 of Algorithm 3).

Consider the running example for understanding the working of the predefined values analysis. The difference area C in Figure 7.2c is processed for predefined values. The oracle (Figure 7.2a) expects the font-style to be normal, but the test web page (Figure 7.2b) has the font-style as italic. Consider the faulty element identified for this visual difference as \langle span style="font-style: italic;" \rangle. GFix explores all the values from the predefined set of values {italic, oblique, normal} for the font-style property, and eventually observes that with the value normal, the difference pixels in the area C are reduced to 0. This indicates that the correct candidate fix value has been identified.

7.3 Evaluation

I conducted an empirical evaluation to determine the accuracy and analysis time of GFix for MDDPs and RDPs in web applications. The specific research questions that were considered are as follows:

RQ1: How effective is GFix in reducing MDDPs and RDPs in web pages?

RQ2: How long does GFix take to find repairs?

RQ3: What is the quality of the fixes generated by GFix?
Table 7.3: Subjects used in the evaluation of $\mathcal{G}$Fix

<table>
<thead>
<tr>
<th>Name</th>
<th>URL</th>
<th>#HTML</th>
<th>#CSS</th>
<th>#T</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RDPs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td><a href="http://dbi.perl.org">http://dbi.perl.org</a></td>
<td>269</td>
<td>1,771</td>
<td>30</td>
</tr>
<tr>
<td>GTK</td>
<td><a href="http://www.gtk.org">http://www.gtk.org</a></td>
<td>75</td>
<td>1,334</td>
<td>28</td>
</tr>
<tr>
<td>Konqueror</td>
<td><a href="http://konqueror.org">http://konqueror.org</a></td>
<td>247</td>
<td>7,240</td>
<td>24</td>
</tr>
<tr>
<td>Amulet</td>
<td><a href="http://www.cs.cmu.edu/~amulet">http://www.cs.cmu.edu/~amulet</a></td>
<td>98</td>
<td>94</td>
<td>21</td>
</tr>
<tr>
<td>UCF</td>
<td><a href="http://www.ucf.edu">http://www.ucf.edu</a></td>
<td>315</td>
<td>2,414</td>
<td>29</td>
</tr>
<tr>
<td><strong>MDDPs</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gmail</td>
<td><a href="http://www.gmail.com">http://www.gmail.com</a></td>
<td>75</td>
<td>310</td>
<td>32</td>
</tr>
<tr>
<td>USC CS Research</td>
<td><a href="https://tinyurl.com/y9me423d">https://tinyurl.com/y9me423d</a></td>
<td>771</td>
<td>5,107</td>
<td>32</td>
</tr>
<tr>
<td>Craigslist</td>
<td><a href="http://losangeles.craigslislist.org">http://losangeles.craigslislist.org</a></td>
<td>1,145</td>
<td>31,356</td>
<td>28</td>
</tr>
<tr>
<td>Java Tutorial</td>
<td><a href="https://tinyurl.com/y8nrlwum">https://tinyurl.com/y8nrlwum</a></td>
<td>160</td>
<td>308</td>
<td>31</td>
</tr>
<tr>
<td>Virgin America</td>
<td><a href="http://www.virginamerica.com">http://www.virginamerica.com</a></td>
<td>396</td>
<td>3,730</td>
<td>25</td>
</tr>
</tbody>
</table>

### 7.3.1 Implementation

For the purpose of the evaluation, I implemented the approach in prototype tool, $\mathcal{G}$Fix, in Java. To compute the different visual symptoms listed in Table 7.1, $\mathcal{G}$Fix uses Selenium WebDriver for making dynamic changes to web pages, such as applying candidate fix values, to take screenshots of the web pages, and extract bounding rectangle information of the HTML elements. jStyleParser is used to extract the different CSS properties defined for an HTML element. $\mathcal{G}$Fix uses OpenCV to compare the screenshots, extract color information, crop screenshots, and perform sub-image searching. For stage 3 of the approach, the number of candidate repairs, $n$, is set to 50. For the input functions $D$ and $L$, I used the WebSee tool [103, 104], which provides as output a set of difference pixels identified by perceptual image differencing and a ranked list of suspicious HTML elements. I parallelized the search process (stage 2 and 3) using four threads on a 4th Generation Intel Core i7-4770 Processor with 32GB RAM. All of the experiments were run on an Ubuntu 14.04 platform. For rendering the subject web pages, I used Mozilla Firefox v46.0.01 with the browser window maximized to the screen size.

### 7.3.2 Subjects

Table 7.3 shows the ten subject web pages selected for the evaluation of $\mathcal{G}$Fix, five for each of the two types of presentation failures, i.e., RDPs and MDDPs. The columns labeled “#HTML” and “#CSS” report the total number of HTML elements present in the DOM tree of a subject, and the total number of CSS properties defined for the HTML elements in the page respectively. These metrics of size give an estimate of a page’s complexity in debugging and finding potential fixes for the observed presentation failures.
7.3.2.1 Selection of subjects for RDPs

In the evaluation, I focused on one scenario in which regression debugging might be needed, refactoring web pages. The goal of web page refactoring is to change the structure or HTML code of a page without altering its visual appearance. I chose refactoring because it is a topic that has received a lot of attention in the research and development community (e.g., [81]) and as a result there are objective and independently defined processes for how to refactor for certain objectives. Based on a literature review, I chose to focus on three refactoring techniques that were widely mentioned and had clearly defined processes. The first refactoring is to migrate HTML 4 to HTML 5. For example, converting `<div id="header">` into `<header>`. The W3C provides official guidelines for converting a typical HTML 4 web page into an HTML 5 page [32]; The second refactoring is to convert a table based layout to a div based layout. Related work [108] provides a set of best practices to be used in this conversion that will maintain the original appearance of the page. The third refactoring is to replace deprecated tags. For example `<font>` is deprecated in favor of using the CSS `font` property. Here again, the W3C provides a complete list of the deprecated HTML tags [31].

My choice of refactorings guided the selection of subject pages. The initial starting point was to gather URLs from an online random URL generator (http://www.uroulette.com). I then filtered this set to include only web pages to which all three refactorings could apply. The resulting set of subjects is shown in Table 7.3. For each of these subjects, I created a refactored version with all three refactorings applied. I performed the refactorings manually, but followed the instructions provided for each to make the process objective and repeatable. To confirm that the visual appearance was unchanged, I used an image differencing tool to ensure that there were zero difference pixels between the original and refactored version. On average, refactoring and verifying each subject took 5-10 hours depending on its complexity.

7.3.2.2 Selection of subjects for MDDPs

For the experiments, I utilized the five subject web pages shown in Table 7.3 for repairing MDDPs. I chose these web pages because they represent a mix of different implementation technologies and layouts that are widely used across all web applications. In particular, I chose the set of test subjects to include web pages that were defined by statically generated HTML, CSS, and JavaScript and pages defined by dynamically generated HTML.
7.3.2.3 Test case generation

To create test cases for the evaluation, I used a random seeding based approach that inserted presentation failures into the subject web pages. I used this approach because I was not able to obtain a sufficiently large enough set of mockups and refactoring errors in real-world web applications in order to ensure that GFix could be validated against a wide range of MDDPs and RDPs.

To create test cases for the evaluation, I seeded presentation failures into the each subject web page. Note that for RDPs the seeding was performed on the refactored versions of each subject. We used the following process for each subject page $p$: (1) download $p$ and all files required to display it; (2) take an image capture of $p$ to serve as the oracle $O$; and (3) create a set $P'$ that contains variants of $p$, each variant created by randomly seeding a unique presentation fault. To identify the types of faults to seed, I first manually analyzed the W3C HTML and CSS specifications to identify visual properties — HTML attributes or CSS properties that could change the visual appearance of an element. I seeded faults by changing the original value of each unique visual property present in $p$. I eliminated any variant of $p$ with a seeded presentation fault that did not actually produce a presentation failure. To identify these, I computed the set of pixel differences between the rendering of $p$ and $O$, and only included the variant if this set’s size was non-zero. The visual impact of a seeded fault varied, making the test cases vary in complexity for GFix. In some cases, the seeded fault caused almost all of a page to be shown as having a pixel-level difference. For example, changing the value of the padding CSS property in the ⟨body⟩ tag. In other cases, the seeded fault impacted only a small area (e.g., changing the text color of a ⟨span⟩ tag).

Each test case was comprised of an appearance oracle ($O$) and the page with a seeded fault (variant in $P'$). The number of test cases generated for each subject is shown under the column “#T” in Table 7.3.

7.3.3 Experiment One

To address RQ1 and RQ2, I ran GFix five times on each of the subjects to mitigate the non-determinism in the search. For RQ1, I used WebSee to determine the initial number of MDDPs and RDPs, represented as visual difference pixels, in a test case and the average number of difference pixels remaining after each of the five runs of GFix. From these numbers I calculated the reduction of MDDPs and RDPs as a percentage. For RQ2, I collected the median total running
times of $\mathcal{G}$Fix and the median time required for the three types of search analyses, namely, size and position analysis, color analysis, and predefined values analysis.

### 7.3.3.1 Presentation of Results

Table 7.4 shows the results of RQ1. The average initial number of MDDPs and RDPs across each test case of a subject are shown under the column “Avg. #Before”. The column, “Avg. #After” shows the average number of MDDPs and RDPs remaining after the five runs of each of the test cases of a subject. The average percentage reduction in the number of MDDPs and RDPs for each of the test cases of a subject are shown under the column “Reduction”.

Table 7.5 shows the results of RQ2. The columns show the median running time per root cause for the different analyses and the total running time across each of the test cases of a subject.
7.3.3.2 Discussion of Results

The results show that $G\text{Fix}$ reported an average 94% reduction in MDDPs and RDPs. This shows that $G\text{Fix}$ was effective in finding fixes for MDDPs and RDPs. Of the total 280 test cases across all subjects, $G\text{Fix}$ was able to reduce the number of difference pixels to zero, i.e., resolve all of the reported MDDPs and RDPs for 89% of the test cases.

I investigated the results to understand why $G\text{Fix}$ was not able to find suitable fixes for all of the MDDPs and RDPs. I found two dominant reasons for this. First, WebSee’s perceptual image differencing used in the fitness function was not sensitive to small delta changes tried by the AVM search in the size and position analysis. This behavior was particularly observed if the test case had a significant amount of distortion seeded in it, i.e., if the test case had a high amount of pixel-level differences with its oracle. In such cases, no improvement in the fitness score was observed, leaving the presentation failure unresolved. Second, the faulty CSS property that was required to be adjusted in order to repair the observed failure was not included in the set of relevant CSS properties identified by the visual symptoms.

The total running time of $G\text{Fix}$ ranged from five minutes to 55 minutes, with a median of 14 minutes. As the results of the analyses show, the predefined values analysis was the most time consuming, followed by size and position analysis and color analysis. This is an expected result, since predefined values analysis performs an exhaustive search over the set of predefined values. The total time of $G\text{Fix}$ was dependent on the size of the page and the number of potentially faulty HTML elements reported by WebSee. Despite the use of four threads for parallelization, the runtime for some subjects is lengthy. This can be further improved by using cloud instances for further parallelization, as has been achieved in related work [85].

7.3.4 Experiment Two

For addressing RQ3, I conducted a user study to understand the visual quality of $G\text{Fix}$’s suggested fixes from a human perspective. The general format of the survey was to present, in random order, a MDDP (or RDP) containing a UI snippet from a subject web page before and after repair. The participants were then asked to compare the appearance similarity of the two UI snippets with the corresponding UI snippet from the oracle. Each UI snippet showing a failure was captured in context of its surrounding region to allow participants to view the failure from a broader perspective. Examples of UI snippets are shown in Figure 7.4. To select the “after” version of a test case to display the best repair identified by $G\text{Fix}$, I used the run with the best fitness score across the five runs of $G\text{Fix}$ in Experiment One.
To make the survey length manageable for the participants, I selected a random test case from each subject and presented it in the survey, thus making the length of each survey 10 questions, one question for each subject. For each question, the survey presented, in random order, the UI snippets of the before and after versions of the selected test case and a corresponding reference UI snippet from the oracle. I conducted a total of five surveys.

I used Amazon Mechanical Turk (AMT) service to conduct the surveys. AMT allows users (requesters) to anonymously post jobs which it then matches them to anonymous users (workers) who are willing to complete those tasks to earn money. To avoid workers who had a track record of haphazardly completing tasks, we only allowed workers that had high approval ratings for their previously completed tasks (over 95%) and had completed more than 5,000 approved tasks to complete the survey. In general, this is considered a fairly selective criteria for participant selection on AMT. 10 anonymous participants undertook each survey, giving a total of 50 completed surveys and 500 data points. Each participant was paid $0.20 for completing a survey.
Figure 7.5: \( \text{GFix}'s \) user study results

### 7.3.4.1 Presentation of Results

The results for the appearance similarity preference given by the participants for each survey are shown in Figures 7.5a, 7.5b, 7.5c, 7.5d and 7.5e. The aggregated results from all of the five surveys are shown in Figure 7.5f. In each figure, the number of participants are shown on the Y-axis and the subjects are shown on the X-axis. The light gray bars indicate the participants reporting the after repair version as being more similar to the reference than the before. Similarly, the dark gray bars show the preference for the before repair version as being more similar to the reference than the after.

### 7.3.4.2 Discussion of Results

Based on the analysis of the results of the five surveys, I found that, overall, 79% of the participants reported the repaired (after) versions of the subjects as more similar to the reference than the unrepairod (before) versions. This indicates that \( \text{GFix} \) generates repairs for MDDPs and RDPs that makes the pages aesthetically similar to their oracles from a human perspective. Interestingly,
out of the 50 test cases presented in the surveys, 10 test cases received responses in favor of the
after version from all of the participants and in 30 test cases, the after version was rated as having
more similarity to the reference by at least eight participants.

As can be seen from Figure 7.5, all of the test cases expect two ("ucf" from survey 1 and
"konqueror" from survey 4) showed a majority agreement among the participants. I investigated
these test cases to understand the reason for high discordance among the participants. The
dominant reason I found was that both the test cases had failures that impacted only a small
area of the pages, likely causing both, the before and after versions of the test cases, to appear
the same. The participants could have likely missed noticing such small failures, resulting in a
mixed preference given for the versions and a majority disagreement among the participants. A
similar finding was observed for the test case “usc cs research” from survey 3, which in fact is the
only test case showing a majority of the participants in the favor of the faulty version over the
repaired version. Although the failure in this test case was resolved by $\text{GFX}$, perhaps it was was
not very noticeable to the participants.

7.3.5 Threats to Validity

In this section I discuss threats to the validity of my conclusions the results of $\text{GFX}$. A possible
threat is that I used a fault seeding mechanism for generating the test cases for RQ1 and RQ2
instead of real-world faults. This was unavoidable as I did not have access to a large enough source
of web pages with mockups, multiple versions, and refactorings. However, there are three factors
that minimize this threat. First, the fault seeding mechanism was based on mutation testing
techniques, which have been shown to produce useful and representative test suites. Second, my
prior work has shown that detection and localization results achieved on artificially generated test
suites are similar to those achieved when using real mockups [103]. Third, I used a systematic
seeding process based on the actual visual properties present in a page to guide the seeding process.
This was done to reduce unintentional bias in the definition of the test cases. The second threat
is that I controlled the changes introduced for simulating regression debugging for RDPs. To
mitigate this, I chose to carry out HTML page refactoring because there were externally defined
standards for how to perform the three different types of refactorings and I could follow these
approaches to minimize the unintentional introduction of bias.
7.4 Conclusion

In this chapter, I presented my novel search-based approach, $\mathcal{G}$Fix, for the automated repair of MDDPs and RDPs in web applications. The approach uses two phases of guided search to find the repair. The first phase uses the AVM search to identify candidate fixes, while the second phase uses a biased random search to find a subset of the candidate fixes that produces an overall best repair. The fitness function used to guide both of the search algorithms employs computer vision techniques to quantify the deviation between the appearance of a rendered page and its intended appearance, with a minimization goal. In the evaluation, $\mathcal{G}$Fix was able to reduce, on average, $94\%$ of the presentation failures reported by WebSee. In a user study assessing the visual similarity of the pages, the participants overwhelmingly reported the repaired versions as more similar to the reference than the original (faulty) versions. Overall, these are positive results and support the hypothesis of my dissertation by showing that this approach using search-based techniques can be highly effective in resolving MDDPs and RDPs in web pages.
Chapter 8

Related Work

In this chapter, I discuss related work for the repair of presentation failures in web applications. I also discuss different approaches in the literature that are related to the four techniques, $\mathcal{X}$Fix, $\mathcal{M}$Fix, $\mathcal{I}$Fix, and $\mathcal{G}$Fix.

8.1 Automated repair of software systems

Automated repair of software systems has been an area of active research. Many different approaches have been proposed that focus on repairing different aspects of the software systems, however, none of them are capable of repairing presentation failures in web applications. These approaches can be broadly divided into two categories as described below.

Techniques for repairing faults in software programs

There has been extensive work on the automated repair of software programs. Several techniques that use search-based algorithms have been proposed. Two examples include GenProg [159, 85], which uses genetic programming to find viable repairs for C programs, and SPR [91], which uses a staged repair strategy to search through a large space of candidate fixes. Alternative analytical approaches also exist, including FixWizard [127], which analyzes bug fixes in a piece of code and suggests fixes to similar parts of the code base; and FlowFixer [167], which repairs sequences of Graphical User Interface (GUI) interactions in modified test scripts for Java programs. However, these techniques are not capable of repairing presentation failures (e.g., Cross Browser Issues (XBIs), Mobile Friendly Problems (MFPs), Internationalization Presentation Failures (IPFs), Mockup-driven Development Problems (MDDPs), and Regression Debugging Problems (RDPs)) in web applications because they are structured to work for general-purpose programming languages, such as Java and C.
Techniques for repairing faults in web applications

Recently, techniques to repair different types of faults in web applications have been proposed. These techniques deal with specific components of the client-side of web applications and as such are not meant for repairing presentation failures in web applications. PhpRepair [142] and PhpSync [126], focus on repairing problems arising from malformed HTML. Although these techniques can help resolve a certain class of presentation failures, i.e., problems caused by HTML syntax errors, they cannot repair presentation failures such as the ones discussed in Chapter 2, as they are not caused by malformed HTML. Another technique [158] assumes that an HTML/CSS fix has been found and focuses on propagating it to the server-side using hybrid analysis. Cassius [131], proposes a framework for repairing faulty CSS in web applications by assuming the availability of a set of faulty source lines in CSS files and HTML page layout examples that the technique can use to synthesize repair. This technique uses the CSS from the page layout examples as the oracle to identify the fix values for the faulty CSS. In general, HTML page layout examples may not be available, for example, the intended layout of a page is available in the form of an image in the case of MDDPs and RDPs. Also, in the case of XBIs, MFPs, and IPFs, the oracle and the page under test (PUT) use the same CSS files. Therefore this technique is generally not applicable for repairing the types of presentation failures described in Chapter 2. A technique, ARROW [156], uses static analysis to repair client-side race conditions that are caused by the concurrent and asynchronous rendering of Javascript, CSS, and images during a page load. This technique corrects the ordering of Javascript in web pages to avoid the race conditions and can thereby repair presentation failures caused by such event races but not XBIs, MFPs, IPFs, MDDPs, and RDPs.

8.2 Cross Browser Issues (XBIs)

Cross Browser Testing (XBT) techniques, such as X-PERT [137, 140, 59, 141], CrossT [118], Browserbite [144], Browsera [12], Webmate [62, 63], and work by Eaton and Memon [66], are effective in detecting XBIs. However, repairing the reported XBIs when using these techniques must still be performed manually. Crossfire [61] presents a protocol for XBI debugging by extending browser developer tools, such as Firefox’s Firebug, to enable cross-browser support. However, the task of using the debugger to find potential fixes is developer-driven. Another technique, FMAP [138], analyzes the traces of client-server communication of a web application on different platforms (e.g., desktop and mobile) to detect unmatched functional and behavioral features. Thus, the problem addressed by FMAP is fundamentally different than XFix as the User Interface (UI) of a web application is expected to be substantially dissimilar on different platforms.
A common resource used by web developers to reduce possible XBIs is to use simple CSS resetting techniques, such as Normalize CSS [72] and YUI 3 CSS Reset [8]. Such techniques establish a consistent CSS baseline for different browsers to minimize the browser differences that can lead to XBIs. For example, Chrome defines the default width of an input/text box as 155px while Firefox defines it as 184px, which can likely lead to an XBI. The CSS reset techniques overwrite such browser specific default CSS values with standard values (e.g., width = 160px) defined in the CSS reset files to potentially reduce cross-browser differences. Such techniques, however, cannot handle complex XBIs that are application dependent, are caused by unsupported CSS properties/values, and are caused by complex interaction between HTML and CSS. For example, Internet Explorer (IE) does not support the value initial for the CSS property left, leading to an unpredictable behavior of the HTML elements in the PUT containing this property-value pair. This can likely introduce layout XBIs in the PUT as the positioning of such elements is application dependent, i.e., based on the styling and layout of their surrounding elements, requiring such XBIs to resolved on a case by case basis.

### 8.3 Mobile Friendly Problems (MFPs)

There are approaches that circumvent MFPs by presenting alternative versions of a desktop website, rather than repairing MFPs in the website. For example, commercial services such as bMobilized [14], WompMobile [45], Mobilifyit [36], Duda [20], and Mobify [34], can convert a given desktop website to a mobile friendly version using pre-designed templates. Although helpful, these solutions are not appropriate in all situations. Firstly, the templates are unlikely to capture the carefully crafted layout and graphics designed for the desktop versions, possibly undermining the branding efforts a company is trying to achieve. My approach, MFix, avoids these limitations by maintaining a close similarity to the original version. Second, the output represents a separate mobile friendly website with a new URL, requiring the development team to maintain two websites. In contrast, the MFix approach generates a CSS media query patch that is added to the existing CSS of the original website and that will only be triggered if the page is requested from a device with a smaller screen size. Alternatively, modern browsers, such as Chrome [17], Safari [40], and Firefox [23], provide a “reader” mode intended for easy clutter-free viewing of web pages on mobile devices by presenting only its text and stripping out layout and page styling. The primary purpose of this mode, however, is to allow for easier reading of a page’s primary content, rather than to address mobile friendly problems.
8.4 Internationalization Presentation Failures (IPFs)

Different techniques exist that target detection of internationalization failures in web applications. GWALI [49] and i18n checker [44] are automated techniques, while Apple’s pseudo-localization [11] requires manual checking to identify IPFs. There is also a group of techniques [52, 50, 132] that performs automated checks for identifying internationalization problems, such as corrupted text, inconsistent keyboard shortcuts, and incorrect/missing translations. However, none of the aforementioned techniques are capable of repairing IPFs.

Another technique related to internationalization in web pages is TranStrL [157]. It locates strings in web applications that developers need to translate before internationalization can be considered complete. Although this technique helps developers to carry out the internationalization process more thoroughly, it does not help developers repair when their translations have led to an IPF.

8.5 Mockup-driven Development Problems (MDDPs) and Regression Debugging Problems (RDPs)

There is a body of work related to my detection and localization approach, WebSee [103, 104, 102], that uses different differencing techniques to identify presentation problems in web applications.

DOM Differencing: Techniques based on textual differencing (e.g., diff) of the HTML source or DOM comparisons [141, 59, 137, 118, 37, 147, 146, 161, 76, 75] are not applicable for detecting MDDPs and are of limited use for RDPs. There are two problems with these techniques. First, only a textual difference between two pages does not necessarily imply that there is a visual difference. This is because (1) there are often several ways to implement the styling of HTML elements to make them appear the same (e.g., margin and padding can be used interchangeably to achieve the same visual effect or setting the CSS property of font-size to ‘small’ will have the same visual effect as setting it to 13px) or (2) a page may have been restructured in a way that did not translate into a visual difference (e.g., when a \(<\text{table}\>) is converted to a table-less layout with \(<\text{div}\>) tags). Second, the lack of a textual difference does not imply that there are no presentation failures. For example, a failure can occur without any textual change to an \(<\text{img}\>) tag if the tag does not specify size attributes and the dimensions of the image file change on disk.

Invariant specification techniques: A group of techniques allows developers to specify invariants that will be checked and enforced on a web page. These include Selenium [42], Sikuli [57, 165], Cucumber [19], Crawljax [119, 116, 120], and Cornipickle [78]. There are two
main disadvantages to this type of invariant specification approach compared to WebSee. First, these techniques require testers to exhaustively specify every correctness property to be checked, which may be very labor intensive. Second, the correctness properties are expressed in terms of HTML syntax, not the visual appearance of an HTML element. Therefore, these techniques may miss presentation failures caused by incorrect inheritance of an ancestor element’s CSS properties, for example.

Sikuli [57, 165] is an automation framework based on computer vision techniques that uses sub-image searching to identify and manipulate GUI controls in a web page. Although not intended for verification, one could provide a set of screenshots of each GUI element and use Sikuli to ensure that they match (i.e., there are no presentation failures). However, since Sikuli uses a sub-image based search of the page, it could match the provided screenshots against any portion of the page, not necessarily the intended region. This means it would be ineffective if there were visually identical elements in the page. Furthermore, Sikuli only provides an element after a positive match; therefore when there is a failure, no match will be made and no element(s) will be provided to the testers to help with localization.

**Automated Oracles:** Work by Sprenkle and colleagues proposes a suite of HTML-based automated oracle comparators to detect presentation failures in the current version of a web application compared to its previous version [147, 146]. Therefore these techniques cannot be used for detecting MDDPs, where there is no prior working version of the web application that can be compared against. Another work facilitates the use of a graphical oracle to detect presentation problems [64]. However, this technique requires the testers to specify the characteristics that should be used to compare the images of the oracle and the page under test and then implement them in the form of extractors. The tester is also expected to implement the similarity function used to determine how close in appearance the two images are.

**Visual Regression Testing:** Tools, “Wraith” [46] and “PhantomCSS” [38], are developed by BBC News and Huddle, respectively, for helping developers with visual/CSS regression testing. Both of these tools use pixel-level screenshot comparison to identify the differences between test and baseline pages, and report the pixel-level image differences to the developers. These tools have several shortcomings limiting their applicability in debugging presentation failures in web applications. First, the false positive rate in detecting presentation problems is high owing to the strict pixel-to-pixel equivalence comparison to identify differences. Small differences that represent concessions to coding simplicity or failures that are within a level of tolerance that the development team does not consider to be a presentation failure are reported. Second, the tools do not facilitate the handling of the dynamic portions of pages. Third, the tools only report
image-level differences and do not identify the faulty HTML elements. WebSee addresses all of
the above limitations by using computer-vision techniques to report only the human perceptible
differences, providing special regions handling for dynamic portions of pages, and localizing the
presentation failures to HTML elements in the page by using rendering maps.

**Browser Solutions:** Browser plug-ins, such as “PerfectPixel” [160] for Chrome and “Pixel
Perfect” [121] for Firefox help developers to detect pixel-level differences with an image based
oracle. They overlay a semitransparent version of the oracle over the HTML page under test,
enabling developers to do per pixel comparison to detect presentation failures. However, they re-
quire the developer to manually locate the faulty elements. In contrast, WebSee is fully automated
for detection and localization.

**Visual slicing:** Recent work in the field of program slicing of picture description languages
(e.g., postscript) also uses a visual-based analysis [166, 54]. These techniques capture the seman-
tics of picture description languages through visual differences. However, such techniques address
a fundamentally different problem of slicing non-traditional programming languages rather than
detecting presentation failures in web pages.

### 8.6 Other presentation failure detection techniques

There exist techniques for detecting other types of presentation failures besides the ones listed
above. The ReDeCheck technique [155, 153, 154] uses a layout graph to find regression failures
in responsive web pages that adjust their layout according to the size of the browser’s viewport.
Another technique, “Fighting Layout Bugs” [149] can be used to automatically find application
agnostic presentation failures, such as overlapping text and unreadable text. However, both the
techniques can only detect presentation failures, and are not able to repair them.

### 8.7 Testing and analysis of web app client-side components

Techniques to test JavaScript [51, 129, 128, 83] and analyze CSS [117] have been proposed recently.
These techniques deal with specific components of the client side and as such are not meant for
repairing presentation failures in a web application. Another technique does impact analysis of
CSS changes across a website [86], and notifies the developer if changes made in a CSS file are
introducing new presentation failures in other web pages of the website. However, this technique
is not capable of repairing presentation failures.
There exist several techniques [112, 109, 110, 111, 150, 134, 135, 125] that use program slicing to extract features or behaviors of a web application for code reuse and debugging. However, these techniques cannot repair presentation failures in web applications.

### 8.8 GUI Testing

Memon and colleagues [148, 115, 162, 122, 124] have done extensive work in the area of model-based GUI testing. These techniques test the behavior of a software system by triggering event sequences from the GUI. The purpose of their work is not fixing presentation issues in the GUI, but rather using the GUI as a driver to find behavioral problems in the system.
Chapter 9

Conclusion and Future Work

In this chapter, I conclude by giving a summary of my dissertation work and discuss directions of future work.

9.1 Summary

Web applications have become an important part of our daily lives for performing both professional and personal activities, such as shopping, banking, networking, and email. We can very conveniently access websites from a range of different browsers that run on a variety of different platforms and devices, and render the sites in a language of our choice. Although this model is very convenient for the end-users, it is extremely challenging for developers to ensure that a website renders consistently across the wide range of browsers, that the website is as user friendly on a mobile device as it is on a desktop device, and that the website adapts its layout gracefully for all of the different languages. The inability to do so can result in the website having User Interface (UI) issues, such as Cross Browser Issues (XBIs), Mobile Friendly Problems (MFPs), Internationalization Presentation Failures (IPFs), Mockup-driven Development Problems (MDDPs), and Regression Debugging Problems (RDPs). I collectively refer to such UI issues as presentation failures — a discrepancy between the actual appearance of a website and its intended appearance — that can degrade the aesthetics of a website and affect its usability and functionality likely resulting in a frustrating and poor user experience. Despite the importance of presentation failures, there exist no techniques for their automated repair, making it a manual task that is labor-intensive and requires significant expertise.

To address these limitations, the goal of my research is to automate the process of repairing presentation failures in web applications. My dissertation work furthers this goal by developing
different techniques for the automated repair of different types of presentation failures in web applications, such as XBIs, MFPs, and IPFs. The hypothesis statement of my dissertation is:

*Search-based techniques can repair presentation failures in a web page with high effectiveness.*

To evaluate the hypothesis of my dissertation, I designed and developed four approaches for repairing different types of presentation failures in web applications, namely, XBIs, MFPs, IPFs, MDDPs, and RDPs. All of my four repair approaches were designed using search-based techniques. The effectiveness of my approaches in repairing presentation failures was evaluated by measuring the reduction in the number of presentation failures reported by existing detection techniques in the before and after repair versions of the pages. I also conducted user studies to measure the effectiveness of the repair approaches in reducing the human-observable presentation failures and understand the impact of the generated repairs on the aesthetic quality of the pages from a human perspective.

The first approach is $\lambda$Fix, which targets the repair of layout XBIs in web pages. $\lambda$Fix uses two phases of guided search. The first phase finds candidate fixes for each of the root causes identified for an XBI. The second phase then finds a subset of the candidate fixes that together minimizes the number of XBIs in the web page. The empirical evaluation of $\lambda$Fix on 15 real world web pages showed that it was able to resolve 86% of the XBI reported by X-PERT, a well-known XBI detection tool, and 99% of the XBIs observed by humans. In a user study assessing the improvement in consistency between the repaired and reference page, 78% of the participant ratings reported an improvement in the cross-browser consistency of the repaired web pages.

The second approach, $\mathcal{M}$Fix, targets the repair of MFPs in web pages. $\mathcal{M}$Fix first segments the page into areas that form natural visual groupings. It then builds graph-based models of the segments and layout of the page and uses the constraints represented by these graphs to compute a repair that can improve mobile friendliness while minimizing layout disruption. In the evaluation, $\mathcal{M}$Fix was successfully able to resolve MFPs for 95% of the subjects. In a conducted user study, the participants overwhelmingly preferred the repaired version of the website for use on mobile devices, and also considered the repaired page to be significantly more readable than the original.

The third approach, $\mathcal{I}$Fix, targets the repair of IPFs in web pages. $\mathcal{I}$Fix first uses a clustering technique that identifies groupings of elements that are stylistically similar and adjusts them together in order to maintain the visual consistency of the page. It then uses a guided search-based technique that quantifies the amount of distortion in a page by leveraging existing IPF detection techniques and UI change metrics. In the evaluation, $\mathcal{I}$Fix was able to successfully resolve 98% of the reported IPFs in the subjects. In a user study of the repaired web pages, the
repairs met with high user approval, with over 70% of the user responses rating the visual quality of the fixed pages as significantly higher than the unfixed versions.

The fourth and final approach, $\mathcal{G}$Fix, targets the repair of MDDPs and RDPs in web pages. $\mathcal{G}$Fix uses guided search-based techniques to automatically find repairs for the detected MDDPs and RDPs in web pages. As its fitness function, $\mathcal{G}$Fix uses computer-vision techniques to quantify the amount of visual differences between the actual appearance of a web page and its intended appearance. In the evaluation of $\mathcal{G}$Fix on a set of real-world subjects, I found that the approach was able to accurately identify repairs for the failures and met with a high user approval rate.

Overall, the four approaches have demonstrated high effectiveness in repairing different types of presentation failures in web pages while maintaining or enhancing the visual appeal of the pages, thereby confirming the hypothesis of my dissertation. To the best of my knowledge, my research is the first automated approach for generating repairs for presentation failures, and the first to apply search-based repair techniques to web pages.

### 9.2 Future Work

In the future, addressing presentation failures will continue to be an important problem for developers not just in the domain of web applications, but in other domains as well, such as mobile applications. My dissertation identifies several real-world challenges in the domain of presentation failures that motivates this direction of research and also lays the foundation for developing automated techniques that can repair presentation failures in different software applications.

One possible direction of future work is to design approaches using search-based techniques for repairing more types of presentation failures in web applications. One such example of presentation failures is accessibility issues. Web accessibility is important as it allows inclusion of people with disabilities in the current technology-savvy society and enables them to independently perform work and personal activities, such as shopping and banking. An approach can be designed for the repair of accessibility issues by identifying detection techniques to quantify their impact and designing a suitable search-based algorithm to identify successful fixes. Another example of presentation failures that can be repaired using search-based techniques is responsive web design problems. Responsive web design is a paradigm that allows developers to design web pages that dynamically adapt their layout to different device sizes. The repair of presentation problems caused by responsive web design would pose new research challenges. For example, analysis to identify which lists of hyperlinks could be grouped into a drop down menu, a refactoring to carry out this change, and a method to quantify the change’s impact.
Another direction of future work is to retarget my developed repair approaches to areas other than web applications, particularly native mobile apps and apps on IoT (Internet of Things) devices with a Graphical User Interface (GUI), such as smartwatch, thermostat, and gaming systems. This would pose new research challenges, such as analyzing the characteristics of the presentation failures in these domains, a method to quantify the impact of the failures, and ways to carry out their repair.

Search-based techniques employed by my repair approaches are non-deterministic in nature, requiring the approach to be run multiple times and selecting the best repair. Another direction of future work can be to explore deterministic algorithms for repairing presentation failures in web applications. One possible way is to model the layout of web pages as a system of constraints and find suitable repairs using constraint solving. Another possible way is to mine large web app repositories to identify repair patches.
References


